The Compiler Design Handbook

Optimizations and Machine Code Generation

Edited by

Y.N. Srikant
P. Shankar

CRC PRESS

The Compiler Design Handbook

Optimizations and Machine Code Generation

The Compiler Design Handbook

Optimizations and Machine Code Generation

Edited by

Y.N. Srikant

P. Shankar

CRC PRESS

Boca Raton London New York Washington, D.C.

Library of Congress Cataloging-in-Publication Data

Srikant, Y.N.
The compiler design handbook : optimizations and machine code generation / Y.N.
Srikant, Priti Shankar
p. cm.
Includes bibliographical references and index.
ISBN 0-8493-1240-X (alk. paper)
1. Compilers (Computer programs). 2. Code generators. I. Shankar, Priti. II. Title.
QA76.76C65 S65 2002
005.4′53—dc21 2002067413

International Standard Book Number 0-8493-1240-X
Library of Congress Card Number 2002067413
Printed in the United States of America 1 2 3 4 5 6 7 8 9 0
Printed on acid-free paper

Preface

In the last few years, changes have had a profound influence on the problems addressed by compiler designers. First, the proliferation in machine architectures has necessitated the fine-tuning of compiler back ends to exploit features such as multiple memory banks, pipelines and clustered architectures. These features provide potential for compilers to play a vital role in the drive for improved performance. Areas in which such possibilities can be exploited are, for example, speculative execution, software pipelining, instruction scheduling and dynamic compilation. Another trend that is continuously explored is the generation of parts of the compiler from specifications. Such techniques are well established for the front end, but still in the experimental stages for the problems of code optimization and code generation. Optimizers in compilers are relying on newer intermediate forms like static single assignment (SSA) form. The growing use of object-oriented languages has brought into focus the need for optimizations that are special to such languages. Algorithms for program slicing and shape analysis have made it possible to gather detailed information about a program that can be used in tasks such as debugging, testing, scheduling and parallelization.

The Compiler Design Handbook: Optimizations and Machine Code Generation addresses some of these issues listed that are of central concern to compiler designers. However, it does not attempt to be an encyclopedia of optimization and code generation. Several topics could not be included mainly because of space limitations. Notable among omissions are just-in-time (JIT) compilation, implementation of parallel languages, interpreters, intermediate languages, compilation of hardware description languages, optimization for memory hierarchies, constraint programming languages and their compilation, functional programming languages and their compilation, code generation using rewriting systems, peephole optimizers and runtime organizations. The topics included are of relevance mainly to the compilation of imperative languages.

The material presented in this volume is addressed to readers having a sound background in traditional compiler design. In particular, we assume that the reader is familiar with parsing, semantic analysis, intermediate code generation, elements of data flow analysis, code improvement techniques and principles of machine code generation. We hope that the material presented here can be useful to graduate students, compiler designers in industry and researchers in the area.

Acknowledgments

The patience and efforts of the contributors are the major factors that have made this handbook possible. We thank each one of them for their contribution and for sparing their time to write a chapter. All of them have responded promptly to our requests and have readily cooperated in reviewing other contributors' chapters. We thank Ramesh Subrahmanyam, Uday Khedker, Vineeth Kumar, Wei Qin, S. Rajagopalan, Sanjay Rajopadhye, Mahmut Kandemir, Evelyn Duesterwald, Rajiv Gupta, Rajib Mall, R. Venugopal, Ravindra B. Keskar, Eduard Mehofer, Sanjeev Kumar Aggarwal, Reinhard Wilhelm, Thomas Reps, Andreas Krall, Nigel Horspool, Vicki H. Allan, Matthew T. Jacob, D.V. Ravindra and S.R. Prakash for reviewing the chapters. The assistance and continuous cooperation of the editors and other staff of CRC Press have proved to be invaluable in producing this text.

Editors

Y.N. Srikant is a Professor at the Indian Institute of Science (IIS), Bangalore, India, and is currently the chairman of the department of computer science and automation. He received his Ph.D. in computer science from the Indian Institute of Science, and is the recipient of young scientist medal of the Indian National Science Academy. He joined IIS in 1987 as a faculty member. Since then, he has guided a number of doctoral and master degree students and has consulted for a large number of industries. His areas of interest are compiler design, and application of compiler technology to the development of software tools used in software architecture design and software testing.

He started the compiler laboratory in the department of computer science and automation of IIS, where several major research projects in the following areas have been carried out in the last few years: automatic machine code generation, specification and generation of compiler optimizations, parallelization and HPF compilation, incremental compilation and just-in-time (JIT) compilation of Java. Some of the research projects currently in progress are implementation of .NET CLR on Linux, implementation of UDDI on Linux, code generation for embedded processors and optimizations for the Itanium processor.

Priti Shankar is at the department of computer science and automation at the Indian Institute of Science (IIS) where she is currently professor. She received a BE in Electrical Engineering at the Indian Institute of Technology (IIT), Delhi in 1968, and the M.S. and Ph.D. degrees at the University of Maryland at College Park in 1971 and 1972, respectively. She joined the IIS in 1973. She has guided a large number of research students in the areas of formal languages, tools for automatic compiler generation and code optimization, and algorithms for pattern matching and image processing. Several tools for automatic code generation have been designed and implemented under her supervision. Her principal interests are in the applications of formal techniques in the design of software tools and in algebraic coding theory. She is a member of the American Mathematical Society and the Association for Computing Machinery.

Contents

1

Data Flow Analysis

Uday P. Khedker
Indian Institute of Technology
(IIT) Bombay

1.1 How to Read This Chapter

This chapter provides a thorough treatment of data flow analysis for imperative languages. Because our focus is analysis instead of applications, we describe optimizations very briefly. Besides, we do not cover high level data flow analyses of declarative languages, static single assignment (SSA) based data flow analyses and intermediate representations; the latter two are covered in separate chapters.

0-8493-1240-7/03/$0.00+$1.50

We distinguish between what data flow analysis is and how data flow analysis is performed by delaying the latter as much as possible; we believe that this provides a much cleaner exposition of ideas. We begin from the first principles and evolve all basic concepts intuitively. In the early part of the chapter, rigor follows intuition until the peak is reached in Section 1.5 where the situation is completely reversed. We also provide the larger perspective of program analysis to highlight that data flow analysis may look very different from other program analyses but it is only a special case of a more general theme. This is achieved through a highly abstract summary in Section 1.2, which may be read cursorily in the first reading. Revisiting it after reading the rest of the chapter may bring out the relevance of this section better. Section 1.3 evolves the basic concepts of data flow analysis and it must be understood thoroughly before reading the rest of the chapter. Section 1.4 describes the expanding horizons of data flow analysis; the believers may skip it but the skeptics will do well to dissect it to appreciate the significance of the concept of information flow paths in Section 1.5.5. Sections 1.5.1, 1.5.2, and Figures 1.16 and 1.17 are very important and should not be skipped. Solution methods of data flow analysis are discussed in Section 1.6. Finally, the most important section of this chapter is Section 1.7; we urge the reader to make the most of it.

1.2 Program Analysis: The Larger Perspective

Language processors analyze programs to derive useful information about them. Syntactic analyses derive information about the structure of a program (which is typically static) whereas semantic analyses derive information about the properties of dynamic computations of a program. The semantic information can capture properties of both operations and data (together as well as independently).

1.2.1 Characteristics of Semantic Analysis

Semantic analyses cover a large spectrum of motivations, basic principles and methods. On the face of it, the existence of very many diverse analyses lead to the impression that there are more variances than similarities. Though this is true of practical implementations, at a conceptual level almost all semantic analyses are characterized by some common aspects. These characteristics are highly abstract, largely orthogonal and far from exhaustive. Although very elementary, they provide a good frame of reference; finding the coordinates of a given analysis on this frame of reference yields valuable insights about the analysis. However, a deeper understanding of the analysis would require exploring many more analysis-specific details.

1.2.1.1 Applications of Analysis

Typical uses of the information derived by semantic analyses can be broadly classified as:

- *Determining the validity of a program.* An analysis may be used to validate programs with regard to some desired properties (e.g., type correctness).
- *Understanding the behavior of a program.* An analysis may verify (or discover) useful properties of programs required for debugging, maintenance, verification, testing, etc.
- *Transforming a program.* Most analyses enable useful transformations to be performed on programs. These transformations include constructing one program representation from another as well as modifying a given program representation.
- *Enabling program execution.* Semantic analysis can also be used for determining the operations implied by a program so that the program can be executed (e.g., dynamic-type inferencing).

1.2.1.2 Theoretical Foundations of Analysis

Specification and computation of semantic properties can be based on any of the following:

- *Inference systems.* These systems consist of a set of axioms and inductive and compositional definitions constituting rules of inference. In such systems, the properties are inferred by repeatedly discovering the premises that are satisfied and by invoking appropriate rules of inference.
- *Constraint resolution systems.* These consist of a constraint store and a logic for solving constraints. In such systems, a program component constrains the semantic properties. These constraints are expressed in form of inequalities and the semantics properties are derived by finding a solution that satisfies all the constraints. The solution may have to satisfy some other desirable properties (namely, largest or smallest) too. [1] The constraints could be of the following types:
    - *Structured constraints.* These take advantage of the temporal or spatial structures of data and operations by grouping the related constraints together. Traditionally they have been unconditional, and are called *flow-based* constraints because they have been solved by traversals over trees or general graphs. Grouping of structured constraints often leads to replacing groups of related inequalities by equations. Structured constraints often lead to more efficient analyses, both in terms of time and space.
    - *Unstructured constraints.* These are not restricted by the structure and can capture more powerful semantics because they can be conditional. They often lead to comparatively less efficient analyses because the related constraints may not be grouped together.
- *Abstract interpretations.* These use abstraction functions to map the concrete domains of values to abstract domains, perform the computations on the abstract domains and use concretization functions to map the abstract values back to the concrete domains.
- *Other approaches.* These include *formal semantics*-based approaches (typically using *denotational semantics*), or *model checking*-based approaches, which are relatively less common.

These foundations are neither exclusive nor exhaustive. Though some foundations are preferred for some semantic analyses, in general, an analysis can be expressed in any of the preceding models.

1.2.1.3 Time of Performing Analysis

An analysis performed before the execution of a program is termed *static analysis*, whereas an analysis performed during the execution of a program (in an interleaved fashion) is termed *dynamic analysis*. Thus, an interpreter can perform static analysis (by analyzing a program just before execution) as well as dynamic analysis (by analyzing the program during execution). A compiler, however, can perform static analysis only.

In principle, the choice between static and dynamics analysis is governed by the availability of information on which the analysis depends, the amount of precision required (or the imprecision that can be tolerated), and the permissible runtime overheads.

An analysis that depends on runtime information is inherently dynamic. For example, if type annotations can be omitted in a language and type associations could change at runtime, types can be discovered only at runtime. This requires dynamic-type inferencing. If some amount of imprecision can be tolerated (e.g., if precise-type information is not expected but only sets of possible types are expected), it may be possible to perform an approximate static analysis for an otherwise inherently dynamic analysis. This obviates dynamic analysis only if a compromise on the precision

[1]The terms *largest* and *smallest* depend on the particular semantics derived and need to be defined appropriately.

of information is acceptable; otherwise it requires a subsequent dynamic analysis. In any case, it reduces the amount of dynamic analysis and hence runtime overheads.

If runtime overheads are a matter of concern, dynamic analyses should either be avoided or preceded by corresponding (approximate) static analyses. In practice, a majority of analyses performed by language processors are indeed static. In addition, many dynamic analyses have a static counterpart. For instance, many languages require array bounds to be checked at runtime; optimizing compilers can minimize these checks by a static array bound checking optimization.

1.2.1.4 Scope of Analysis

Semantic analyses try to discover summary information about some program unit by correlating the summary information discovered for smaller constituent units. As such, an analysis may be confined to a small unit of a program such as an expression or a statement, to larger units such as a group of statements or function and procedure blocks or to still larger units such as modules or entire programs. Program units with the same level of scope are analyzed independently. Analysis of units with differing levels of scope may be interleaved or may be nonoverlapped (and cascaded); in either case, the larger units can be analyzed only after their constituent units.

1.2.1.5 Granularity of Analysis

An exhaustive analysis derives information starting from scratch whereas an incremental analysis updates the previously derived semantic information to incorporate the effect of some changes in the programs. These changes may be caused by transformations (typically for optimization) or by user edits (typically in programming environments). In general, an incremental analysis must be preceded by at least one instance of the corresponding exhaustive analysis.

1.2.1.6 Program Representations Used for Analysis

An analysis is typically performed on an intermediate representation of the program. Though the theoretical discussions of many analyses are in terms of the source code, e.g., in the case of parallelization, in practice these analyses are performed on a suitable internal representation.

1.2.2 Characteristics of Data Flow Analysis

Data flow analysis discovers useful information about the flow of data (i.e., uses and definitions of data) in a program. We list the following characteristics of data flow analysis as a special case of semantic analysis:

- *Applications*. Data flow analysis can be used for:
  - Determining the semantic validity of a program, e.g., type correctness based on inferencing, prohibiting the use of uninitialized variables, etc.
  - Understanding the behavior of a program for debugging, maintenance, verification or testing
  - Transforming a program. This is the classical application of data flow analysis and data flow analysis was originally conceived in this context.
- *Foundations*. Data flow analysis uses constraint resolution systems based on structured constraints. These constraints are unconditional and traditionally related constraints are grouped together into data flow equations.
- *Time*. Data flow analysis is mostly static analysis.[2]
- *Scope*. Data flow analysis may be performed at almost all levels of scope in a program. Traditionally the following terms have been associated with data flow analysis for different scopes in the domain of imperative languages:

[2]Some form of dynamic data flow analysis for dynamic slicing, etc. is an active area of current research.

  - *Local data flow analysis*. Analysis across statements but confined to a maximal sequence of statements with no control transfer other than fall through (i.e., within a basic block).
  - *Global (intraprocedural) data flow analysis*. Analysis across basic blocks but confined to a function or procedure.
  - *Interprocedural data flow analysis*. Analysis across functions and procedures.

  It is also common to use the term local data flow analysis for analysis of a single statement and global data flow analysis for analysis across statements in a function and procedure. Effectively, the basic blocks for such analyses consist of a single statement.
- *Granularity*. Data flow analysis can have exhaustive as well as incremental versions. Incremental versions of data flow analysis are conceptually more difficult compared with exhaustive data flow analysis.
- *Program representations*. The possible internal representations for data flow analysis are abstract syntax trees (ASTs), directed acyclic graphs (DAGs), control flow graphs (CFGs), program flow graphs (PFGs), call graphs (CGs), program dependence graphs (PDGs), static single assignment (SSA) forms, etc. The most common representations for global data flow analysis are CFGs, PFGs, SSA and PDGs whereas interprocedural data flow analyses use a combination of CGs (and CFGs or PFGs). Though ASTs can and have been used for data flow analysis, they are not common because they do not exhibit control flow explicitly.

In this chapter, we restrict ourselves to CFGs.

1.3 Basic Concepts of Data Flow Analysis

This section is a gentle introduction to data flow analysis. We evolve the concepts through a running example contained in Figure 1.1(a), which is a stripped down version of a typical function in signal processing applications. The function `RunningTotal` calculates the running sum of energy at every instant in an interval `From` to `To` from the amplitude of the signal.[3]

1.3.1 Program Representation for Data Flow Analysis

Figure 1.1(b) contains the corresponding program in a typical linear intermediate representation generated by a compiler front end after scanning, parsing, declaration processing and type checking. We observe that the instructions in the intermediate representation correspond to the actual imperative statements of the C code; the declarative statements would populate the symbol table. The compiler has performed array address calculation using *Int_Size* to identify the position of an element from the start of the arrays. The instructions to access the location of an element in the memory (by adding the array base address) can be inserted by the code generator later. This is a matter of design choice. Finally, the most important observation is as follows:

> Expression $i * Int\_Size$ is computed on lines 5 and 16. *Int_Size* is a constant so it does not change; however, if we can ascertain that the value of i would not change during the time between execution of the instructions on line 5 and 16, then the computation of $i * Int\_Size$ on line 16 is redundant because it would compute the same value that would have been computed during the execution of the instruction on line 5.

[3]A digital signal processor (DSP) programmer may probably write some other kind of code but that is beside the point.

```
# define INSTANTS 10000
# define SFACTOR 0.8 /* Scaling Factor */
float Amplitude [INSTANTS], Energy [INSTANTS];

float RunningTotal (int From, int To)
{ float Sum;
  int i;
  i = From; Sum = 0;
  if (From < = To)
  { do
     {  Amp = Amplitude[i];
        if  (Amp  <  0)
           Sum = Sum − Amp * SFACTOR;
        else
           Sum = Sum + Amp * SFACTOR;
        Energy[i++] = Sum;
     }
     while (i <= To);
  }
  return Sum;
}
```

(a) The C code fragment

```
1.        i = From
2.        Sum = 0
3.        $t_0$ = (From $\leq$ To)
4.        **if not** $t_0$ **goto** $L0$
5. L1:    $t_1 = i \times Int\_Size$
6.        Amp = Amplitude[$t_1$]
7.        $t_2$ = Amp < 0
8.        **if** $t_2$ **goto** $L2$
9.        $t_3$ = Amp $\times$ 0.8
10.       $t_4$ = Sum + $t_3$
11.       Sum = $t_4$
12.       **goto** $L3$
13. L2:   $t_5$ = Amp $\times$ 0.8
14.       $t_6$ = Sum$-t_5$
15.       Sum = $t_6$
16. L3:   $t_7$ = i $\times$ $Int\_Size$
17.       Energy[$t_7$] = Sum
18.       $t_8$ = i + 1
19.       i = $t_8$
20.       $t_9$ = ($i \leq$ To)
21.       **if** $t_9$ **goto** $L1$
22. L0:   **return** Sum
```

(b) An intermediate representation

FIGURE 1.1 Calculating sum of energies of some signal at distinct instants.

The optimization that eliminates such redundant computations is called *common subexpression elimination* and it uses a data flow analysis called *available expressions analysis*.

Though it appears that the value of i is not modified at any time between the execution of lines 5 and 16, there are **gotos** and we need to ensure that we have covered all execution paths. Toward this end, we convert the program into a more convenient representation as shown in Figure 1.2.

We make the following observations about this representation:

- This representation is a directed graph with two well-defined relations *pred* and *succ*:

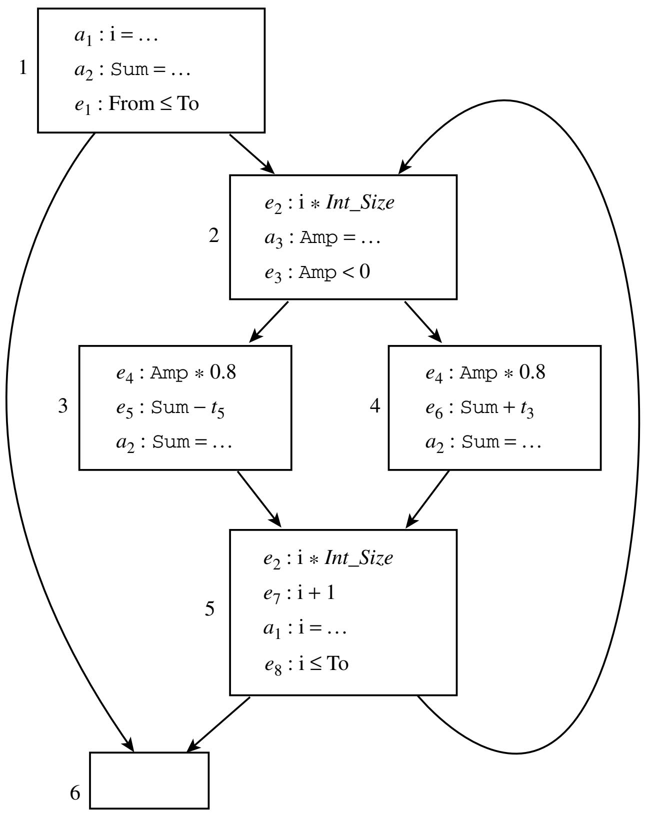
$$pred(i) = \{p \mid \exists \text{ edge } p \rightarrow i\}$$

$$succ(i) = \{s \mid \exists \text{ edge } i \rightarrow s\}$$

  Note that *succ* includes only the immediate successors and not all descendants. Similarly, *pred* includes only the immediate predecessors and not all ancestors.
- This representation makes the control transfer explicit in form of (directed) edges between nodes; the nodes themselves represent maximal group of instructions that are executed sequentially without any control transfer into or out of them. Such a group of statements is called a *basic block*.
- We have made a distinction between uses and modifications of program variables by separating expression computations and assignments; the assignments to variables are called *definitions* of variables.

1.3.2 Defining the Semantic Information to Be Captured

In this section we look at some standard data flow problems that have been widely used in theory and practice of data flow analysis; our goal is to see how they define the information to be captured by analysis. We provide a motivation for each analysis and the criterion to be satisfied by the data flow information.



Definitions : $\{a_1, a_2, a_3\}$

Expressions : $\{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8\}$

Array references have been ommitted and the “uses” and “modifications” of program variables have been highlighted by separating the computations of expressions from the assignments to program variables; the assignments to variables are called “definitions” of variables.

FIGURE 1.2 Program flow graph for the intermediate representation in Figure 1.1(b).

We talk about the information associated with a program point. Two important categories of program points are the entry and exit points of a basic block, denoted $\texttt{entry}_i$ and $\texttt{exit}_i$ for block i. The $\texttt{entry}_i$ is the point just before the execution of the first statement in the block, whereas the $\texttt{exit}_i$ is the point just after the execution of the last statement in the block.

1.3.2.1 Available Expressions Analysis

By generalizing our observation about the expression $i * Int\_Size$ in our example program, we can identify the set of expressions whose previously computed values can be reused at a given program point to eliminate redundant recomputations of the expressions.

An expression e is available at a program point p if, along every path from the start node to p, a computation of e exists that is not followed by a definition of any of its operands.

1.3.2.2 Reaching Definitions Analysis

Consider a situation where a variable x has been assigned some value at a program point (say p). If x is used at some other program point (say q), we may want to know whether the definition of x at p can influence the use of x at q. In particular, this is used for copy propagation optimization that replaces the use of a variable by the right-hand side variable of its preceding definition. This is in the hope that the preceding definition would become dead code (i.e., a definition whose value is not used), which can be eliminated. For example, if the definition in node 2 of Figure 1.2 is $\texttt{Amp} = c$, the expression e_3 in node 2 can be replaced by $c < \ldots$ and the expression e_4 in nodes 3 and 4 can be replaced by $c * 0.8$. Then variable Amp has no use and its definition can be safely eliminated.

A definition $d : v = c$ *may reach* program point p, if d appears on some path from the start node to p and is not followed by any other definition of v.

1.3.2.3 Live Variables Analysis

This analysis tries to find out if a variable has any further uses after a program point. This information can be used to identify the live range of a variable (i.e., the region of a program over which the value of the variable should be considered for keeping in a register); if the variable has no further use, its value does not need to be kept in a register. Consider the variables in node 1 in Figure 1.2. Variable `From` does not have any subsequent use whereas variable `Sum` is used in nodes 3 and 4 and `To` is used in node 5. Thus, `From` is not live at $\texttt{exit}_1$ whereas `Sum` and `To` are.

A variable v may be live at a program point p, if v is used along some path from p to a program exit, and is not preceded by its definition.

1.3.2.4 Very Busy Expressions (Anticipability) Analysis

This analysis tries to find out the set of expressions at a program point whose value is used along all paths toward program exits so as to decide whether the value of the expression should be kept in a register. In our example flow graph (Figure 1.2), expression $i * Int\_Size$ is very busy in node 2 but not in node 5.

An expression e is very busy (or anticipatable) at a program point p, if a computation of e exists that is not preceded by a definition of any operand of e along any path from p to program exits.

1.3.2.5 Constant Propagation

If it can be asserted at compile time that an expression would compute a fixed (known) value in every execution of the program, the expression computation can be replaced by the known constant value. This can then be propagated further as the value of the result of the expression to identify whether other expressions compute a constant value. We elaborate on this optimization in Section 1.3.6.2.

A variable is a constant with value c at the entry of the node if along all paths it has value c, or if it has value c along some path and is undefined along all other paths. For a definition $v := e$ in a node, the variable v is a constant with value c at the exit of the node if it can be shown that e evaluates to c and this definition is not followed by any other definition of v.

1.3.2.6 Partial Redundancy Elimination

Available expressions analysis captures redundancy of an expression along all paths (i.e., complete redundancy). However, in some situations an expression may be only partially redundant (i.e., redundant along some path but nonredundant along some other).

In Figure 1.3(a), expression $x * y$ is clearly not available at node 3. When the execution follows the path $2 \rightarrow 3$, only one computation of $x * y$ occurs on the path. However, along the path $1 \rightarrow 3$, two computations of $x * y$ occur. Thus, $x * y$ is partially redundant in node 3. This redundancy can be eliminated by inserting $x * y$ in node 2; this makes its computation in node 3 completely redundant, which can then be eliminated. This can also be seen as hoisting expression $x * y$ from node 3 into node 2 (Figure 1.3(b)).

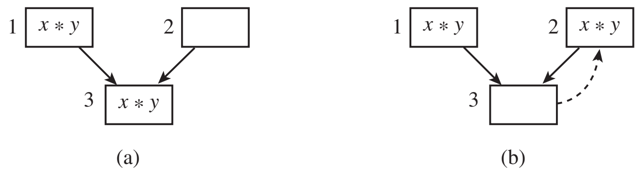


FIGURE 1.3 Partial redundancy elimination.

The precise criterion for PRE is quite subtle; intuitively, the expression must be partially redundant at a node (say i) and some predecessors of i must exist in which the expression can be inserted without changing the semantics of the program (see Appendix A for more details).

1.3.3 Modeling Data Flow Analysis: Defining Constraints

This section evolves some of the basic concepts of data flow analysis, namely, data flow information, data flow properties, data flow constraints, flow functions and data flow equations. We achieve this by using the data flow problem of available expressions analysis.

Intuitively, two reasons why an expression may not be available at a given program point could be:

- Some path exists from program entry to program point may not contain any computation of the given expression (i.e., no basic block falling on that path computes the expression). Such a path does not "generate" the value of the expression.
- Some path from program entry to the program point contains a definition of some operand of the expression (in some basic block falling on that path), which is not followed by a recomputation of the expression along the remaining path. Such a path "kills" the value of the expression.

1.3.3.1 Data Flow Information and Data Flow Properties

We can find the set of available expressions by identifying the set of expressions that are generated in a node and the set of expressions that are killed in a node. Such sets capture the data flow information used or gathered by data flow analysis. In the context of available expressions analysis, we use the following notation for these sets:

$\texttt{AvGen}_i$	Set of expressions that are computed in node i and are not followed (within the node) by a modification of any of their operands
$\texttt{AvKill}_i$	Set of expressions whose operands are modified in node i
$\texttt{AVIN}_i$	Set of expressions available at $\texttt{entry}_i$
$\texttt{AVOUT}_i$	Set of expressions available at $\texttt{exit}_i$

Each of the preceding names represents a data flow property. Thus, data flow analysis gathers data flow information by computing the values of data flow properties.

1.3.3.2 Data Flow Constraints

We define the constraint for expressions available at $\texttt{entry}_i$ as follows:

$$\forall p \in \mathit{pred}(i), \quad \texttt{AVIN}_i \subseteq \texttt{AVOUT}_p \tag{1.1}$$

This should be read as:

Only those expressions can be available at the entry point of a node, which can be available at the exit points of all predecessors, and no more.

We can make some suitable assumptions for the graph entry node that does not have any predecessor. If we restrict ourselves to the expressions involving local variables, then we can assume that $\texttt{AVIN} = \emptyset$ for the entry node. For other expressions we can initialize $\texttt{AVIN}$ of the entry node to the value available from interprocedural analysis. The constraint for expressions available at the exit of a node is defined as follows:

$$\texttt{AVOUT}_i \subseteq \texttt{AvGen}_i \cup (\texttt{AVIN}_i - \texttt{AvKill}_i) \tag{1.2}$$

This should be read as:

> Only those expressions can be available at the exit of a node, which are either generated within the node, or are available at the entry of the node and are not killed in the node — and no more.
>
> Constraint (1.2) can also be looked on as:

$$\texttt{AVOUT}_i \subseteq f_i(\texttt{AVIN}_i) \tag{1.3}$$

where f_i is a flow function associated with node i and is defined as follows:

$$f_i(X) = \texttt{AvGen}_i \cup (X - \texttt{AvKill}_i) \tag{1.4}$$

1.3.3.3 Data Flow Equations

Because a set intersection gives a subset of all the sets intersected, inequality (1.1) can be rewritten as:

$$\texttt{AVIN}_i \subseteq \bigcap_{p \in \mathit{pred}(i)} \texttt{AVOUT}_p \tag{1.5}$$

Typically, we want the largest set so it may further be converted into an equation:

$$\texttt{AVIN}_i = \bigcap_{p \in \mathit{pred}(i)} \texttt{AVOUT}_p \tag{1.6}$$

Inequalities (1.2) and (1.3) can be converted to the following equation:

$$\texttt{AVOUT}_i = \texttt{AvGen}_i \cup (\texttt{AVIN}_i - \texttt{AvKill}_i) = f_i(\texttt{AVIN}_i) \tag{1.7}$$

Use of equality in the first conversion, that is, Equation (1.6), may imply some loss of information for nondistributive analyses. Section 1.3.6.2 elaborates on this. Fortunately, available expressions analysis is distributive.

1.3.3.4 Modeling Available Expressions Analysis for Example Program

In keeping with our observation that analysis of smaller constituent units precedes the analysis of larger units, we first define the *local analysis*, which is then used for defining the *global analysis*. The former is confined to the statements within a node whereas the latter spreads across various nodes.

- *Modeling local analysis: Defining transfer functions.* For finding the expressions that are available at various nodes we need to compute `AvGen` and `AvKill` sets. `AvGen` can be directly computed. For computing `AvKill`, we list the expressions that are killed by various definitions in the following:

Definition	Variable	Expressions Killed
a_1	i	e_2, e_7, e_8
a_2	`Sum`	e_5, e_6
a_3	`Amp`	e_3, e_4

By using the preceding details, the `AvGen` and `AvKill` sets (and hence the flow functions) are defined as follows:

Node $_i$	$\texttt{AvGen}_i$	$\texttt{AvKill}_i$	$f_i(X)$
1	$\{e_1\}$	$\{e_2, e_5, e_6, e_7, e_8\}$	$\{e_1\} \cup (X - \{e_2, e_5, e_6, e_7, e_8\})$
2	$\{e_2, e_3\}$	$\{e_3, e_4\}$	$\{e_2, e_3\} \cup (X - \{e_3, e_4\})$
3	$\{e_4\}$	$\{e_5, e_6\}$	$\{e_4\} \cup (X - \{e_5, e_6\})$
4	$\{e_4\}$	$\{e_5, e_6\}$	$\{e_4\} \cup (X - \{e_5, e_6\})$
5	$\{e_8\}$	$\{e_2, e_7, e_8\}$	$\{e_8\} \cup (X - \{e_2, e_7, e_8\})$
6	$\emptyset$	$\emptyset$	X

(1.8)

- *Modeling global analysis: Defining constraints.* By embedding the flow functions from the preceding table in constraints (1.1) and (1.2), we get the following constraints for available expressions analysis of our example program:

$$\begin{aligned}\texttt{AVIN}_1 &= \emptyset \\ \texttt{AVOUT}_1 &\subseteq \{e_1\} \cup (\texttt{AVIN}_1 - \{e_2, e_5, e_6, e_7, e_8\}) \\ &\subseteq \{e_1\}\end{aligned} \tag{1.9}$$

$$\begin{aligned}\texttt{AVIN}_2 &\subseteq \texttt{AVOUT}_1 \\ &\subseteq \{e_1\} \\ \texttt{AVIN}_2 &\subseteq \texttt{AVOUT}_5\end{aligned} \tag{1.10}$$

$$\begin{aligned}\texttt{AVOUT}_2 &\subseteq \{e_2, e_3\} \cup (\texttt{AVIN}_2 - \{e_3, e_4\}) \\ \texttt{AVIN}_3 &\subseteq \texttt{AVOUT}_2 \\ \texttt{AVOUT}_3 &\subseteq \{e_4\} \cup (\texttt{AVIN}_3 - \{e_5, e_6\}) \\ \texttt{AVIN}_4 &\subseteq \texttt{AVOUT}_2\end{aligned}$$

$$\begin{aligned}\texttt{AVOUT}_4 &\subseteq \{e_4\} \cup (\texttt{AVIN}_4 - \{e_5, e_6\}) \\ \texttt{AVIN}_5 &\subseteq \texttt{AVOUT}_3 \\ \texttt{AVIN}_5 &\subseteq \texttt{AVOUT}_4 \\ \texttt{AVOUT}_5 &\subseteq \{e_8\} \cup (\texttt{AVIN}_5 - \{e_2, e_7, e_8\})\end{aligned}$$

$$\texttt{AVIN}_6 \subseteq \texttt{AVOUT}_1 \tag{1.11}$$

$$\begin{aligned}\texttt{AVIN}_6 &\subseteq \texttt{AVOUT}_5 \\ \texttt{AVOUT}_6 &\subseteq \texttt{AvGen}_6 \cup (\texttt{AVIN}_6 - \texttt{AvKill}_6) \\ &\subseteq \texttt{AVIN}_6\end{aligned} \tag{1.12}$$

1.3.4 Solutions of Data Flow Analysis

An assignment of data flow information to nodes in control flow graph represents a potential solution of data flow analysis; an assignment that satisfies the constraints represents an actual solution. It is interesting to look at a range of assignments to examine the amount and the precision of desired semantic information captured by them. This discussion motivates some important concepts using some terms intuitively; their formal definitions follow in Sections 1.5.1 and 1.5.6.

1.3.4.1 Examples of Assignments

We present several assignments for our example program without discussing how they may have been found.

1. A trivial assignment is:

$$\forall i,\ \texttt{AVIN}_i = \texttt{AVOUT}_i = \emptyset \tag{1.13}$$

   This implies that no expression is available at any point in the program and may look like an incorrect solution because several expressions are not followed by a modification of their operands. However, this assignment satisfies the constraints; and if we use it for common subexpression elimination, it cannot violate the intended semantics of the program (because no subexpression is eliminated). Hence, it is a safe assignment.
2. The other extreme is to guess the universal set **U** as the assignment at every program point:

$$\forall i,\ \texttt{AVIN}_i = \texttt{AVOUT}_i = \mathbf{U} = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8\} \tag{1.14}$$

   Clearly, this does not satisfy all constraints and performing common subexpression elimination based on this assignment can surely lead to a wrong program. This assignment is unsafe.
3. This is another assignment:

Node	AVIN	AVOUT
1	$\emptyset$	$\{e_1\}$
2	$\emptyset$	$\{e_2, e_3\}$
3	$\{e_2, e_3\}$	$\{e_2, e_3, e_4\}$
4	$\{e_2, e_3\}$	$\{e_2, e_3, e_4\}$
5	$\{e_2, e_3, e_4\}$	$\{e_3, e_4, e_8\}$
6	$\emptyset$	$\emptyset$

(1.15)

   This assignment satisfies all the constraints. However, we discover that expression e_1 is computed in node 1 and is not killed anywhere (no assignment to `From` or `To` anywhere) and yet is not available anywhere. This implies that this assignment is not the largest assignment (and hence is imprecise).
4. Yet another assignment includes e_1 at appropriate program points:

Node	AVIN	AVOUT
1	$\emptyset$	$\{e_1\}$
2	$\{e_1\}$	$\{e_1, e_2, e_3\}$
3	$\{e_1, e_2, e_3\}$	$\{e_1, e_2, e_3, e_4\}$
4	$\{e_1, e_2, e_3\}$	$\{e_1, e_2, e_3, e_4\}$
5	$\{e_1, e_2, e_3, e_4\}$	$\{e_1, e_3, e_4, e_8\}$
6	$\{e_1\}$	$\{e_1\}$

(1.16)

   Addition of any expression anywhere to this assignment can violate the constraints. Hence, this is the largest safe assignment.

1.3.4.2 Properties of Assignments

Having discussed how to interpret the assignments of data flow information, we now look at some interesting properties of these assignments.

1.3.4.2.1 ***Largeness of an Assignment***

It can be easily verified that assignment (1.16) is the largest assignment satisfying the constraints for our example program. In the general situation, the adjective large can be associated with either the size of the set used to represent the data flow information, or the amount of data flow information captured by a set. The former is an instance of set theoretical largeness whereas the latter is an instance of information theoretical largeness and the two are orthogonal. In the case of available expressions analysis, large sets indicate more information whereas in the case of reaching definitions analysis small sets represent more information. In some other situations, namely, constant propagation, the amount of information may be independent of the size of the set. This depends on the criteria to be satisfied by data flow information and can be explained as follows:

- *Available expressions analysis*. An expression is available at a program point if it is available along *all* paths leading to that program point; if it is not available along any path, it must be excluded from $\texttt{AVIN}_i$. This implies that $\texttt{AVIN}_i$ can only be a subset of $\texttt{AVOUT}$ of predecessors — constraints (1.1) and (1.5). Clearly, larger $\texttt{AVIN}_i$ (in set theoretical terms) satisfying the criteria implies that a larger number of common subexpressions can be (potentially) eliminated. To compute larger sets we replace $\subseteq$ by in constraint (1.5) by $=$ to get constraint (1.6).
- *Reaching definitions analysis*. A definition may reach a program point, if it may reach along some path leading to that program point; if it can reach along some path, it must be included in $\texttt{RDIN}_i$ even if it does not reach along some other path. This implies that $\texttt{RDIN}_i$ can only be a superset of $\texttt{RDOUT}$ of predecessors:

$$\forall p \in \mathit{pred(i)} \quad \texttt{RDIN}_i \supseteq \texttt{RDOUT}_p \tag{1.17}$$

  This should be read as:

  > The actual set of definitions satisfying constraint (1.17) may be larger than the set of definitions reaching from some predecessor p because they may not reach i from p but may reach i from some other predecessor p'.

  Another way of writing this constraint is:

$$\texttt{RDIN}_i \supseteq \bigcup_{p \in \mathit{pred(i)}} \texttt{RDOUT}_p \tag{1.18}$$

  This information is used for performing copy propagation optimization (and possibly, a subsequent dead code elimination). Note that this is possible if only one definition reaches a program point. If multiple definitions reach the program point, obviously we do not know for sure which value the variable may have and hence cannot perform optimization.
  A trivial assignment for the constraints for reaching definitions analysis is:

$$\forall i \;\; \texttt{RDIN}_i = \mathbf{U} \tag{1.19}$$

  where **U** denotes the universal set. Obviously, this assignment serves no useful purpose and we want the smallest set satisfying (1.17). In other words, the maximum meaningful information is captured by the smallest $\texttt{RDIN}_i$ set satisfying (1.17); larger $\texttt{RDIN}_i$ (in set theoretical terms) contains some "noise" (i.e., extra information that may not be meaningful). To compute smaller sets we replace $\supseteq$ by $=$:

$$\texttt{RDIN}_i = \bigcup_{p \in \mathit{pred(i)}} \texttt{RDOUT}_p \tag{1.20}$$

Whenever we talk about largeness of assignment, we refer to the information theoretical largeness of the amount of data flow information and not the set theoretical largeness of the set representing the data flow information. It must be emphasized that the largeness of data flow information is a direct consequence of the way the information is merged, which in turn is governed by the criteria to be satisfied by the data flow information. We dwell more on information theoretical largeness later in Figure 1.16 in Section 1.5.6.

1.3.4.2.2 *Safe and Maximum Safe Assignments*

If the use of any information changes the intended semantics of a program, either the information is incorrect or its use is improper (i.e., out of context or inconsistent with the intended semantics of the constraints). If an assignment satisfies the constraints, then a proper use of this assignment cannot lead to violation of the semantics of the program analysis. Hence, such an assignment is called a *safe* assignment. We explain this with the following examples:

- *Available expressions analysis*. We say that "an expression is available (i.e., included in `Avin`), *if* ... " and not "an expression is available, *if and only if* ... ." This implies that:
  - When an expression is included in an `AVIN` set, it must satisfy the conditions of availability, else it may lead to wrong optimization.
  - When an expression is not included in `AVIN` and yet satisfies the the conditions of availability, it does not contradict the statement. Hence, it is not an error but an imprecision. Though it may not lead to any optimization, the assignment is still safe.
- *Reaching definitions analysis*. For semantic soundness, we use a special value *undef* and it is assumed that the following definitions reach the entry node:

  $$\texttt{RDIN}_{entry} = \{x := \textit{undef} \mid x \text{ is a variable in the program}\} \tag{1.21}$$

  This allows us to guard against the use of undefined variables. The analysis statement says that "a definition may reach (i.e., may be included in `RDIN`), if ... ." This implies that:
  - If a definition $x := c$ is included in $\texttt{RDIN}_i$ and does not satisfy the conditions of reachability, there are two possibilities:
    - If there are multiple definitions of x in $\texttt{RDIN}_i$, then optimization cannot be performed; hence spurious inclusion of definitions in $\texttt{RDIN}_i$ is safe.
    - The situation that $x := c$ is the only definition included in $\texttt{RDIN}_i$ and in reality it does not satisfy the conditions of reachability is not possible: if no definition of x reaches i in the program, then according to our constraints, the definition $x := \textit{undef}$ must reach i (because a path must exist from the entry node to i). With spurious inclusions we may have at least two definitions in $\texttt{RDIN}_i$ unless $x := \textit{undef}$ is removed from $\texttt{RDIN}_i$; the latter is erroneous and is covered by the next case.
  - When a definition is not included in $\texttt{RDIN}_i$ but it actually satisfies the condition of reachability, we may make erroneous optimization. For instance, if two definitions $x := c_1$ and $x := c_2$ satisfy the condition of reachability at i but $\texttt{RDIN}_i$ contains only one of them, then optimization can still be performed and it can be wrong. Replace $x := c_1$ by $x := \textit{undef}$ in this argument and it is easy to see why removing $x := \textit{undef}$ erroneously may lead to wrong optimization.

A safe assignment represents a valid solution of data flow analysis. Assignments (1.13), (1.15) and (1.16) are safe assignments. However, assignment (1.13) does not capture all useful information, thereby missing some optimization opportunities. The maximum[4] safe assignment is the largest safe assignment (in information theoretical terms). It represents the largest possible solution and implies that no useful information (which could be inferred from the constraints) has been missed in the assignment. A common traditional term for the maximum safe assignment is meet over paths (MOP) solution, which basically follows from the formal definition presented in Section 1.5.6. Assignment (1.16) is the maximum safe assignment.

From assignment (1.13), it is easy to conclude that the existence of a safe assignment is trivially guaranteed.[5] Thus, the existence of the maximum safe assignment automatically follows. Any assignment that satisfies the constraints is safe. However, the formal definition of safe assignments is quite subtle — it admits not only those assignments that satisfy the constraints but also some that do not satisfy the constraints. We elaborate later on this in Figure 1.18 in Section 1.5.6.

1.3.4.2.3 *Fixed Point and Maximum Fixed Point Assignments*

Though assignment (1.13) is safe, if we try to recompute AVIN and AVOUT — using the values in assignment (1.13) — it is possible to add a few expressions to them. This happens because the use of inequalities allows multiple values to satisfy a given constraint. Thus if a solution uses a smaller value satisfying the constraint, it is possible to update the solution by choosing a larger value satisfying the constraint. With equalities in constraints, the AVIN sets of assignment (1.13) are not consistent with the AVOUT sets of the predecessors and the AVOUT sets are not consistent with the AVIN sets of the same node.

Mathematically, a fixed point of a function $h : A \mapsto A$ is a value $x \in A$ such that $h(x) = x$. We use this concept to define a *fixed point assignment* that captures the fact that no value can be updated any further by recomputations. This enables characterization of termination of analysis.

Assignment (1.13) is a safe assignment but neither the maximum safe assignment nor a fixed point assignment. Assignment (1.15), on the other hand, is not only a safe assignment but also a fixed point assignment, although it is not the maximum safe assignment. It might appear that it is possible to update $\texttt{AVIN}_2$ in this assignment by adding e_1 to it because it is contained in $\texttt{AVOUT}_1$. However, e_1 is not contained in $\texttt{AVOUT}_5$ and because node 5 is a predecessor of node 2, we cannot conclude that e_1 is available at $\texttt{entry}_2$. Ironically, we observe that if we assume that e_1 is available at $\texttt{exit}_5$, then it becomes available at node 2. Because neither From nor To is modified anywhere, e_1 becomes available at node 5, thereby confirming its initial inclusion in $\texttt{AVOUT}_5$. An "if" qualifies the preceding statement and unless we make that assumption, we cannot update this assignment any further and it is indeed consistent, though not maximally consistent. Assignment (1.16) is the maximum safe assignment and a fixed point assignment. It is maximally consistent because it assumed e_1 to be available in $\texttt{AVOUT}_5$ (we revisit this in Section 1.3.7).

Unlike safe assignment, the existence of a fixed point assignment is not guaranteed; we discuss this in Section 1.3.6.

The maximum fixed point assignment is the largest fixed point assignment (in information theoretical terms).[6] Assignment (1.16) is the maximum safe and the maximum fixed point assignment.

[4]Is it "maximal" or "maximum"? If a unique largest safe assignment exists, then it is the maximum safe assignment; if multiple largest safe assignments exists, then we have maximal safe assignments. Section 1.5.1 assures us that in our case, we have a unique largest safe assignment.

[5]If you do not want to make a mistake, one way of guaranteeing it is to not do any work.

[6]The traditional data flow analysis literature usually does not distinguish between the information theoretical largeness and the set theoretical largeness. For available expressions analysis, the desired solution is called the *maximum fixed point* solution whereas for reaching definitions analysis, it is called the *minimum fixed point* solution. Thanks to our general characterization, such a dual term is not required for our discussion.

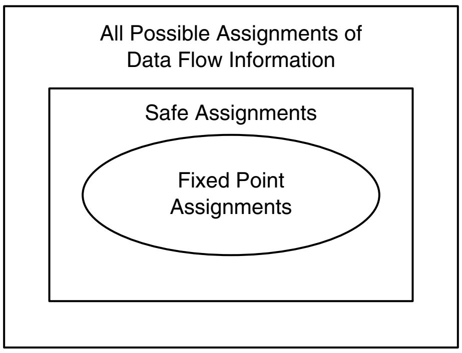


FIGURE 1.4 Categories of assignments of a data flow problem.

1.3.4.2.4 *Relationship between Maximum Safe, Safe and Fixed Point Assignments*

Safety guarantees the correctness of information derived from an analysis, maximum safety guarantees exhaustiveness of correct information and fixed points enable characterization of the termination of analysis. Figure 1.4 summarizes the relationship between different kinds of assignments; several interesting conclusions can be drawn from it. We make this figure more precise and reproduce it later as Figure 1.17 in Section 1.5.6.

In the case of nondistributive problems, the maximum fixed point assignment may not be the maximum safe assignment, and the maximum safe assignment does not need to be a fixed point assignment. These are indicated by the containment of fixed point assignments in safe assignments in Figure 1.4. We explain this in Section 1.3.6.2.

1.3.5 Soundness of Data Flow Analysis

Soundness of data flow analysis is a combination of the soundness of modeling and the soundness of the use of the information derived by analysis. It is possible that the modeling is sound but the use of safe assignment is improper. Conversely, it is also possible that the modeling is unsound. It is too easy to rely on intuition and make subtle mistakes (or conversely, to take rigor for granted and accept unintuitive assignments as valid). An example of an unsound modeling is leaving out definitions (1.21) from reaching definitions analysis. In such a case, if no definition of x actually reaches node i and a definition $x := c$ is included in $\texttt{RDIN}_i$, the solution would have this definition as the only definition of x in $\texttt{RDIN}_i$. Hence, copy propagation for x may be performed and definitions of x may be eliminated as dead code leading to incorrect program.

An example of an improper use of a safe assignment is presented in Section 1.5.6.

1.3.6 Some Theoretical Properties of Data Flow Analysis

Having looked at the possible assignments of data flow analysis, it is important to understand some properties of data flow analysis that influence the possibilities of finding these assignments. These properties form the basis of the actual methods of performing analysis; hence, this discussion precedes the discussion of solution methods of data flow analysis.

1.3.6.1 Properties Influencing the Convergence of Analysis

Each constraint corresponds to the shortest segment of a data flow path. These segments can be combined in many ways and many times depending on the interdependence of constraints (this, in turn, depends on the structure of the flow graph). This raises two important questions:

- How many paths would have to be traversed for performing analysis?
- How long will these paths be?

We answer these questions by examining constraints (1.11) and (1.12) for $\texttt{AVIN}_6$. By successively substituting the right-hand sides of relevant constraints, we can rewrite (1.12) as two different constraints (one along the path $2 \rightarrow 3 \rightarrow 5 \rightarrow 6$ and the other along path $2 \rightarrow 4 \rightarrow 5 \rightarrow 6$):

$$\texttt{AVIN}_6 \subseteq f_5(f_3(f_2(\texttt{AVIN}_2))) \tag{1.22}$$

$$\texttt{AVIN}_6 \subseteq f_5(f_4(f_2(\texttt{AVIN}_2))) \tag{1.23}$$

$\texttt{AVIN}_2$ depends on $\texttt{AVOUT}_1$ and $\texttt{AVOUT}_5$. Further substitutions in (1.22) and (1.23) yield multiple constraints, two constraints resulting from substituting for $\texttt{AVOUT}_1$ and the other two resulting from substituting for $\texttt{AVOUT}_5$, namely:

$$\begin{aligned} \texttt{AVIN}_6 &\subseteq f_5(f_3(f_2(f_1(\texttt{AVIN}_1)))) \\ \texttt{AVIN}_6 &\subseteq f_5(f_3(f_2(f_5(\texttt{AVIN}_5)))) \end{aligned} \tag{1.24}$$

$$\begin{aligned} \texttt{AVIN}_6 &\subseteq f_5(f_4(f_2(f_1(\texttt{AVIN}_1)))) \\ \texttt{AVIN}_6 &\subseteq f_5(f_4(f_2(f_5(\texttt{AVIN}_5)))) \end{aligned} \tag{1.25}$$

Note the recursive application of f_5 in inequalities (1.24) and (1.25). If we substitute for $\texttt{AVIN}_5$, we get recursive applications of f_4 and f_2 (or f_3 and f_2) also. Each recursive function application implies that the path (traced by the flow function composition) is retraversed.

Because the traversal of a cyclic path is not bounded in length (i.e., generally the number of times a loop can be executed is not known at compile time), we can have potentially an infinite number of paths, many of which may be potentially infinite in length. Let $\mathscr{P}_{IN(i)}$ represent the set of all paths from the initial node to $\texttt{entry}_i$. Then the constraint for $\texttt{AVIN}_6$ can be expressed as:

$$\forall \rho \in \mathscr{P}_{IN(6)}, \quad \texttt{AVIN}_6 \subseteq f_\rho(\texttt{AVIN}_1) \tag{1.26}$$

where f_ρ denotes the function compositions along a path in $\mathscr{P}_{IN(6)}$. Though theoretically we have to deal with infinitely many path traversals of infinite length, practically all flow functions are monotonic (i.e., the values either do not decrease for increasing inputs or do not increase for decreasing inputs). Monotonicity guarantees convergence of recursive calls if:

- Either the number of possible values of a data flow property is finite
- Or the confluence operation (in our case $\cap$) has some desirable properties

This eliminates the infiniteness of both the number as well as the length of path traversals by bounding the number of times a cycle would have to be traversed in a path. In practice, it is desirable that this bound should be a (small) constant instead of a function of some measure of the program to be analyzed. For available expressions analysis, this number is 1.[7] We discuss these concepts with due rigor in Section 1.5.1.

1.3.6.2 Properties Influencing the Quality of Solutions

Yet another important question is: Will data flow information be computed afresh along each path even if path segments are shared? Clearly, this is undesirable and can be avoided by:

[7]Formally, the bound is one more than the number of times a cycle may have to be traversed.

- Summarizing this information at a node by grouping the constraints
- Remembering this information and reusing it instead of recomputing it afresh each time it is required

Summarization of data flow information requires grouping of related constraints and implies merging of shared path segments. Then, retraversals of shared path segments can be avoided by remembering and reusing the summarized information. This implies that for constraint (1.26), there is no need to actually compute the function composition f_ρ explicitly. Instead, it is sufficient to apply f_5 to $\texttt{AVIN}_5$ and f_1 to compute $\texttt{AVOUT}_5$ and $\texttt{AVOUT}_1$ and then compute $\texttt{AVIN}_6$ by:

$$\texttt{AVIN}_6 \subseteq \texttt{AVOUT}_1 \cap \texttt{AVOUT}_5$$

This is under the assumption that $\texttt{AVIN}_1$ and $\texttt{AVIN}_5$ have also been computed in the same way. For the maximality of values, $\subseteq$ can be replaced by $=$.

Summarization of information is not without any price. It may lead to some loss of information. We explain this with the help of the data flow analysis for constant propagation.

1.3.6.2.1 *An Aside on Constant Propagation*

Figure 1.5 contains the program flow graph for our example. The question that is of interest is, Is z constant in node 5? A manual inspection of the flow graph reveals that z is indeed constant whose value is 24. We explain the effect of summarization on this information by defining the flow functions and the confluence operation for constant propagation.

- *Data flow information*. Let the data flow information (denoted by $\texttt{CIN}_i$ and $\texttt{COUT}_i$) be represented by a set of tuples $\langle v, t\rangle$ where v is a variable and t is a tag that can have the following values:

  - *ud*, if v is undefined
  - *nc*, if v is not a constant
  - Number n, if v is a constant with value n

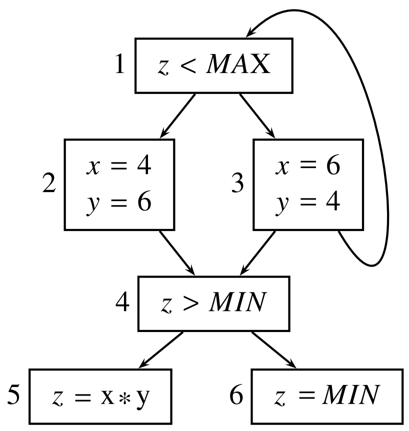


FIGURE 1.5 Loss of information in constant propagation.

- *Flow functions*. We need to define flow functions for each arithmetic expression. For brevity, we define the flow function for multiplication (denoted *mult*) only. Let a_1 and a_2 be the two arguments and r, the result (i.e., $r = a_1 * a_2$). Then:

mult	$\langle a_1, ud\rangle$	$\langle a_1, nc\rangle$	$\langle a_1, c_1\rangle$
$\langle a_2, ud\rangle$	$\langle r, ud\rangle$	$\langle r, nc\rangle$	$\langle r, c_1\rangle$
$\langle a_2, nc\rangle$	$\langle r, nc\rangle$	$\langle r, nc\rangle$	$\langle r, nc\rangle$
$\langle a_2, c_2\rangle$	$\langle r, c_2\rangle$	$\langle r, nc\rangle$	$\langle r, (c_1 * c_2)\rangle$

(1.27)

The preceding definition says that the result is a constant if either the two arguments are constants or one of them is undefined and the other is constant. If both arguments are undefined, then the result is undefined. In all other cases, the result is not constant.

- *Confluence operation.* The confluence operation for a single element in the set is defined as follows and can be lifted to arbitrary sets elementwise.

$\sqcap$	$\langle v, ud\rangle$	$\langle v, nc\rangle$	$\langle v, c_1\rangle$
$\langle v, ud\rangle$	$\langle v, ud\rangle$	$\langle v, nc\rangle$	$\langle v, c_1\rangle$
$\langle v, nc\rangle$	$\langle v, nc\rangle$	$\langle v, nc\rangle$	$\langle v, nc\rangle$
$\langle v, c_2\rangle$	$\langle v, c_2\rangle$	$\langle v, nc\rangle$	$\langle v, c_1\rangle$ if $c_1 = c_2$ $\langle v, nc\rangle$ otherwise

(1.28)

- *Constant propagation without summarization.* In this case we try to find out if x and y are constant in node 5 along different paths reaching node 5 without merging the information about x and y at intermediate points in the shared path segments. The two paths reaching node 5 are $1 \rightarrow 2 \rightarrow 4 \rightarrow 5$ and $1 \rightarrow 3 \rightarrow 4 \rightarrow 5$.

  - Along the path $1 \rightarrow 2 \rightarrow 4 \rightarrow 5$, the value of x is 4 and the value of y is 6. Thus, both x and y are constant. The value of z is 24.
  - Along the path $1 \rightarrow 3 \rightarrow 4 \rightarrow 5$, the value of x is 6 and the value of y is 4. Thus, both x and y are constant. The value of z is 24.

  Because the value of z is same (i.e., 24) along both the paths, z indeed is constant in node 5. Thus the maximum safe assignment is:

Node	$\texttt{CIN}_i$	$\texttt{COUT}_i$
1	$\{\langle x, ud\rangle, \langle y, ud\rangle, \langle z, ud\rangle\}$	$\{\langle x, ud\rangle, \langle y, ud\rangle, \langle z, ud\rangle\}$
2	$\{\langle x, ud\rangle, \langle y, ud\rangle, \langle z, ud\rangle\}$	$\{\langle x, 4\rangle, \langle y, 6\rangle, \langle z, ud\rangle\}$
3	$\{\langle x, ud\rangle, \langle y, ud\rangle, \langle z, ud\rangle\}$	$\{\langle x, 6\rangle, \langle y, 4\rangle, \langle z, ud\rangle\}$
4	$\{\langle x, nc\rangle, \langle y, nc\rangle, \langle z, ud\rangle\}$	$\{\langle x, nc\rangle, \langle y, nc\rangle, \langle z, ud\rangle\}$
5	$\{\langle x, nc\rangle, \langle y, nc\rangle, \langle z, ud\rangle\}$	$\{\langle x, nc\rangle, \langle y, nc\rangle, \langle z, 24\rangle\}$
6	$\{\langle x, nc\rangle, \langle y, nc\rangle, \langle z, ud\rangle\}$	$\{\langle x, nc\rangle, \langle y, nc\rangle, \langle z, MIN\rangle\}$

(1.29)

- *Constant propagation with summarization.* In this case we try to find whether x and y are constant in node 5 by merging the information about x and y in the shared path segments along different paths reaching node 5. The information can be merged at node 4.

  - The value of x is 4 along the path segment $1 \to 2 \to 4$ and 6 along the path segment $1 \to 3 \to 4$. Thus, x is not constant in node 4 (and hence in node 5 also).
  - The value of y is 6 along the path segment $1 \to 2 \to 4$ and 4 along the path segment $1 \to 3 \to 4$. Thus, y is not constant in node 4 (and hence in node 5 also).

  Because x and y are not constant in node 5, z is not constant. Thus, our maximum fixed point assignment is:

Node	$\texttt{CIN}_i$	$\texttt{COUT}_i$
1	$\{\langle x, ud\rangle, \langle y, ud\rangle, \langle z, ud\rangle\}$	$\{\langle x, ud\rangle, \langle y, ud\rangle, \langle z, ud\rangle\}$
2	$\{\langle x, ud\rangle, \langle y, ud\rangle, \langle z, ud\rangle\}$	$\{\langle x, 4\rangle, \langle y, 6\rangle, \langle z, ud\rangle\}$
3	$\{\langle x, ud\rangle, \langle y, ud\rangle, \langle z, ud\rangle\}$	$\{\langle x, 6\rangle, \langle y, 4\rangle, \langle z, ud\rangle\}$
4	$\{\langle x, nc\rangle, \langle y, nc\rangle, \langle z, ud\rangle\}$	$\{\langle x, nc\rangle, \langle y, nc\rangle, \langle z, ud\rangle\}$
5	$\{\langle x, nc\rangle, \langle y, nc\rangle, \langle z, ud\rangle\}$	$\{\langle x, nc\rangle, \langle y, nc\rangle, \langle z, nc\rangle\}$
6	$\{\langle x, nc\rangle, \langle y, nc\rangle, \langle z, ud\rangle\}$	$\{\langle x, nc\rangle, \langle y, nc\rangle, \langle z, MIN\rangle\}$

(1.30)

  Clearly, summarization misses the fact that z is a constant in node 5.

Use of equations gives us a fixed point assignment, but in this case the maximum safe assignment is not a fixed point; it is larger than the maximum fixed point.[8] Fortunately, such loss of useful information takes place for only for a few practical data flow problems. For most (not all) practical data flow problems, summarization gives the same assignment as would be computed by traversing all paths independently (i.e., the maximum fixed point assignment is also the maximum safe assignment); such data flow problems are called *distributive*. Accordingly, available expressions analysis is distributive whereas the analysis for constant propagation is nondistributive; intuitively, this happens because constantness of z in node 5 depends on constantness of variables other than z (in this case x and y). We look at this more precisely and rigorously in Section 1.5.2.

1.3.7 Performing Data Flow Analysis: Solving Constraints

Because constraints depend on each other, one would like to solve them by systematic substitution of variables by the corresponding right-hand sides.[9] For our example, we substitute the right-hand side of $\texttt{AVIN}_1$ in the right-hand side of $\texttt{AVOUT}_1$. The resulting right-hand side is substituted in the right-hand side of $\texttt{AVIN}_2$. However, the second constraint for $\texttt{AVIN}_2$, that is, constraint (1.10) requires the value of $\texttt{AVOUT}_5$. Thus:

[8]Maximum safe assignment can never be smaller than a fixed point. Why?

[9]When we use substitution in the case of inequalities, we should consistently choose either the largest or the smallest value satisfying the constraint.

$$
\begin{aligned}
\mathsf{AVIN}_2 &\subseteq \mathsf{AVOUT}_5 \\
&\subseteq \underbrace{(\{e_8\} \cup (\mathsf{AVIN}_5 - \{e_2, e_7, e_8\}))}_{\text{RHS for } \mathsf{AVOUT}_5} \\
&\subseteq \underbrace{(\{e_8\} \cup (\mathsf{AVOUT}_3 - \{e_2, e_7, e_8\}))}_{\text{RHS for } \mathsf{AVOUT}_5} \\
&\subseteq \underbrace{(\{e_8\} \cup (\underbrace{(\{e_4\} \cup (\mathsf{AVIN}_3 - \{e_5, e_6\}))}_{\text{RHS for } \mathsf{AVOUT}_3} - \{e_2, e_7, e_8\}))}_{\text{RHS for } \mathsf{AVOUT}_5} \\
&\subseteq \underbrace{(\{e_8\} \cup (\underbrace{(\{e_4\} \cup (\mathsf{AVOUT}_2 - \{e_5, e_6\}))}_{\text{RHS for } \mathsf{AVOUT}_3} - \{e_2, e_7, e_8\}))}_{\text{RHS for } \mathsf{AVOUT}_5} \\
&\subseteq \underbrace{(\{e_8\} \cup (\underbrace{(\{e_4\} \cup (\underbrace{(\{e_2, e_3\} \cup (\mathsf{AVIN}_2 - \{e_3, e_4\}))}_{\text{RHS for } \mathsf{AVOUT}_2} - \{e_5, e_6\}))}_{\text{RHS for } \mathsf{AVOUT}_3} - \{e_2, e_7, e_8\}))}_{\text{RHS for } \mathsf{AVOUT}_5}
\end{aligned}
\tag{1.31}
$$

Thus, $\mathtt{AVIN}_2$ depends on itself. This is what was captured by the observation of recursive applications of functions in the previous section. If there had not been a recursive dependence like this, a systematic substitution of the terms would give us a solution for the constraint. Clearly, solving the system of constraints requires the elimination of recursive dependences. This can be achieved in one of two ways:

- Assume a suitable initial value of recursively dependent data flow information, substitute it in right-hand sides to circumvent the recursive dependence and recompute the value.
- Delay the computation of recursively dependent data flow information by substituting a (group of) variables by a new variable and eliminate the recursive dependence.

We discuss these two approaches in the rest of this section.

1.3.7.1 Solving Constraints by Successive Refinement

In this case we assume an approximate value for each variable and then recompute it by substituting it in the right-hand side of the constraints defining the variable. If the initial choice of the value is appropriate, the recomputations can make the values more precise. This process may have to be repeated until the values stabilize. In the current context two obvious choices for the initial guess are:

- *Empty set* $\emptyset$. When we substitute $\emptyset$ for $\mathtt{AVIN}_2$ in the right-hand side of constraint (1.31), it reduces to:

$$\mathtt{AVIN}_2 \subseteq \{e_2, e_3, e_4, e_8\} \tag{1.32}$$

  The initial guess ($\emptyset$) satisfies this constraint. If we substitute the right-hand side of (1.32) back in the right-hand side of (1.31), we get the same solution, implying that it does not need to be recomputed any more. However, $\mathtt{AVIN}_2$ must also satisfy constraint (1.9). Hence, the only solution possible for $\mathtt{AVIN}_2$ in this case is $\emptyset$. This is how solution (1.15) was computed.

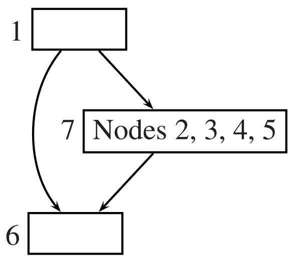


FIGURE 1.6 Grouping of interdependent nodes.

- *Universal set* ($\{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8\}$). When this is substituted in (1.31), it becomes:

$$\texttt{AVIN}_2 \subseteq \{e_1, e_2, e_3, e_4, e_8\} \tag{1.33}$$

  Because $\texttt{AVIN}_2$ must also satisfy constraint (1.9), in this case two values are possible: $\emptyset$, and $\{e_1\}$. Because we want the largest solution, we should select the latter value. This is how assignment (1.16) was computed.

Other choices are also possible — in general any initial value larger than the largest solution can serve the purpose. Also note that when successive recomputations are used, the constraints can be solved in any order; no particular order is necessary, although one order may be more efficient (i.e., may require less computation) than the other.

1.3.7.2 Solving Constraints by Delaying Some Computations

We can delay some computations by identifying all maximal groups of mutually dependent sets and abstracting out each set by a single variable. It can also be seen as abstracting out a group of nodes in the graph by a single node. In our case a single group exists that consists of the following sets: $\texttt{AVIN}_2$, $\texttt{AVOUT}_2$, $\texttt{AVIN}_3$, $\texttt{AVOUT}_3$, $\texttt{AVIN}_4$, $\texttt{AVOUT}_4$, $\texttt{AVIN}_5$ and $\texttt{AVOUT}_5$. In terms of nodes, we can replace the group 2, 3, 4 and 5 by a single newly constructed node (say 7) as shown in Figure 1.6. This replacement is only conceptual and is used for computing the flow functions (for the interdependent nodes) and values at other program points.

Now if we define constraints for $\texttt{AVIN}_7$ and $\texttt{AVOUT}_7$, there can be no recursive dependence and a systematic substitution can give us the solution of the new system of constraints. Then, we are able to compute the values of the AVIN/AVOUT for nodes 2, 3, 4 and 5 from the value of $\texttt{AVIN}_7$. This approach is the same as the standard elimination methods used for solving linear systems of simultaneous equations.[10] We explain this method in greater detail in Section 1.6.2.

1.3.8 Generic Abstractions in Data Flow Analysis

After examining the basic concepts of data flow analysis, it is time now to discuss some generic abstractions in data flow analysis to cover the data flow problems mentioned in Section 1.3.2.

[10]Our equations involve sets, and, though the solution methods can be borrowed from the mathematics of linear system of simultaneous equations, its theory is not applicable in this context. We cannot use it to reason about the existence of, or the uniqueness of, solutions — or about the convergence of the solution process. For us, the relevant mathematics is set algebra and our equations can be viewed as Boolean equations, much in the genre of satisfiability problems that are NP complete. However, we have discussed some properties in Section 1.3.4.2 that deal with the issues of existence of, and convergence on, solutions. These properties are presented with due rigor in Section 1.5.1.

1.3.8.1 Data Flow Values

By data flow values we mean the values associated with $\texttt{entry}_i$ or $\texttt{exit}_i$ variables. They represent the aggregate data flow information about all the data items that are analyzed. For available expressions analysis, we used sets of expressions as data flow values. We can also view the value associated with a single expression as a boolean value that is *true* if the expression is available and *false* if the expression is not available. In general, the data flow values can be seen as sets of (or aggregates of) scalar values or tuples. As we have seen in Section 1.3.6.2, we cannot use simple Boolean values for representing the constantness information in constant propagation; instead we have to use a tuple. As long as the data flow values satisfy some well-defined rigorous conditions (formally defined in Section 1.5), no restrictions exist on the types of values.

1.3.8.2 Confluence Operation

We used $\cap$ in available expression analysis to combine the data flow information because we wanted to derive information that was the same along all paths. As noted in Section 1.3.2, for reaching definitions analysis we relax the condition that the same information be carried along all paths because the relevant information may be carried by only some paths. We have used $\cup$ operation in Equation (1.18) for capturing such information. We say that a data flow problem uses:

- *Any Path* flow confluence if the relevant information can be carried by any path
- *All Paths* flow confluence if the (same) relevant information must be carried by all paths

This still is not sufficiently general because $\cap$ and $\cup$ are not the only operations that are used; a different operation may have to be devised depending on the data flow values. Whenever the values can be booleans, we can use $\cap$ and $\cup$. However, if the values are not booleans, we need to define appropriate confluence operation on them. We have done this for constant propagation in Section 1.3.6.2. When we use a special confluence operation such as this, classifying it as any path or all paths is hard; it could only be looked on as a combination of the two. For generality, we use $\sqcap$ to denote all the three possibilities of confluences.

1.3.8.3 Direction of Data Flow

Constraint (1.1) for available expressions analysis suggests that the data flow information at a node is influenced by the data flow information at its predecessors. In other words, the flow of information between two nodes is from predecessors to successors (or from ancestors to descendants, in general); within a node, it is from `entry` to `exit`. Because such a flow is along the direction of control flow, it is called a *forward flow*. A *backward flow*, on the other hand, is against the direction of control flow. The flow of information between two nodes is from successors to predecessors (or from descendants to ancestors, in general); within a node, it is from `exit` to `entry`. From the description in Section 1.3.2, it is clear that live variables analysis and very busy expressions analysis use backward flows.

It is possible to view data flow analysis at an abstract level independent of the direction of flow; the forward problems can be considered duals of backward data flow problems (and vice versa) with regard to the direction and no separate theoretical treatment is required for them.

1.3.8.4 Data Flow Equations

For forward data flow problems, the generic data flow equations are:

$$\texttt{IN}_i = \begin{cases} \mathit{Outside\_Info} & \text{if } i \text{ is a graph entry node} \\ \displaystyle\bigsqcap_{p \in pred(i)} \texttt{OUT}_p & \text{otherwise} \end{cases} \tag{1.34}$$

$$\texttt{OUT}_i = f_i(\texttt{IN}_i) \tag{1.35}$$

They can be adapted to any forward data flow problem by merely substituting appropriate confluence operation $\sqcap$ and flow functions f.

Note the special treatment given to the graph entry nodes. Clearly, we cannot assume anything *a priori* about the information reaching the graph from the outside world. Traditionally, this has almost always received a misleading treatment of the form: assume ∅ for any path problems and assume the universal set for all path problems. This is incorrect for the graph entry and exit nodes (and hence exceptions to the general rule are introduced). The actual information at these nodes depends on the language to be analyzed, the exact semantics of analysis and the categories of data items covered by the analysis. For instance, if we want to include the expressions involving global variables, it may be fair to expect the interprocedural data flow analysis to infer (if possible) whether the values of such expressions are available at the graph entry. We leave it for the designer of data flow analysis to choose appropriate value for *Outside_Info* because it is not connected with any path or all paths confluence in any way. In the case of reaching definitions analysis, *Outside_Info* is captured by definition (1.21).

For backward data flow problems, the generic data flow equations are:

$$\mathtt{IN}_i = f_i(\mathtt{OUT}_i) \tag{1.36}$$

$$\mathtt{OUT}_i = \begin{cases} \mathit{Outside\_Info} & \text{if } i \text{ is a graph exit node} \\ \bigsqcap_{s \in succ(i)} \mathtt{IN}_s & \text{otherwise} \end{cases} \tag{1.37}$$

1.3.8.5 Taxonomy of Data Flow Analysis

After developing all basic concepts in data flow analysis, it is now possible to compare and contrast some data flow problems for the way they employ the generic abstractions. It is easy to see that the four classical data flow problems represent the four possible combinations of the direction of flow and the confluence of flow, as follows:

	Direction of Flow	
Confluence	Forward	Backward
All paths	Available expressions analysis	Anticipability analysis
Any path	Reaching definitions analysis	Live variables analysis

This taxonomy is actually quite simplistic and fails to cover many data flow problems (e.g., partial redundancy elimination). We present some advanced issues in data flow analysis in Section 1.4, and some examples of advanced data flow and more elaborate taxonomies in Appendix A.

1.4 Advances in Data Flow Analysis

The concepts that we have discussed so far belong to the classical data flow analysis in that they have been in existence for a long time, and have been used widely in theory and practice of data flow analysis. One thread that is common to these concepts is that the flow components in any data flow problem are *homogeneous* in nature (i.e., of the same kind); either all of them are forward or all of them are backward — no combinations are allowed. Similarly, only one kind of confluence is allowed. Data flow analysis has progressed beyond these limiting concepts in that heterogeneous components have also been used fruitfully. In this section, we look at some advanced concepts; some of them have been theorized in a satisfactory manner whereas some others are still waiting for technical breakthroughs required to devise sound theoretical foundations for them.

1.4.1 Background

The discussion in this section is mostly driven by data flow analysis requirements of the partial redundancy elimination (PRE) and type inferencing. There have been many formulations of PRE, but the one that stretches the limits of our theory (and hence enhances it) is the original formulation by Morel and Renvoise. Hence, we use this formulation instead of the other formulations; to highlight this distinction we call it Morel–Renvoise partial redundancy elimination (MR-PRE). We would like to emphasize that:

> Our use of MR-PRE is motivated by the need of understanding the nature of flows that data flow analysis may have to support. *This does not require understanding the semantics of the data flow dependencies in MR-PRE; it merely requires understanding the structure of flows.* Toward this end, we merely present the original formulation of MR-PRE in Appendix A. From these equations, it is easy to decipher the flows, though it is very hard to divine why they have been defined in that way or whether they can be defined in an alternative way; the latter is irrelevant for the discussions in this chapter.

Similar remarks apply to type inferencing; we restrict ourselves to a specific formulation of type inferencing (which we call KDM type inferencing) that basically formalizes Tennenbaum's type inferencing data flow analysis.

1.4.2 Enriching the Classical Abstractions in Data Flow Analysis

The generic abstractions that are used in new ways or are enriched and made more precise by the preceding data flow analyses are confluences of flows, directions of flows and flow functions.

1.4.2.1 Multiple Confluences of Flows

The classical data flow problems have only one confluence that is manifested in the form of a single $\sqcap$ in data flow Equations (1.34) and (1.37). However, it is possible to have multiple confluences in a data flow problem (i.e., multiple $\sqcap$ in data flow equations). MR-PRE has two confluences; the data flow information is merged at both $\texttt{entry}_i$ and $\texttt{exit}_i$ (Equations (1.61) and (1.62) in Appendix A). In the general situation, the multiple confluences do not need to be the same; one of them may be $\cap$ whereas the other may be $\cup$. The classical theory of data flow analysis can deal with only one confluence; the more advanced generalized theory of data flow analysis can deal with multiple but homogeneous confluences. Data flow analysis has still not advanced sufficiently to provide a theoretical foundation for data flow problems that use heterogeneous confluences. However, such data flow problems do exist in practice; we describe modified Morel–Renvoise algorithm (MMRA) and edge placement algorithm (EPA) data flow analyses in Appendix A.

1.4.2.2 Multiple Directions of Flows

It is possible to have both forward and backward data flows in the same problem (PRE uses forward as well as backward data flows). The classical theory can deal with only *unidirectional* data flow problems that have either forward or backward flows, but not both. The more advanced generalized theory provides sound theoretical foundations for *bidirectional* data flow problems — these foundations are uniformly applicable to unidirectional data flow problems also.

The data flow dependencies in the classical data flow problems are:

- *Forward data flows.* $\texttt{IN}_k$ is computed from $\texttt{OUT}_i$, $i \in \mathit{pred}(k)$ and $\texttt{OUT}_k$ is computed from $\texttt{IN}_k$, and so on — Figure 1.8(a).
- *Backward data flows.* $\texttt{OUT}_k$ is computed from $\texttt{IN}_m$, $m \in \mathit{succ(k)}$ and $\texttt{IN}_k$ is computed from $\texttt{OUT}_k$, and so on — Figure 1.8(b).

Apart from these dependencies it is possible that $\texttt{IN}_i$ is computed from $\texttt{OUT}_i$ and $\texttt{OUT}_i$ is computed from $\texttt{IN}_i$, and $\texttt{IN}_i$ is computed from $\texttt{OUT}_p$, $p \in \mathit{pred(i)}$ and $\texttt{OUT}_p$ is computed from $\texttt{IN}_i$ in the same data flow problem. Such dependencies give rise to the flows in Figure 1.8(c) and 1.8(d).

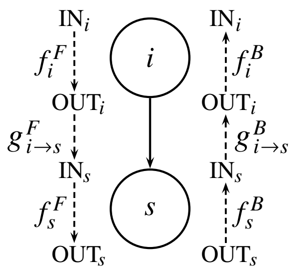


FIGURE 1.7 Node and edge flow functions.

1.4.2.3 Multiple Kinds of Flow Functions

In the case of available expressions analysis, it is possible to use simple algebraic substitution to combine Equations (1.6) and (1.7) into a single equation:

$$\texttt{AVOUT}_i = \bigcap_{p\in\ pred(i)} f_i(\texttt{AVOUT}_p) \tag{1.38}$$

Alternatively:

$$\texttt{AVIN}_i = \bigcap_{p\in\ pred(i)} f_{p\to i}(\texttt{AVIN}_p) \tag{1.39}$$

This implies that the flow function f_i can be associated with node i or with the edge $p \to i$ without any loss of generality. Such flow functions can be used for unidirectional data flows only. They cannot be used for more general flows (Figures 1.8(c) and 1.8(d)). The generalized theory of data flow analysis enables the characterization of these general flows by carefully distinguishing between the node flow functions and the edge flow functions. Node flow functions map information between entry and exit points of a node whereas edge flow functions map information from a node to its predecessors and successors.

Figure 1.7 illustrates the various kinds of flow functions, for node i and $s \in$ *succ(i)*:

- *Forward node flow functions* f_i^F map $\texttt{IN}_i$ to $\texttt{OUT}_i$ ($\texttt{OUT}_i$ is computed from $\texttt{IN}_i$ using f_i^F).
- *Backward node flow functions* f_i^B map $\texttt{OUT}_i$ to $\texttt{IN}_i$ ($\texttt{IN}_i$ is computed from $\texttt{OUT}_i$ using f_i^B).
- *Forward edge flow functions* $g_{i\to s}^F$ map $\texttt{OUT}_i$ to $\texttt{IN}_s$ ($\texttt{IN}_s$ is computed from $\texttt{OUT}_i$ using $g_{i\to s}^B$).
- *Backward edge flow functions* $g_{i\to s}^B$ map $\texttt{IN}_s$ to $\texttt{OUT}_i$ ($\texttt{OUT}_i$ is computed from $\texttt{IN}_s$ using $g_{i\to s}^B$).

By using these functions, all the flows can be characterized as shown in Figure 1.8 where the function compositions are denoted by $f \circ g$ where $[f \circ g](x) \equiv f(g(x))$. Note that for traditional unidirectional flows the edge flow functions are identity functions.

We list these abstractions for some of the data flow problems mentioned in Section 1.3.2.

Data Flow Problem	Direction of Node Flows	Direction of Edge Flows	Confluence of Edge Flows
Available expressions analysis	Forward	Forward	All paths
Reaching definitions analysis	Forward	Forward	Any path
Live variables analysis	Backward	Backward	Any path
Very busy expressions analysis	Backward	Backward	All paths
Partial redundancy elimination	Forward	Forward and backward	All paths

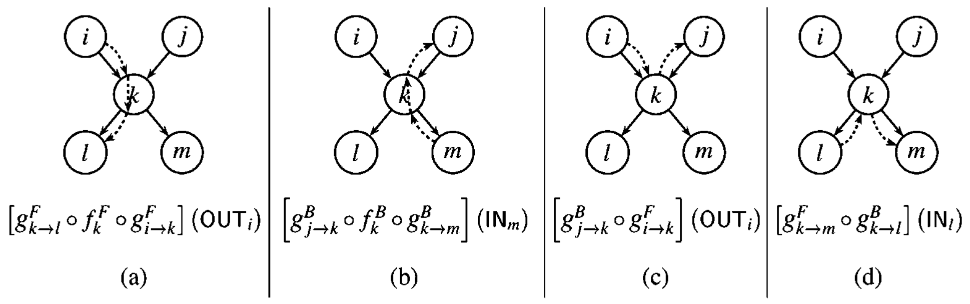

$[g^F_{k\to l} \circ f^F_k \circ g^F_{i\to k}]$ (OUT$_i$)
$[g^B_{j\to k} \circ f^B_k \circ g^B_{k\to m}]$ (IN$_m$)
$[g^B_{j\to k} \circ g^F_{i\to k}]$ (OUT$_i$)
$[g^F_{k\to m} \circ g^B_{k\to l}]$ (IN$_l$)
(a)
(b)
(c)
(d)

FIGURE 1.8 Characterizing general flows in data flow analysis.

1.4.2.4 Generic Data Flow Equations

It is possible to write generic data flow equations as follows:[11]

$$\texttt{IN}_i = \begin{cases} \textit{Outside\_Info} \sqcap f_i^B(\texttt{OUT}_i) & \text{if } i \text{ is a graph entry node} \\ \displaystyle\bigsqcap_{p\in pred(i)} g^F_{p\to i}(\texttt{OUT}_p) \sqcap f_i^B(\texttt{OUT}_i) & \text{otherwise} \end{cases} \tag{1.40}$$

$$\texttt{OUT}_i = \begin{cases} \textit{Outside\_Info} \sqcap f_i^F(\texttt{IN}_i) & \text{if } i \text{ is a graph exit node} \\ \displaystyle\bigsqcap_{s\in succ(i)} g^B_{i\to s}(\texttt{IN}_s) \sqcap f_i^F(\texttt{IN}_i) & \text{otherwise} \end{cases} \tag{1.41}$$

Note that these equations assume homogeneous confluence operations for all flows. Hence, they cannot be used for MMRA and EPA data flow analyses described in Appendix A. However, they are applicable to almost all other known data flow problems including constant propagation and type inferencing. These data flow equations are customized for unidirectional data flows as follows:

- For forward data flow problems, the backward flow functions do not exist; hence, g^B and f^B disappear along with their confluences. Also, for almost all unidirectional problems, the edge flow functions are identity functions; hence, even g^F disappears leaving behind its confluence operator.[12] The data flow equations reduce to Equations (1.34) and (1.35) where f appears in place of f^F.
- For backward data flow problems, the forward flow functions do not exist and g^F and f^F disappear along with their confluences. Because the typical edge flow functions are identity functions, even g^B disappears leaving behind its confluence operator. The data flow equations reduce to Equations (1.36) and (1.37) where f appears in place of f^B.

1.5 Theoretical Foundations of Data Flow Analysis

We have seen that data flow analysis is defined in terms of constraints on data flow information. In general, the data flow values are aggregates of booleans, multivalued scalars or tuples. The individual

[11]Actually we need a constant term for PRE that we ignore for simplicity.

[12]We do this in a more systematic and formal way in Section 1.5.

components of these aggregates capture the data flow information of a single data item. The basic operations that we need to perform on data flow values are:

- Comparison (larger or smaller)
- Merging ($\cup$, $\cap$ or special confluence)
- Computations using flow functions

These operations are used to specify the constraints defining the data flow information associated with a program point. Clearly, the operations of comparison and merging can be defined on aggregates only componentwise (i.e., we need to be able to define the operations of comparison and merging on each component of the aggregate). Computations using flow functions may not be definable in terms of aggregates of components.

1.5.1 Algebra of Data Flow Information

Formally, we define the algebra of data flow information in terms of a *semilattice* $\langle \mathscr{L}, \sqcap \rangle$, where $\mathscr{L}$ is a nonempty set and $\sqcap : \mathscr{L} \times \mathscr{L} \mapsto \mathscr{L}$ is a binary operation that is commutative, associative and idempotent. Relation $\sqcap$ computes the greatest lower bound (*glb*) of two elements in $\mathscr{L}$ and thereby induces a partial order relation[13] denoted by $\sqsubseteq$, which is transitive, reflexive and antisymmetric. The partial order is defined by:

$$\forall a, b \in \mathscr{L}: \quad a \sqcap b \sqsubseteq a \text{ and } a \sqcap b \sqsubseteq b \tag{1.42}$$

Note that $\sqsubseteq$ relation allows equality. Strict partial order, which excludes equality, is denoted by $\sqsubset$. Because the ordering is only partial instead of total, it is possible that two elements are incomparable, as indicated by $a \not\sqsubseteq b$ and $b \not\sqsubseteq a$.

For the purpose of data flow analysis, this semilattice has two special features:

- There are two special elements called *top* ($\top$) and *bottom* ($\bot$) such that:

$$\forall a \in \mathscr{L}, a \sqcap \top = a \quad \text{(i.e., } a \sqsubseteq \top\text{)} \tag{1.43}$$

$$\forall a \in \mathscr{L}, a \sqcap \bot = \bot \quad \text{(i.e., } \bot \sqsubseteq a\text{)} \tag{1.44}$$

  Such a semilattice is called a *complete* semilattice. Uniqueness of $\top$ and $\bot$ guarantee the uniqueness of minimal and maximal solutions.
- The length of every strictly descending chain $a_1 \sqsubset a_2 \sqsubset a_3 \sqsubset \cdots \sqsubset a_n$(i.e., the number of $\sqsubset$) is finite. We say that $\mathscr{L}$ has finite height. If this length is bounded by some constant l, we say that $\mathscr{L}$ has height l.

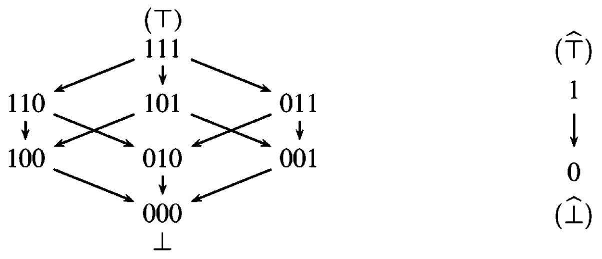
Because the number of data items can be large, it is hard to construct and understand the lattices for data flow values for aggregate data flow information for all the data items together. Instead we factor the aggregate data flow information in terms of the data flow information for individual data items and view the lattice for the aggregate data flow information as consisting of a (Cartesian) product of lattices (one for each data item). We use the notation of $\widehat{\mathscr{L}}$, $\widehat{\sqcap}$ and $\widehat{\sqsubseteq}$ for the component lattices.[14] If there are k data items, then:

- Lattice: $\mathscr{L} \equiv \langle \widehat{\mathscr{L}}_1, \widehat{\mathscr{L}}_2, \ldots, \widehat{\mathscr{L}}_k \rangle$
- Confluence operation: $\sqcap \equiv \langle \widehat{\sqcap}_1, \widehat{\sqcap}_2, \ldots, \widehat{\sqcap}_k \rangle$

[13]If we choose a different confluence operation, the partial order changes.

[14]Cartesian product of functions and relations imply pointwise products of the values in ranges.

x and y are comparable *iff* they fall on a path from $\top$ to $\bot$. Besides, $x \rightarrow y$ indicates $y \sqsubset x$.



(a) The product lattice $\mathcal{L}$ **(b) The component lattice $\widehat{\mathcal{L}}$**

FIGURE 1.9 Lattices for available expressions analysis where $\mathbf{U} = \{e_1, e_2, e_3\}$.

- Partial order: $\sqsubseteq \equiv \langle \widehat{\sqsubseteq}_1, \widehat{\sqsubseteq}_2, \ldots, \widehat{\sqsubseteq}_k \rangle$
- Top and bottom elements: $\top \equiv \langle \widehat{\top}_1, \widehat{\top}_2, \ldots, \widehat{\top}_k \rangle$, and $\bot \equiv \langle \widehat{\bot}_1, \widehat{\bot}_2, \ldots, \widehat{\bot}_k \rangle$.
- Height: For the product lattice we are interested in the *effective height*, which is defined as the maximum l_i, where l_i is the height of $\widehat{\mathscr{L}}_i$, $1 \leq i \leq k$.

Because all data items for which the analysis is performed are homogeneous[15] $\widehat{\mathscr{L}}$, $\widehat{\sqcap}$, $\widehat{\sqsubseteq}$, $\widehat{\top}$ and $\widehat{\bot}$ have similar structures across various components.

We present some examples of data flow information:

- *Available expressions analysis.* Let us assume that we have only three expressions in our program (i.e., $\mathbf{U} = \{e_1, e_2, e_3\}$). Then the sets of values are $\widehat{\mathscr{L}}_1 = \{\{e_1\}, \emptyset\}$, $\widehat{\mathscr{L}}_2 = \{\{e_2\}, \emptyset\}$, and $\widehat{\mathscr{L}}_3 = \{\{e_3\}, \emptyset\}$. The first value in each set indicates that the expression is available, whereas the second value indicates that the expression is not available. The confluence operation is $\widehat{\sqcap}_1 = \widehat{\sqcap}_2 = \widehat{\sqcap}_3 = \cap$ whereas the partial order is $\widehat{\sqsubseteq}_1 = \widehat{\sqsubseteq}_2 = \widehat{\sqsubseteq}_3 = \subseteq$. If we treat the set $\{e_1\}$ (or for that matter, $\{e_2\}$ and $\{e_3\}$) by a boolean value 1 and $\emptyset$ by boolean 0, we can view $\widehat{\mathscr{L}}_i$ as $\{1, 0\}$ and the product lattice as:

$$\mathscr{L} = \{\langle 111\rangle, \langle 110\rangle, \langle 101\rangle, \langle 100\rangle, \langle 011\rangle, \langle 010\rangle, \langle 001\rangle, \langle 000\rangle\}$$

  where we have dropped the customary "," between the elements in a tuple and a positional correspondence exists between the elements in a tuple and expressions e_1, e_2 and e_3. It is clear that $\subseteq$ over $\mathscr{L}$ as the product of $\subseteq$ over $\widehat{\mathscr{L}}$ and $\cap$ over $\mathscr{L}$ as the product of $\cap$ over $\widehat{\mathscr{L}}$ achieve the desired results. For $\widehat{\mathscr{L}}$, $\widehat{\top} = 1$ and $\widehat{\bot} = 0$ whereas for $\mathscr{L}$, $\top = 111$ and $\bot = 000$. The height of $\mathscr{L}$ is 3 whereas the height of $\widehat{\mathscr{L}}$ (and hence the effective height of $\mathscr{L}$) is 1. Figure 1.9 illustrates the lattices.
- *Reaching definitions analysis.* In this case $\widehat{\sqcap}$ is $\cup$, $\widehat{\sqsubseteq}$ is $\supseteq$, $\widehat{\top}$ is 0 whereas $\widehat{\bot}$ is 1. The structure of the lattice would be the same as in Figure 1.9 except that it would be drawn upside down (or 1's and 0's would get exchanged).
- *Constant propagation.* We illustrate the component lattice $\widehat{\mathscr{L}}$ assuming m constants $c_1, c_2, \ldots, c_m$ in Figure 1.10. Note that the effective height of the product lattice is 2 (even when we allow infinite number of constants).

[15]All of them belong to the same class, namely, expressions or variables, etc.

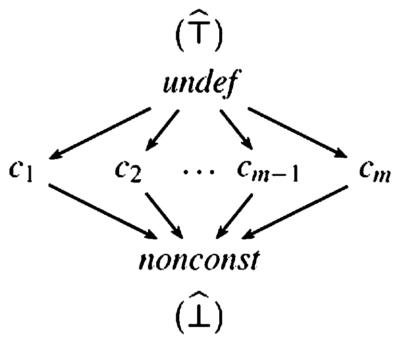


FIGURE 1.10 Component lattice $\widehat{\mathscr{L}}$ for constant propagation (assuming m constants $c_1, c_2, \ldots, c_m$).

1.5.1.1 Interpreting the Algebra of Data Flow Information

This section relates the mathematical definitions of data flow information with the intuitions that we have developed in Section 1.3. These explanations are applicable to both the product as well as the component's lattices:

- The top element $\top$ is the largest element and serves as an initializer to begin the data flow analysis (Section 1.3.7.1). It is a conservative approximation because it does not affect any value (i.e., $a \sqcap \top = a$. The $\top$ element may not be a natural element of the lattice and a fictitious $\top$ may have to be devised.
- The bottom element $\bot$ is the smallest element and signifies that among all possible values of data flow information the most conclusive (i.e., smallest) value has been reached. Because $a \sqcap \bot = \bot$, a value that is $\bot$ changes other values (unless the other value is also $\bot$).
- The properties of the confluence $\sqcap$ include:
  - *Commutativity.* $\forall a, b \in \mathscr{L}\colon\ a \sqcap b = b \sqcap a$. When we merge the information, the order of merging (or the order in which different predecessors or successors are visited) should not matter.
  - *Associativity.* $\forall a, b, c \in \mathscr{L}\colon\ (a \sqcap b) \sqcap c = a \sqcap (b \sqcap c)$. It should be possible to extend the merge operation to more than two values (possibly coming from multiple predecessors or successors) without needing a specific order of merging.
  - *Idempotence.* $\forall a \in \mathscr{L},\ a \sqcap a = a$. Merging a data flow value with itself should not produce a different value.
- The partial order $\sqsubseteq$ captures the notion of information theoretic largeness. Definition (1.42) should be read as: the order used for comparison is defined by saying that when two data flow values are combined, the result cannot exceed either of the values. The actual sets representing these values may be smaller or larger (in set theoretical terms); the confluence may be $\cap$, $\cup$ or special confluence — appropriate ordering is automatically induced. It is easy to conclude that:
  - $\forall a, b \in \mathscr{L}\colon\ a \sqsubseteq b \Leftrightarrow a \sqcap b = a$. If two data flow values are comparable, the result of merging them is the same as the smaller value (with regard to the partial order).
  - $\forall a, b \in \mathscr{L}\colon\ a \not\sqsubseteq b$ and $b \not\sqsubseteq a \Leftrightarrow a \sqcap b \sqsubset a$ and $a \sqcap b \sqsubset b$. If two data flow values are incomparable, the result of merging them is different from both and is strictly smaller than both.
- The properties of partial order relation $\sqsubseteq$ include:
  - *Transitivity.* $\forall a, b, c \in \mathscr{L}\colon\ a \sqsubseteq b$ and $b \sqsubseteq c \Rightarrow a \sqsubseteq c$. Because we need to find out the largest solution satisfying the constraints, the comparison better be transitive.

- *Reflexivity.* $\forall a \in \mathscr{L},\ a \sqsubseteq a$. A data flow value must be comparable to itself.
- *Antisymmetry.* $\forall a, b \in \mathscr{L}:\ a \sqsubseteq b$ and $b \sqsubseteq a \ \Leftrightarrow\ a = b$. If a data flow value a is smaller or equal to b, and b is also smaller or equal to a, then this is possible if and only if they are identical.

1.5.2 World of Flow Functions

We extend the algebra of data flow information by including flow functions to capture the effect of a node and an edge (i.e., transfer of control) on the data flow information. We define our *function space* to be $\mathscr{H} \subseteq \{h \mid h : \mathscr{L} \mapsto \mathscr{L}\}$ such that it contains an identity function ι and is closed under composition. Note that $\mathscr{H}$ contains both node flow as well as edge flow functions. When there is a need to distinguish between node and edge flow functions, we use the earlier notation in which f^F and f^B denote node flow functions whereas g^F and g^B denote edge flow functions. When no need exists to distinguish between the two, we use h to denote a function.

Function spaces are characterized by many interesting properties that influence data flow analysis.

1.5.2.1 Monotonic Function Space

$\mathscr{H}$ is monotonic *iff* the results of the functions either do not decrease for increasing inputs or do not increase for decreasing inputs. Formally:

$$\forall a, b \in \mathscr{L}, \forall h \in \mathscr{H}:\ a \sqsubseteq b \Rightarrow h(a) \sqsubseteq h(b) \tag{1.45}$$

This is necessary for convergence on a fixed point (but is not necessary for its existence). Another way of looking at monotonicity is:

$$\forall\, a, b \in \mathscr{L}, \forall\, h \in \mathscr{H}:\ \ h(a\ \sqcap\ b) \sqsubseteq h(a)\ \sqcap\ h(b) \tag{1.46}$$

This implies that the result of merging the information before applying a function cannot exceed the result if the functions were applied first and the results merged later. Recall that at every stage we want to show that the information would not increase or else it might violate our data flow constraints; decreased information may be less useful, but it is provably correct.

By way of an example, function $h : \{0, 1\} \mapsto \{0, 1\}$, such that $h(0) = 1$ and $h(1) = 0$, is not monotonic. In general, a function is monotonic if and only if it does not negate its arguments, where the semantics of negate depends on the domain of the function.

1.5.2.2 Distributive Function Space

$\mathscr{H}$ is distributive *iff* merging the information at a node does not lead to loss of useful information (i.e., the decrease that is possible in monotonicity is ruled out in distributivity).

$$\forall\, a, b \in \mathscr{L}, \forall\, h \in \mathscr{H}:\ \ h(a\ \sqcap\ b) = h(a)\ \sqcap\ h(b) \tag{1.47}$$

This is a stronger condition than monotonicity and guarantees that the maximum fixed point is the same as the maximum safe assignment because no approximation takes place.

The function space of constant propagation is monotonic but not distributive. Let us consider the example in Figure 1.5, the flow function definition (1.27) and the confluence definition (1.30). Let

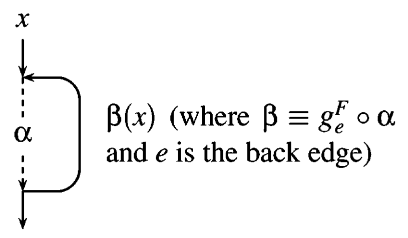

x
α
$\beta(x)$ (where $\beta \equiv g_e^F \circ \alpha$
and e is the back edge)

FIGURE 1.11 Defining loop closure.

the data flow information along the path $1 \rightarrow 2 \rightarrow 4 \rightarrow 5$ be denoted by a and that along the path $1 \rightarrow 3 \rightarrow 4 \rightarrow 5$ be denoted by b. Then:

$$\begin{aligned}
a &= \{\langle x, 4\rangle, \langle y, 6\rangle, \langle z, ud\rangle\} \\
b &= \{\langle x, 6\rangle, \langle y, 4\rangle, \langle z, ud\rangle\} \\
a \sqcap b &= \{\langle x, nc\rangle, \langle y, nc\rangle, \langle z, ud\rangle\} \\
f_5^F(a \sqcap b) &= \{\langle x, nc\rangle, \langle y, nc\rangle, \langle z, nc\rangle\} \\
f_5^F(a) &= \{\langle x, 4\rangle, \langle y, 6\rangle, \langle z, 24\rangle\} \\
f_5^f(b) &= \{\langle x, 6\rangle, \langle y, 4\rangle, \langle z, 24\rangle\} \\
f_5^F(a) \sqcap f_5^F(b) &= \{\langle x, nc\rangle, \langle y, nc\rangle, \langle z, 24\rangle\} \\
f_5^F(a \sqcap b) &\sqsubset f_5^F(a) \sqcap f_5^F(b) \qquad \ldots \text{(from Figure 1.10, } \langle z, nc\rangle \sqsubset \langle z, 24\rangle)
\end{aligned}$$

1.5.2.3 Bounded Function Space

A monotonic $\mathscr{H}$ is bounded *iff* its *loop closures* (i.e., the number of times a loop may have to be traversed during data flow analysis) are bounded (Figure 1.11). Consider the following loop where α is the flow function across the body of the loop.

Before entering the loop for the second time, x is merged with $\beta(x)$ (which is the information along the back edge); note that $\beta \equiv g_e^F \circ \alpha$ and by the closure of $\mathscr{H}$ under composition, $\beta \in \mathscr{H}$. Thus, the second traversal produces $\alpha(x \sqcap \beta(x))$ at the bottom of the loop and the merge at the top of the loop is $x \sqcap \beta(x \sqcap \beta(x))$. We can rewrite this as:

$$x \sqcap \beta(x \sqcap \beta(x)) \sqsubseteq x \sqcap \beta(x) \sqcap \beta(\beta(x)) \quad \ldots (\beta \text{ is monotonic})$$

If we denote the merge at the top of the loop after i traversals as m^i, then:

$$m^i \sqsubseteq \beta^0(x) \sqcap \beta^1(x) \sqcap \cdots \sqcap \beta^i(x) \text{ where } \beta^i(x) = [\beta \circ \beta^{i-1}](x),\ i \geq 1 \text{ and } \beta^0 = \iota$$

The loop closure for a function h is denoted by h^* and can be defined as follows:[16]

$$h^* \equiv h^0 \sqcap h^1 \sqcap h^2 \sqcap \ldots \tag{1.48}$$

It is desirable that the loop closures of $\mathscr{H}$ be bounded by a constant; we say that $\mathscr{H}$ is *k bounded iff*:

$$\forall\, h \in \mathscr{H},\ \exists\, k \geq 1 : \bigsqcap_{i=0}^{k-1} h^i = h^* \tag{1.49}$$

[16]Merging of functions implies pointwise merging of the values in their ranges.

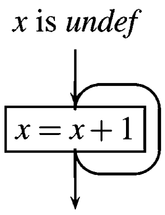


FIGURE 1.12 Loop closure in constant propagation is not bounded.

The boundedness of loop closure h^* requires the existence of a fixed point of h, which in general requires the product lattice $\mathscr{L}$ to have a finite height.[17] Further, convergence on the fixed point requires h to be monotonic and defined on all points common to its domain and range.

It is easy to show that the function space in available expressions analysis is 2-bounded but the function space of constant propagation is not bounded; the latter is illustrated in Figure 1.12. In this case every time the loop is traversed, the analysis finds that x is indeed a constant albeit with a new value; the flow function is monotonic but it does not have a fixed point.

1.5.2.4 Separable Function Space

$\mathscr{H}$ is separable *iff* it can be factored into a product of the functions' spaces for individual data items:

$$\mathscr{H} \equiv \langle \widehat{\mathscr{H}}_1, \widehat{\mathscr{H}}_2, \ldots, \widehat{\mathscr{H}}_k \rangle, \text{ where } \widehat{\mathscr{H}}_i \subseteq \{\widehat{h} \mid \widehat{h} : \widehat{\mathscr{L}}_i \mapsto \widehat{\mathscr{L}}_i\} \tag{1.50}$$

In other words, the functions on the aggregate data flow information can be viewed as products of functions on the individual data flow items and the component functions work only on the component lattices:

$$\forall h \in \mathscr{H},\ h \equiv \langle \widehat{h}_1, \widehat{h}_2, \ldots, \widehat{h}_k \rangle, \text{ where } \widehat{h}_i \in \widehat{\mathscr{H}}_i \tag{1.51}$$

Separability guarantees that data flow analysis can be performed on different data items independently and the result can be combined without any loss of information. Constant propagation does not have a separable function space (recall from Section 1.3.6.2 that constantness of z cannot be determined independently of the constantness of x and y.) Actually, the flow function for constant propagation can be factored into component functions for each variable. However, these functions would be of the form $\widehat{h} : \mathscr{L} \mapsto \widehat{\mathscr{L}}$ instead of the desired form $\widehat{h} : \widehat{\mathscr{L}} \mapsto \widehat{\mathscr{L}}$; the former indicates that a function for a single data item requires the data flow information of all data items whereas the latter indicates that data flow information of only the same data item is required.

We leave it to the reader to verify that if a monotone separable function space $\mathscr{H}$ is defined over a lattice with effective height l, then the data flow values can change at most l times and $\mathscr{H}$ is $l + 1$ bounded.

1.5.2.5 Bit Vector Function Space

$\mathscr{H}$ is a bit vector function space *iff* it is separable and monotonic, and its data flow values can be sets of booleans. In other words, $\widehat{\mathscr{L}} = \{1, 0\}$, and the functions that negate its arguments, that is, $\widehat{h}(0) = 1$ and $\widehat{h}(1) = 0$, are prohibited. Clearly, a bit vector function space is distributive. Most of the traditional analyses use bit vector function spaces; in fact, barring constant propagation and KDM

[17]Note that even if the component lattice $\widehat{\mathscr{L}}$ has finite height, its product lattice $\mathscr{L}$ may not have finite height if $\widehat{\mathscr{L}}$ is infinite. This can be easily verified in the case of constant propagation.

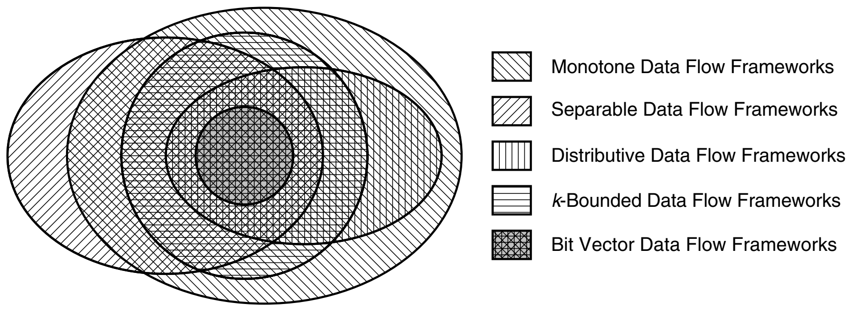

Monotone Data Flow Frameworks
Separable Data Flow Frameworks
Distributive Data Flow Frameworks
k-Bounded Data Flow Frameworks
Bit Vector Data Flow Frameworks

FIGURE 1.13 Relationships between various categories of data flow frameworks.

type inferencing, all data flow analyses mentioned in this chapter use bit vector function spaces. Two interesting properties of the bit vector function spaces are:

- $\widehat{\mathscr{H}}$ is a bit vector function space *iff* its component functions can be written as $\widehat{h}(x) = a + b \cdot x$ where $a, b, x \in \{0, 1\}$.
- A bit vector function space is 2-bounded. Such functions spaces are, for obvious reasons, called *fast*.

1.5.3 Data Flow Frameworks

The algebra of data flow analysis is defined by extending the algebra of data flow information to include the function space. It is called a *data flow framework* and is formally defined as $\langle \mathscr{L}, \sqcap, \mathscr{H} \rangle$. Clearly, a data flow framework defines the domain of data flow information, the confluence operation and the flow function space, thereby providing a rigorous mathematical foundation for the interplay of data flow information. All the adjectives of the function spaces carry over to data flow frameworks. Accordingly, data flow frameworks can be classified as *monotone*, *distributive, bounded, separable*, or *bit vector* data flow frameworks. In general, monotone data flow frameworks form the largest class for which performing data flow analysis is meaningful. Figure 1.13 illustrates the relationships between various categories of data flow frameworks.

Constant propagation is an example of a framework that is monotone but neither distributive, separable nor bounded. Available expressions analysis is monotone, distributive, separable, bounded and bit vector. The combined may-availability and must-availability analysis is monotone, distributive, separable and bounded, but not bit vector (refer to References Section). A (fictitious) example of a framework that is monotone, bounded and separable but not distributive has been provided in Figure 1.14. KDM type inferencing for statically checked languages is monotone, non-separable and distributive. Interestingly, KDM type inferencing for dynamically checked languages is monotone, non-separable and non-distributive.

1.5.4 Instance of a Data Flow Framework

Because a program to be analyzed is not a part of a data flow framework, the actual association of flow functions with nodes and edges and the eventual association of data flow information with the nodes requires instantiating the data flow framework with the program-specific information. Accordingly, an *instance of a data flow framework* is defined by a tuple $\langle G, M_N, M_E \rangle$ where:

- $G = \langle N, E, n_0, n_\infty \rangle$ is a directed graph consisting of a set of nodes N, a set of edges E, a unique graph entry node n_0 with indegree zero and a unique graph exit node n_∞ with outdegree zero. Self-loops are allowed in the graph but parallel edges between nodes are not

$$\begin{tikzcd}
& \widehat{\top} \arrow[dl] \arrow[dr] & \\
v_1 \arrow[dr] & & v_2 \arrow[dl] \\
& v_3 \arrow[d] & \\
& \widehat{\bot} &
\end{tikzcd}$$

$$\begin{array}{l}
\hat{\mathcal{L}} = \{\hat{\top}, v_1, v_2, v_3, \hat{\bot}\} \\
\hat{\mathcal{H}} : \hat{\mathcal{L}} \mapsto \hat{\mathcal{L}} \; is \; \{\iota, \hat{h}\} \\
\text{where } \hat{h}(\hat{\top}) = v_1 \\
\quad \hat{h}(v_1) = v_3 \\
\quad \hat{h}(v_2) = v_3 \\
\quad \hat{h}(v_3) = \hat{\bot} \\
\quad \hat{h}(\hat{\bot}) = \hat{\bot}
\end{array}$$

Functions in $\mathscr{H}$ are monotonic and separable but not distributive

$$\begin{aligned}
\hat{h}(v_1 \sqcap v_2) &= \hat{h}(v_3) \\
&= \hat{\bot} \\
\hat{h}(v_1) \sqcap \hat{h}(v_2) &= v_3 \sqcap v_3 \\
&= v_3 \\
\hat{h}(v_1 \sqcap v_2) &\sqsubset \hat{h}(v_1) \sqcap \hat{h}(v_2)
\end{aligned}$$

FIGURE 1.14 A data flow framework that is monotone and separable but not distributive.

allowed.[18] The uniqueness of n_0 and n_∞ are required basically for mathematical convenience. It is always possible to satisfy this condition by adding dummy nodes to the graph with edges from the dummy n_0 to actual multiple entry nodes and from the actual multiple exit nodes to the dummy n_∞.

This graph is usually called a *control flow graph*. Strictly speaking, a control flow graph does not contain the computations of a program. In practice we use a graph whose nodes also contain the computations (e.g., Figure 1.2); such graphs are called *program flow graphs*.

- $M_N : N \times \{F, B\} \mapsto \mathscr{H}$ is a function that associates the flow functions with graph nodes as either forward (denoted F) or backward (denoted B) node flow functions. For flow function definition (1.8), the data flow framework of available expressions analysis is actually a subset of the mapping M_N of the form $N \times \{F\} \mapsto \mathscr{H}$. In other words (1.8) defines the forward node flow functions f_i^F. The absence of backward node flow functions (i.e., the other subset $N \times \{B\} \mapsto \mathscr{H}$) implies 0-ary constant functions that return the value $\top$. Recall that $a \sqcap \top = a$ for all values of a. Thus, these flow functions do not affect analysis in any adverse way; it is as good as ignoring the existence of these flow functions. This is precisely the meaning of flow functions disappearing from data flow Equations (1.40) and (1.41) in Section 1.4 in the case of unidirectional data flow problems (refer to footnote 12).
- $M_E : E \times \{F, B\} \mapsto \mathscr{H}$ is a function that associates the flow functions with edges. For available expressions analysis, this sets the forward edge flow functions to identity functions ι (because the edges merely propagate the information in the forward direction without modifying it) and the backward edge flow functions to constant $\top$ function (because the edges do not propagate any information against the control flow).

Intuitively, a program flow graph forms the basis of defining G, M_N and M_E.

1.5.5 Information Flow Paths and the Path Flow Function

A fundamental insight that emerges from various observations is:

The data flow information at a program point is always contained in (or cannot exceed) the information that is carried to that point by the flow functions associated with the nodes that appear on a path reaching the program point.

[18]This is a fair assumption for imperative programs on Von Neumann machine architectures.

These containments are captured by the constraints. In Section 1.3.6.1, we took a transitive closure of these containments along various paths for available expressions analysis. Thus, we need to compose the flow functions along the paths to define the data flow information at a node.

1.5.5.1 Information Flow Paths

Our graphs are directed graphs and the graph theoretical paths are not adequate to characterize the data flows. For instance, no path exists from node i to node j in Figure 1.8(c) and yet the data flow information at i may influence the data flow information at j and vice versa. Similarly, data flow may take place from node l to node m in Figure 1.8(d). Intuitively, these flows take place along paths in the underlying undirected graph. However, the directions and the distinction between successors and predecessors cannot be ignored. Hence, we define a new concept of *information flow path* (*ifp*). It is a sequence of program points $\texttt{entry}_i$ and $\texttt{exit}_i$ for all nodes i for capturing the order in which the flow functions could be composed to compute the data flow information reaching a node. Thus the data flow indicated by the dashed line in Figure 1.8(c) is taking place along the *ifp* $(\texttt{exit}_i, \texttt{entry}_k, \texttt{exit}_j)$ whereas the data flow in Figure 1.8(d) is taking place along the *ifp* $(\texttt{entry}_l, \texttt{exit}_k, \texttt{entry}_m)$. It is easy to see that if a data flow framework is k bounded, a particular program point needs to appear in its *ifp's* at most $k-1$ times because loop closure is reached in $k-1$ traversals over the loop.

We denote an *ifp* from a program point u to a program point v by $\langle u, v\rangle$ or by listing all program points between u and v along the *ifp*, namely, $(q_0 \equiv u, q_1, \ldots, q_i, q_{i+1}, \ldots, q_k \equiv v)$. Composing all node and edge flow functions (in proper order) along this path gives us the path flow function with which the data flow information reaching v from u can be computed. To understand the order of function compositions:

- Let q_i be $\texttt{entry}_m$. Then q_{i+1} can be:
    - $\texttt{exit}_m$. In this case, the flow is from $\texttt{entry}_m$ to $\texttt{exit}_m$ and hence we need to apply forward node flow function f_m^F.
    - $\texttt{exit}_p$, $p \in pred(m)$. In this case, the flow is from $\texttt{entry}_m$ to $\texttt{exit}_p$ and hence we need to apply backward edge flow function $g_{p\to m}^B$.
- Let q_i be $\texttt{exit}_m$. Then q_{i+1} can be:
    - $\texttt{entry}_m$. In this case, the flow is from $\texttt{exit}_m$ to $\texttt{entry}_m$ and hence we need to apply backward node flow function f_m^B.
    - $\texttt{entry}_s$, $s \in succ(m)$. In this case, the flow is from $\texttt{exit}_m$ to $\texttt{entry}_s$ and hence we need to apply forward edge flow function $g_{m\to s}^F$.

Note that in both the situations, the two options are mutually exclusive for unidirectional data flow problems. This implies that an *ifp* has a well-defined structure that depends on the existence of flow functions in a data flow framework.

1.5.5.2 Path Flow Function

A *path flow function* captures the compositions of flow functions along an *ifp*. Let $PP = \{\texttt{entry}, \texttt{exit}\} \times N$ be the set of all program points of an instance $\langle G, M_N, M_E\rangle$ and let *IFP* be the set of all *ifp*'s of the instance. *IFP* includes the *null ifp*'s (i.e., an *ifp* consisting of a single program point) too. We define the following operations on *IFP* and *PP*.

- Construction # : $IFP \times PP \mapsto IFP$. This operation adds a program point to an *ifp* to create a longer *ifp*. Traversing a program point p farther from an *ifp* ρ constructs a new *ifp* $\rho' = \rho \# p$.
- Extracting the last program point $last : IFP \mapsto PP$. If $\rho' = \rho \# p$ then $last(\rho') = p$.

By using the preceding abstractions, we define the path flow function $M_P : IFP \mapsto \mathscr{H}$ recursively:

$$M_P(\rho') = \begin{cases} \iota & \rho = null \text{ (i.e., identity function for null } ifp) \\ M_N(m, F) \circ M_P(\rho) & last(\rho) \equiv \texttt{entry}_m,\ \rho' = \rho \# \texttt{exit}_m \\ M_E(p \to m, B) \circ M_P(\rho) & last(\rho) \equiv \texttt{entry}_m,\ \rho' = \rho \# \texttt{exit}_p,\ p \in pred(m) \\ M_N(m, B) \circ M_P(\rho) & last(\rho) \equiv \texttt{exit}_m,\ \rho' = \rho \# \texttt{entry}_m \\ M_E(m \to s, F) \circ M_P(\rho) & last(\rho) \equiv \texttt{exit}_m,\ \rho' = \rho \# \texttt{entry}_s,\ s \in succ(m) \end{cases} \tag{1.52}$$

Figure 1.15 illustrates the compositions in the second case of definition (1.52). The dashed double arrow indicates the *ifp*, whereas the solid arrow indicates the flow functions along the *ifp* fragments. The dashed boxes indicate the program points of interest.

For a forward unidirectional data flow problem, $M_E(p \to m, B)$ and $M_N(m, B)$ map the edge and node flow functions to constant $\top$ function (hence, they are as good as nonexistent). Besides, $M_E(m \to s, F)$ is an identity function. Thus, the path flow function reduces to:

$$M_P(\rho) = \begin{cases} \iota & \rho = null \text{ (i.e., identity function for null } ifp) \\ M_N(m, F) \circ M_P(\rho) & last(\rho) \equiv \texttt{entry}_m,\ \rho' = \rho \# \texttt{exit}_m \\ M_P(\rho) & last(\rho) \equiv \texttt{exit}_m,\ \rho' = \rho \# \texttt{entry}_s,\ s \in succ(m) \end{cases} \tag{1.53}$$

For available expressions analysis, the first case allows us to capture the boundary condition of computing $\texttt{AVIN}_{n0}$ as *Outside_Info* (Equation (1.34)). The second case captures the effect of applying f_i to $\texttt{AVIN}_i$ in constraint (1.3); $\texttt{AVIN}_i$ is computed using $M_P(\rho)$ in this case. The third case captures the effect of constraint (1.1); in this case $M_P(\rho)$ computes $\texttt{AVOUT}_i$.

1.5.6 Solutions of an Instance of a Data Flow Framework

This section formalizes the observations made in Section 1.3.4.2. We define safe and fixed point assignments; their maximality gets defined by the partial order relation $\sqsubseteq$.

1.5.6.1 Safe Assignment

Let $\mathscr{P}_{IN} : N \mapsto IFP$ denote the set of *ifp*'s reaching $\texttt{entry}_i$ from either $\texttt{entry}_{n0}$ or $\texttt{exit}_{n\infty}$. Let $\mathscr{P}_{OUT} : N \mapsto IFP$ denote the set of *ifp*'s reaching $\texttt{exit}_i$ from either $\texttt{entry}_{n0}$ or $\texttt{exit}_{n\infty}$. Then:

$$\mathscr{P}_{IN(i)} = \{\rho \mid \rho = \langle \texttt{entry}_{n0}, \texttt{entry}_i \rangle \text{ or } \rho = \langle \texttt{exit}_{n_\infty}, \texttt{entry}_i \rangle\}$$

$$\mathscr{P}_{OUT(i)} = \{\rho \mid \rho = \langle \texttt{entry}_{n_0}, \texttt{exit}_i \rangle \text{ or } \rho = \langle \texttt{exit}_{n_\infty}, \texttt{exit}_i \rangle\}$$

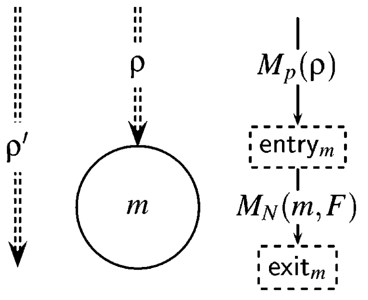


FIGURE 1.15 Computing the path flow function.

A safe assignment ($\texttt{SA}$): $N \mapsto \mathscr{L}$ is an assignment of value pairs $\langle \texttt{SAIn}, \texttt{SAOut} \rangle$ such that the data flow information cannot exceed what can reach a node from the outside world:

$$\forall i \in N, \forall \rho \in \mathscr{P}_{IN(i)},\ \texttt{SAIn}_i \sqsubseteq [M_P(\rho)]\,(\mathit{Outside\_Info}) \tag{1.54}$$

$$\forall i \in N, \forall \rho \in \mathscr{P}_{OUT(i)i},\ \texttt{SAOut}_i \sqsubseteq [M_P(\rho)]\,(\mathit{Outside\_Info}) \tag{1.55}$$

The maximum safe assignment ($\texttt{MSA}$): $N \mapsto \mathscr{L}$ is $\langle \texttt{MSAIn}, \texttt{MSAOut} \rangle$ such that:

$$\begin{aligned} \forall i \in N, \forall\ \texttt{SAIn}_i &: \quad \texttt{SAIn}_i \sqsubseteq \texttt{MSAIn}_i \\ \forall i \in N, \forall\ \texttt{SAOut}_i &: \quad \texttt{SAOut}_i \sqsubseteq \texttt{MSAOut}_i \end{aligned}$$

Clearly, $\texttt{MSA}$ can be computed by:

$$\texttt{MSAIn}_i = \bigsqcap_{\rho \in \mathscr{P}_{IN(i)}} [M_P(\rho)]\,(\mathit{Outside\_Info}) \tag{1.56}$$

$$\texttt{MSAOut}_i = \bigsqcap_{\rho \in \mathscr{P}_{OUT(i)}} [M_P(\rho)]\,(\mathit{Outside\_Info}) \tag{1.57}$$

It can be proven that the MSA can be computed by traversing all *ifp*'s independently without merging the data flow information at intermediate program points.

1.5.6.2 Fixed Point Assignment

A fixed point assignment of an instance of a data flow framework is a fixed point solution of data flow Equations (1.40) and (1.41).

Recall that a fixed point of a function $h : A \mapsto A$ is a value $x \in A$ such that $h(x) = x$. To relate this concept to the fixed point of data flow equations, recall that in general data flow properties are recursively dependent on their own values. From the discussion in Section 1.3.7, this can be viewed as:

$$\texttt{IN}_i = h_R(\texttt{IN}_i) \tag{1.58}$$

where h_R is constructed by composing and merging functions along the information flow paths that lead to recursive dependence.[19] Thus, computations using data flow Equations (1.40) and (1.41) can also be viewed as computations using equations of the kind (1.58). When the values cannot change any further, we say we have computed a fixed point assignment. From the basic fixed point theorem (also known as Tarski's lemma), the maximum fixed point assignment is computed by $h_R^k(\top)$ when $h_R^{k+1}(\top) = h_R^k(\top)$.

In other words, to compute the maximum fixed point assignment, one should initialize $\texttt{IN}_i$ and $\texttt{OUT}_i$ values to $\top$ and compute a fixed point assignment. It is easy to show that the maximum fixed point assignment contains all other fixed point assignments. It can also be proved that the maximum fixed point assignment can be computed by traversing all *ifp*'s by factoring out shared *ifp*'s and merging the information at intermediate program points.

1.5.6.3 Comparing Various Assignments

A large amount of information does not necessarily imply a large amount of useful information. For information to be useful, it has to be correct (i.e., the assignment must be safe) and precise (i.e., maximum safe). The relationship between the precision of information and the amount of information is captured by Figure 1.16.

[19]In the absence of a recursive dependence, h_R is a constant function.

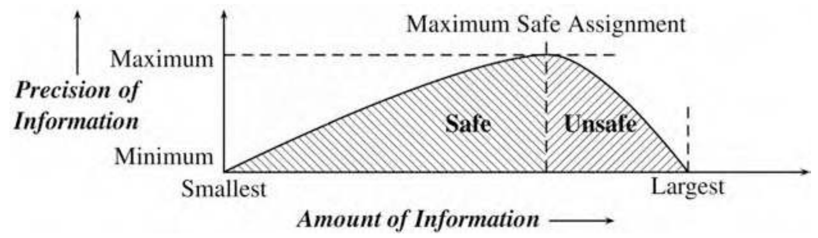
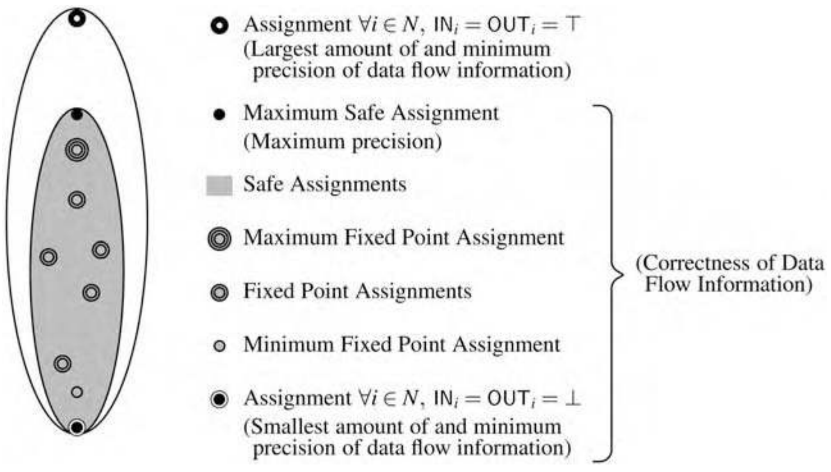


FIGURE 1.16 Amount of information and precision of information.



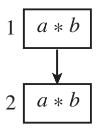
Assignment $\forall i \in N$, $\text{IN}_i = \text{OUT}_i = \top$ (Largest amount of and minimum precision of data flow information)
Maximum Safe Assignment (Maximum precision)
Safe Assignments
Maximum Fixed Point Assignment
Fixed Point Assignments
Minimum Fixed Point Assignment
Assignment $\forall i \in N$, $\text{IN}_i = \text{OUT}_i = \bot$ (Smallest amount of and minimum precision of data flow information)
(Correctness of Data Flow Information)

FIGURE 1.17 Assignments of an instance of a monotone data flow framework.

Figure 1.17 illustrates various possibilities of assignments to data flow properties. The largest amount of information in Figure 1.16 corresponds to the assignment $\texttt{IN}_i = \texttt{OUT}_i = \top$ in Figure 1.17 whereas the smallest amount of information corresponds to the assignment $\texttt{IN}_i = \texttt{OUT}_i = \bot$ for all nodes i.

For general monotone data flow frameworks, the MSA and maximum fixed point assignments may be different, in which case the maximum fixed point assignment is contained in and is not equal to the MSA. This is not the case for distributive data flow frameworks. Unfortunately, the general problem of computing the MSA is undecidable. The intuitive reason behind this is that for computing MSA, we need to traverse all information flow paths in a graph separately; common traversals cannot be factored out. In the presence of cyclic *ifp*'s this implies infinite number of *ifp* traversals, which themselves are infinitely long. Clearly, for bounded monotone frameworks, the problem of computing the MSA is decidable but may be intractable.

Because no deterministic polynomial time algorithm exists for computing the MSA, we settle for the next best option — the maximum fixed point assignment, which for all practical purposes forms the goal of data flow analysis. For the class of distributive data flow frameworks (which is a large class), this implies computing the MSA also because the two assignments are identical. As the next section shows, reasonably efficient methods of computing fixed point assignments are available.



Flow Functions	***Constraints***	***Safe Assignments***
$f_1^F(X) = X \cup \{a * b\}$	$\texttt{AVIN}_1 = \mathit{Outside\_Info} = \emptyset$	$\texttt{SAIn}_1 \subseteq \emptyset$
$g_{(1\to 2)}^F(X) = X$	$\texttt{AVOUT}_1 \subseteq \{a * b\}$	$\texttt{SAOut}_1 \subseteq f_1^F(\emptyset)$
	$\texttt{AVIN}_2 \subseteq \texttt{AVOUT}_1$	$\texttt{SAIn}_1 \subseteq f_2^F(\emptyset)$
$f_2^F(X) = X \cup \{a * b\}$	$\texttt{AVOUT}_2 \subseteq \texttt{AVIN}_2 \cup \{a * b\}$	$\texttt{SAOut}_2 \subseteq [f_2^F \circ f_1^F](\emptyset)$

Assignment $\texttt{AVIN}_1$= $\texttt{AVOUT}_1 \emptyset$, $\texttt{AVIN}_2$=$\texttt{AVOUT}_2 = \{a * b\}$ is safe but does not satisfy the constraints (since $\texttt{AVIN}_2 \subseteq \texttt{AVOUT}_1$).

FIGURE 1.18 How safe is a safe assignment for available expressions analysis?

1.5.6.4 An Aside on Safety

The definition of safety is quite subtle. It does not use the individual constraints but instead uses the path flow function that is a composition of the flow functions in the constraints. Thus, it is possible to have three kinds of assignments:

- Assignment satisfying definitions (1.54) and (1.55)
- Assignment satisfying constraints that use inequalities
- Assignment satisfying constraints that use equations

An assignment that satisfies the constraints can definitely satisfy definitions (1.54) and (1.55). However, definitions (1.54) and (1.55) admit even those assignments that do not satisfy the constraints. This is explained using a simple program flow graph in Figure 1.18.

Assignment $\texttt{AVIN}_1 = \texttt{AVOUT}_1 = \emptyset$, $\texttt{AVIN}_2 = \texttt{AVOUT}_2 = \{a*b\}$ can be safely used to eliminate the computation of $a * b$ in node 2. However, if it is used to decide whether the value of $a * b$ should be preserved in node 1 for a subsequent use in node 2, we discover that because $\texttt{AVOUT}_1$ is $\emptyset$, no expression needs to be preserved. This is clearly incorrect and the fault does not lie with the modeling of analysis but with the use of information derived by the analysis. When an expression is discovered to be available, it is guaranteed to be available; when an expression is discovered to be not available, it is not guaranteed to not be available; hence, such the negation of the derived information should be avoided.

1.6 Solution Methods of Data Flow Analysis

As noted in Section 1.3.7, if no recursive dependence of data flow properties existed, computing the solutions would have been simple. We also observed that basically performing data flow analysis boils down to dealing with these recursive dependences. The two broad approaches that we saw were:

- Circumventing recursive dependences by assuming conservative initial values and then recomputing them progressively until they stabilize
- Eliminating recursive dependences by delaying the computations by substituting a group of interdependent data flow properties by new properties and then computing the values of the properties in the group.

These two methods form the two broad categories of the solution methods of data flow analysis called *iterative* and *elimination* methods.

1.6.1 Iterative Methods of Data Flow Analysis

These methods assume a conservative initial value of data flow properties for each node and then progressively recompute them until they stabilize. From Figures 1.16 and 1.17, it is clear that using the $\top$ value for every program point is the simplest choice for a suitable conservative initialization.

```
procedure $round\_robin\_dfa()$
{   for each node $i$     /* Perform Initialization */
    { if ($i \equiv n_0$) then $\mathtt{IN}_i$ = $Outside\_Info$
      else $\mathtt{IN}_i = \top$
      if ($i \equiv n_\infty$) then $\mathtt{OUT}_i$ = $Outside\_Info$
      else $\mathtt{OUT}_i = \top$
    }
    change = true
    while (change)     /* Keep Recomputing */
    {     change = false
          for each node $i$ /*  Visited in a fixed order */
          {    Compute $\mathtt{IN}_i$ and $\mathtt{OUT}_i$
               if either $\mathtt{OUT}_i$ or $\mathtt{IN}_i$ changes then
                    change = true
          }
    }
}
```

FIGURE 1.19 Round robin iterative method of data flow analysis.

1.6.1.1 Round Robin Iterative Data Flow Analysis

This is the simplest method of performing data flow analysis. Figure 1.19 provides the algorithm that is self-explanatory. It is applicable to the unidirectional and bidirectional data flow problems uniformly. The reason is simple: the data flow Equations (1.40) and (1.41), which are used in this algorithm, allow both forward as well as backward flow functions and distinguish between the node and the edge flow functions. This enables traversing all *ifp*'s; they get traversed in parts and common segments are shared across *ifp*'s.

We leave it to the reader to verify that this method indeed computes the maximum fixed point solution (1.16) for our example program flow graph in Figure 1.2. Note that a fixed order of traversal is not really necessary for convergence on a fixed point. Because available expressions analysis has forward data flows, a forward direction of traversal can converge faster. However, a backward traversal can also converge on the same fixed point, although it would take much longer. We postpone the issue of the precise meaning of forward and backward traversals, the choice of the order of traversal, the number of traversals required, etc. to Section 1.6.3 where we discuss the complexity of iterative data flow analysis.

Interestingly, if we use the initialization $\perp$ instead of $\top$, the resulting solution is the minimum fixed point solution (1.15). The reason should be evident from Figure 1.17.

1.6.1.2 Work List Based Iterative Data Flow Analysis

The work list based method eliminates the drawbacks of the round robin method by computing data flow properties in a demand-driven manner. We evolve the algorithm in the following steps:

- *Avoiding redundant recomputations*. If the data flow property associated with a program point changes in any iteration, the round robin iterative method recomputes the data flow properties of all nodes indiscriminately because it does not know exactly which data flow properties have changed in an iteration. A single scalar variable "change" records the fact that some data flow properties have changed somewhere.

A natural improvement over this algorithm would be to remember the program points where data flow properties change. Then it would be possible to recompute only those data flow properties that may be affected by a change. This gives rise to the *work list based iterative method* of data flow analysis. The basic idea is simple:

> Whenever the data flow property at a program point changes, the program point is added to a work list. Further recomputations are governed by the program points that are included in the work list. A program point is removed from the work list, and the data flow properties of neighboring program points are recomputed. If a change occurs, the corresponding program points are added to the work list and the process is repeated until the work list becomes empty.

- *Refinement of data flow properties.* If the data flow values are initialized to $\top$ (or any value larger than the expected MSA), then the values can only decrease, and can never increase. To appreciate the full implication of this, consider node i such that:

$$\mathtt{IN}_i = \bigsqcap_{p \in pred(i)} g^E_{p\to i}(\mathtt{OUT}_p) \sqcap f^B_i(\mathtt{OUT}_i) \tag{1.59}$$

If only $\mathtt{OUT}_i$ changes (and not $\mathtt{OUT}$ of predecessors), is it necessary to apply $g^E_{p\to i}$ and merge the resulting (unchanged) data flow information to compute $\mathtt{IN}_i$? To answer this, let the new values be indicated by $\mathtt{IN}_i^{New}$ and $\mathtt{OUT}_i^{New}$. Then after the recomputation:

$$\mathtt{IN}_i^{New} = \bigsqcap_{p \in pred(i)} g^F_{p\to i}(\mathtt{OUT}_p) \sqcap f^B_i(\mathtt{OUT}_i^{New}) \tag{1.60}$$

Because a change is only a (potential) decrease instead of an increase:

$$\begin{aligned}
\mathtt{OUT}_i^{New} &\sqsubseteq \mathtt{OUT}_i \\
f_i^B(\mathtt{OUT}_i^{New}) &\sqsubseteq f_i^B(\mathtt{OUT}_i) && (\ldots \text{ assuming monotonicity}) \\
f_i^B(\mathtt{OUT}_i^{New}) &= f_i^B(\mathtt{OUT}_i) \sqcap f_i^B(\mathtt{OUT}_i^{New}) && (\ldots \text{ by the definition of } \sqsubseteq)
\end{aligned}$$

Substituting the preceding in Equation (1.60) results in:

$$\begin{aligned}
\mathtt{IN}_i^{New} &= \bigsqcap_{p \in pred(i)} g^E_{p\to i}(\mathtt{OUT}_p) \sqcap f^B_i(\mathtt{OUT}_i) \sqcap f^B_i(\mathtt{OUT}_i^{New}) \\
\mathtt{IN}_i^{New} &= \left(\bigsqcap_{p \in pred(i)} g^F_{p\to i}(\mathtt{OUT}_p) \sqcap f^B_i(\mathtt{OUT}_i) \right) \sqcap f^B_i(\mathtt{OUT}_i^{New}) && (\ldots \text{by associativity}) \\
\mathtt{IN}_i^{New} &= \mathtt{IN}_i \sqcap f^B_i(\mathtt{OUT}_i^{New}) && (\ldots \text{ from (1.59)})
\end{aligned}$$

> What we have discovered is the powerful mechanism of *refinement* of data flow properties. Whenever a data flow property changes, there is no need to recompute the data flow properties that it affects; these data flow properties can be just refined by merging their old values with the value of the changed data flow property that affects them.

- *Initializing the work list.* Initializing the work list requires identifying the program points from where the process of refinement should begin. A natural choice is the program points where the values of data flow properties are smaller than $\top$ due to the local effect of the node.

In the context of available expressions analysis this means: assume all the expressions to be available at $\texttt{entry}_i$ and find out those that are not available at $\texttt{exit}_i$. If $\texttt{AVOUT}_i$ turns out to be smaller than $\top$, then we have found some such expressions and $\texttt{exit}_i$ should be included in the initial work list so that we can mark all these expressions as not available at $\texttt{AVIN}_s$ of successors of i. Thus, we should use $\top$ values in all the flow functions to identify the program points to be included in the work list:

$$\texttt{AVOUT}_i = f_i^F(\texttt{AVIN}_i) = f_i^F(\top) = \texttt{AvGen}_i \cup (\top - \texttt{AvKill}_i) \tag{1.61}$$

This is the same as the computation performed in the first iteration of round robin data flow analysis (after the initialization).

- *Algorithm*. The resulting algorithm is provided in Figure 1.20. Note that for unidirectional data flow problems, some flow functions are constant $\top$ functions. They do not contribute to the data flow information and hence can be removed. In particular:
  - For forward data flow problems the computations on line numbers 26 to 30, 33 to 35 and the function application f_i^B on line numbers 7 and 8, and $g_{i\to s}^B$ on line number 11 are redundant.
  - For the backward data flow problems the computations on line numbers 23 to 25, 36 to 40 and the functions application f_i^E on line numbers 10 and 11, and $g_{p\to i}^F$ on line number 8 are redundant.

We perform available expressions analysis for the program flow graph in Figure 1.2. Definition (1.8) provides its flow functions. In this context, $\top$ is the universal set $\mathbf{U} = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8\}$ and *Outside_Info* is assumed to be $\emptyset$. The values of the data flow properties with $\top$ as the argument of the flow functions are:

Node	Avin	Avout
1	$\emptyset$	$\{e_1, e_3, e_4\} \subset \top$
2	$\top$	$\{e_1, e_2, e_3, e_5, e_6, e_7, e_8\} \subset \top$
3	$\top$	$\{e_1, e_2, e_3, e_4, e_7, e_8\} \subset \top$
4	$\top$	$\{e_1, e_2, e_3, e_4, e_7, e_8\} \subset \top$
5	$\top$	$\{e_1, e_3, e_4, e_5, e_6, e_8\} \subset \top$
6	$\top$	$\top$

The work list is initialized to $\langle\texttt{entry}_1, \texttt{exit}_1, \texttt{exit}_2, \texttt{exit}_3, \texttt{exit}_4, \texttt{exit}_5\rangle$. Inserting the program points in the direction of the data flows would lead to a faster convergence than inserting it either at the end or in the beginning indiscriminately. In the context of available expressions analysis we insert the program points in forward direction (to be made more precise in Section 1.6.3) because of forward data flows. As far as correctness is concerned, it does not matter where a program point is inserted in the work list.

Step No.	Program Point Selected	Data Flow Properties Refined	Resulting Work List
1	$\texttt{entry}_1$	$\texttt{AVOUT}_i = \{e_1\}$	$\langle \texttt{exit}_1, \texttt{exit}_2, \texttt{exit}_3, \texttt{exit}_4, \texttt{exit}_5 \rangle$
2	$\texttt{exit}_1$	$\texttt{AVIN}_2 = \texttt{AVIN}_6 = \{e_1\}$	$\langle \texttt{entry}_2, \texttt{exit}_2, \texttt{exit}_3, \texttt{exit}_4, \texttt{exit}_5, \texttt{entry}_6 \rangle$
3	$\texttt{entry}_2$	$\texttt{AVOUT}_2 = \{e_1, e_2, e_3\}$	$\langle \texttt{exit}_2, \texttt{exit}_3, \texttt{exit}_4, \texttt{exit}_5, \texttt{entry}_6 \rangle$
4	$\texttt{exit}_2$	$\texttt{AVIN}_3 = \texttt{AVIN}_4 = \{e_1, e_2, e_3\}$	$\langle \texttt{entry}_3, \texttt{exit}_3, \texttt{entry}_4, \texttt{exit}_4, \texttt{exit}_5, \texttt{entry}_6 \rangle$
5	$\texttt{entry}_3$	$\texttt{AVOUT}_3 = \{e_1, e_2, e_3, e_4\}$	$\langle \texttt{exit}_3, \texttt{entry}_4, \texttt{exit}_4, \texttt{exit}_5, \texttt{entry}_6 \rangle$
6	$\texttt{exit}_3$	$\texttt{AVIN}_5 = \{e_1, e_2, e_3, e_4\}$	$\langle \texttt{entry}_4, \texttt{exit}_4, \texttt{entry}_5, \texttt{exit}_5, \texttt{entry}_6 \rangle$
7	$\texttt{entry}_4$	$\texttt{AVOUT}_4 = \{e_1, e_2, e_3, e_4\}$	$\langle \texttt{exit}_4, \texttt{exit}_5, \texttt{entry}_6 \rangle$
8	$\texttt{exit}_4$	No change in $\texttt{AVIN}_5$	$\langle \texttt{entry}_5, \texttt{exit}_5, \texttt{entry}_6 \rangle$
9	$\texttt{entry}_5$	$\texttt{AVOUT}_5 = \{e_1, e_3, e_4, e_8\}$	$\langle \texttt{exit}_5, \texttt{entry}_6 \rangle$
10	$\texttt{exit}_5$	No change in $\texttt{AVIN}_2$ and $\texttt{AVIN}_6$	$\langle \texttt{entry}_6 \rangle$
11	$\texttt{entry}_6$	$\texttt{AVOUT}_6 = \{e_1\}$	$\langle \texttt{exit}_6 \rangle$
12	$\texttt{exit}_6$	Does not affect any property	$\langle\rangle$

It is easy to see that the resulting solution is indeed the same as assignment (1.16). It would be interesting to use $\bot$ (i.e., $\emptyset$) to compute the data flow properties and compare them with $\top$ (and not $\bot$) to initialize the work list; this would compute the minimum fixed point solution (1.15).

1.6.2 Elimination Methods of Data Flow Analysis

Elimination methods try to reduce the amount of effort required to solve a data flow problem by utilizing the structural properties of a flow graph. This approach basically consists of the following two steps:

- Identify regions with some desirable properties in the flow graph.
  - Construct the flow functions for each region by composing and merging flow functions along the *ifp*'s in the region.
  - Reduce each region to a single node.

  If successive applications of graph transformations reduce the graph to a single node, the graph is said to be *reducible*.
- Compute the data flow properties of the nodes (representing regions) in the derived graphs using the flow functions constructed in the first step. This process starts with the last derived graph (i.e., single node graph) and terminates with the original graph. The data flow properties of a region are used to compute the data flow properties of the constituent nodes that represent regions in the lower level derived graph. This enables delayed substitution of some values in the simultaneous equations.

1.6.2.1 Identifying Regions

Section 1.3.7 provided a glimpse of this approach through Figure 1.6. The complete sequence of reductions and the derived graphs is shown in Figure 1.21 that follows. We use *interval analysis* that tries to identify the maximal single entry regions in a graph.[20]

[20]Interval analysis falls in the category of control flow analysis, which we do not discuss here. Section 1.7 provides references and suggestions for further reading.

```
procedure worklist_dfa()
{   init ()
    settle ()
}
procedure init()
{       for each node $i$
        {       if ($i \equiv n_0$) then $\mathsf{IN}_i = Outside\_Info \sqcap f_i^B(\top)$
                else $\mathsf{IN}_i = \prod_{p \in pred(i)} g^F_{p\to i}(\top) \sqcap f_i^B(\top)$
                if ($\mathsf{IN}_i \sqsubset \top$) then Insert entry$_i$ in LIST
                if ($i \equiv n_\infty$) then $\mathsf{OUT}_i = Outside\_Info \sqcap f_i^F(\top)$
                else $\mathsf{OUT}_i = \prod_{s \in succ(i)} g^B_{i\to s}(\top) \sqcap f_i^F(\top)$
                if ($\mathsf{OUT}_i \sqsubset \top$) then Insert exit$_i$ in LIST
        }
}
procedure settle()
{   while the LIST is not empty
    {   Remove the first program point from LIST
        if the program point is entry$_i$ for some node$_i$ then propagate_in($i$)
        else propagate_out($i$)    /* must be exit$_i$ for some node $i$ */
    }
}
procedure propagate_in($i$)
{ $\mathtt{OUT}_i^{New} = \mathtt{OUT}_i \sqcap f_i^F(\mathtt{IN}_i)$      /* refinement using $f_i^F$ */
    if ($\mathtt{OUT}_i^{New} \sqsubset \mathtt{OUT}_i$)then
        Update $\mathtt{OUT}_i = \mathtt{OUT}_i^{New}$ and Insert exit$_i$ in LIST
    for all $p \in pred(i)$
    {   $\mathtt{OUT}_i^{New} = \mathtt{OUT}_p \sqcap g^B_{p\to i}(\mathtt{IN}_i)$      /* refinement usig $g^B_{p\to i}$ */
        if ($\mathtt{OUT}_p^{New} \sqsubset \mathtt{OUT}_p$)then
            Update $\mathtt{OUT}_p = \mathtt{OUT}_p^{New}$ and Insert exit$_p$ in LIST
    }
}
procedure propagate_out($i$)
{   $\mathtt{IN}_i^{New} = \mathtt{IN}_i \sqcap f_i^B(\mathtt{OUT}_i)$      /* refinement usig $f_i^B$ */
    if ($\mathtt{IN}_i^{New} \sqsubset \mathtt{IN}_i$)then
        Update $\mathtt{IN}_i = \mathtt{IN}_i^{New}$ and Insert entry$_i$ in LIST
    for all $s \in succ(i)$
    {   $\mathtt{IN}_s^{New} = \mathtt{IN}_s \sqcap g^F_{i\to s}(\mathtt{OUT}_i)$      /* refinement usig $g^F_{i\to s}$ */
        if ($\mathtt{IN}_s^{New} \sqsubset \mathtt{IN}_s$)then
            Update $\mathtt{IN}_s = \mathtt{IN}_s^{New}$  and Insert entry$_s$ in LIST
    }
}
```

FIGURE 1.20 Work list based iterative method of data flow analysis. (From Khedker, U.P. and Dhamdhere, D.M., *ACM TOPLAS*, 16(5), 1472–1511, 1994. Reproduced with permission of ACM.)

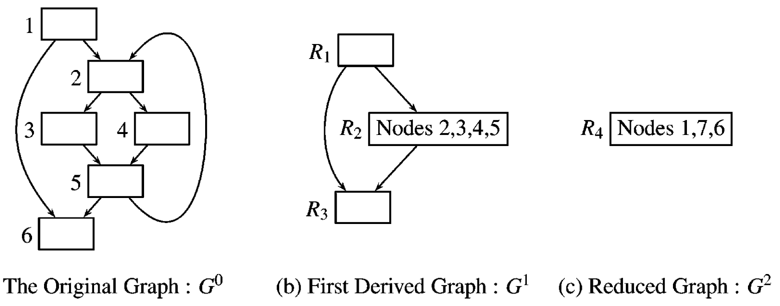

1
2
3
4
5
6
R_1
R_2
Nodes 2,3,4,5
R_3
R_4
Nodes 1,7,6
(a) The Original Graph : G^0
(b) First Derived Graph : G^1
(c) Reduced Graph : G^2

FIGURE 1.21 Reducing graphs for elimination method of data flow analysis.

G^1 is the first derived graph constructed from the original graph G^0. Region R_1 and R_3 consist of single nodes (1 and 6, respectively) whereas region R_2 consists of nodes 2, 3, 4 and 5. Regions R_1, R_2 and R_3 are combined to form the lone region R_4 representing the *limit graph* G^2.

1.6.2.2 Constructing the Flow Functions

Because available expressions analysis is a bit vector framework, its functions are of the form $h(X) = \texttt{Gen} \cup (X - \texttt{Kill})$. Also, because it is two bounded, the loop closure of any function h is $h^* = \iota \cap h$ where ι is the identity function. Function compositions and loop closures are defined as follows:

- *Function composition.* We write $h(X) = \texttt{Gen} \cup (X \cap \neg\texttt{Kill})$ for ease of algebraic manipulation. Let $h_1(X) = \texttt{Gen}_1 \cup X \cap \neg\texttt{Kill}_1$ and $h_2(X) = \texttt{Gen}_2 \cup X \cap \neg\texttt{Kill}_2$. Let $h'(X) = h_2(h_1(X))$.

$$\begin{aligned} h'(X) = \texttt{Gen}' \cup (X - \texttt{Kill}') &= \texttt{Gen}_2 \cup (\texttt{Gen}_1 \cup X \cap \neg\texttt{Kill}_1) \cap \neg\texttt{Kill}_2 \\ &= \texttt{Gen}_2 \cup (\texttt{Gen}_1 \cap \neg\texttt{Kill}_2 \cup X \cap \neg\texttt{Kill}_1 \cap \neg\texttt{Kill}_2) \\ &= \texttt{Gen}_2 \cup \texttt{Gen}_1 \cap \neg\texttt{Kill}_2 \cup X \cap \neg(\texttt{Kill}_1 \cup \texttt{Kill}_2) \\ &= \texttt{Gen}_2 \cup (\texttt{Gen}_1 - \texttt{Kill}_2) \cup (X - (\texttt{Kill}_1 \cup \texttt{Kill}_2)) \end{aligned}$$

Thus for composition, $\texttt{Gen}' = \texttt{Gen}_2 \cup (\texttt{Gen}_1 - \texttt{Kill}_2)$ and $\texttt{Kill}' = \texttt{Kill}_1 \cup \texttt{Kill}_2$.
- *Loop closure.* Loop closure is worked out in a similar way. For this example, we need a simpler function intersection $\iota \cap h$ for loop closure h^*:

$$\begin{aligned} h^*(X) &= \iota(X) \cap h(X) = X \cap (\texttt{Gen} \cup (X - \texttt{Kill})) \\ &= X \cap \texttt{Gen} \cup (X \cap (X - \texttt{Kill})) \\ &= X \cap \texttt{Gen} \cup (X - \texttt{Kill}) \qquad (\ldots (X - \texttt{Kill}) \subseteq X) \\ &= X - (\texttt{Kill} - \texttt{Gen}) \end{aligned}$$

Thus for loop closure, $\texttt{Gen}' = \emptyset$ and $\texttt{Kill}' = \texttt{Kill} - \texttt{Gen}$. This implies that no expression can be generated within the loop that can be available at the loop entry along the looping edge (i.e., the looping edge cannot add any expression to the set of expressions available at the loop entry; it can only remove some expressions if they are not available at the loop exit).

Now we construct the flow functions for each region. All the edge flow functions are identity functions. Because all the node flow functions are forward functions, we drop the superscript F.

- $f_{R_1} = f_1$.
- $f_{R_2} = f_5 \circ (f_3 \circ f_2 \cap f_4 \circ f_2)^* = f_5 \circ f_3 \circ f_2$ because f_3 and f_4 are identical from (1.8). We first compose f_3 and f_2 to construct f_{23} (i.e., compute $\texttt{AvGen}_{23}$ and $\texttt{AvKill}_{23}$) using the preceding definition of function composition:

$$\begin{aligned}
\texttt{AvGen}_{23} &= \texttt{AvGen}_3 \cup (\texttt{AvGen}_2 - \texttt{AvKill}_3)\\
&= \{e_4\} \cup (\{e_2, e_3\} - \{e_5, e_6\})\\
&= \{e_2, e_3, e_4\}\\
\texttt{AvKill}_{23} &= \texttt{AvKill}_3 \cup \texttt{AvKill}_2\\
&= \{e_3, e_4\} \cup \{e_5, e_6\}\\
&= \{e_3, e_4, e_5, e_6\}
\end{aligned}$$

  Then we compose $f_5 \circ f_{23}$ to construct f_{R_2}:

$$\begin{aligned}
\texttt{AvGen}_{R_2} &= \texttt{AvGen}_5 \cup (\texttt{AvGen}_{23} - \texttt{AvKill}_5)\\
&= \{e_8\} \cup (\{e_2, e_3, e_4\} - \{e_2, e_7, e_8\})\\
&= \{e_3, e_4, e_8\}\\
\texttt{AvKill}_{R_2} &= \texttt{AvKill}_5 \cup \texttt{AvKill}_{23}\\
&= \{e_2, e_7, e_8\} \cup \{e_3, e_4, e_5, e_6\}\\
&= \{e_2, e_3, e_4, e_5, e_6, e_7, e_8\}
\end{aligned}$$

  Now we construct the loop closure:

$$\begin{aligned}
f^*_{R_2}(X) &= X - (\texttt{AvKill}_{R_2} - \texttt{AvGen}_{R_2})\\
&= X - (\{e_2, e_3, e_4, e_5, e_6, e_7, e_8\} - \{e_3, e_4, e_8\})\\
&= X - \{e_2, e_5, e_6, e_7\}
\end{aligned}$$

- $f_{R_3} = f_6$.
- $f_{R_4} = f_{R_3} \circ (f_{R_1} \cap f_{R_2} \circ f_{R_1})$. Because our motivation basically is to compute $\texttt{AVIN}_i$ (from which $\texttt{AVOUT}_i$ can be computed by applying f_i), constructing the flow function R_4 (i.e., for the region representing the limit graph) is not required.

1.6.2.3 Computing Data Flow Properties

After computing the flow functions, now we are in a position to compute the data flow properties. We begin with the limit graph G^2:

- *For limit graph* G^2: $\texttt{AVIN}_{R_4} = \textit{Outside\_Info} = \emptyset$.
- *For graph* G^1:

*We assume that function composition has higher precedence than function intersection.

$$\begin{aligned}
\texttt{AVIN}_{R_1} &= \texttt{AVIN}_{R_4} = \emptyset && (\ldots\ \text{Node}R_1 \text{ is the header of region } R_4)\\
\texttt{AVIN}_{R_2} &= f^*_{R_2}(f_{R_1}(\texttt{AVIN}_{R_1})) && (\ldots\ \text{Loop closure must be taken for } R_2)\\
&= f^*_{R_2}(f_{R_1}(\emptyset)) = f^*_{R_2}(f_1(\emptyset))\\
&= f^*_{R_2}(\{e_1\} \cup (\emptyset - \{e_2, e_5, e_6, e_7, e_8\})) && (\ldots \text{Substituting for } f_1 \text{ from (1.8)})\\
&= f^*_{R_2}(\{e_1\}) = \{e_1\} - \{e_2, e_5, e_6, e_7\}\\
&= \{e_1\}
\end{aligned}$$

$$\begin{aligned}
\texttt{AVIN}_{R_3} &= f_{R_1}(\texttt{AVIN}_{R_1}) \cap f_{R_2}(\texttt{AVIN}_{R_2})\\
&= f_1(\emptyset) \cap f_{R_2}(\{e_1\})\\
&= (\{e_1\} \cup (\emptyset - \{e_2, e_5, e_6, e_7, e_8\}))\\
&\quad \cap(\{e_3, e_4, e_8\} \cup (\{e_1\} - \{e_2, e_3, e_4, e_5, e_6, e_7, e_8\}))\\
&= \{e_1\} \cap (\{e_1, e_3, e_4, e_8\})\\
&= \{e_1\}
\end{aligned}$$

- *For Graph G^0*:

$$\begin{aligned}
\texttt{AVIN}_1 &= \texttt{AVIN}_{R_1} = \emptyset && (\ldots \text{Node 1 is the header of region } R_1)\\
\texttt{AVIN}_2 &= \texttt{AVIN}_{R_2} = \{e_1\} && (\ldots \text{Node 2 is the header of region } R_2)\\
\texttt{AVIN}_3 &= f_2(\texttt{AVIN}_2) = f_2(\{e_1\})\\
&= \{e_2, e_3\} \cup (\{e_1\} - \{e_3, e_4\}) && (\ldots \text{Substituting for } f_2 \text{ from (1.8)})\\
&= \{e_1, e_2, e_3\}\\
\texttt{AVIN}_4 &= f_2(\texttt{AVIN}_2) = f_2(\{e_1\})\\
&= \{e_2, e_3\} \cup (\{e_1\} - \{e_3, e_4\})\\
&= \{e_1, e_2, e_3\}
\end{aligned}$$

$$\begin{aligned}
\texttt{AVIN}_5 &= f_3(\texttt{AVIN}_3) \cap f_4(\texttt{AVIN}_4)\\
&= f_3(\{e_1, e_2, e_3\}) \cap f_4(\{e_1, e_2, e_3\})\\
&= (\{e_4\} \cup (\{e_1, e_2, e_3\} - \{e_5, e_6\})) && (\ldots \text{Substituting for} f_3 \text{ and } f_4 \text{ from (1.8)})\\
&\quad \cap(\{e_4\} \cup (\{e_1, e_2, e_3\} - \{e_5, e_6\}))\\
&= \{e_1, e_2, e_3, e_4\}\\
\texttt{AVIN}_6 &= \texttt{AVIN}_{R_3} = \{e_1\} && (\ldots \text{Node 6 is the header of region } R_3)
\end{aligned}$$

From $\texttt{AVIN}_i$, we can compute $\texttt{AVOUT}_i$ by applying f_i. It is easy to see that the resulting solution is the same as the maximum fixed point assignment (1.16).

1.6.3 Complexity of Data Flow Analysis

In this section we discuss the complexity of the various methods of data flow analysis.

1.6.3.1 Round Robin Iterative Methods of Data Flow Analysis

It was suggested in Section 1.6.1.1 that the round robin method would converge faster if the nodes are traversed in the direction of the data flow. This section explains the order of traversal more precisely.

The entire discussion in this section is with respect to a depth-first spanning tree of a control flow graph. A spanning tree is a tree that spans (i.e., covers) all nodes in a graph whereas depth-first traversal visits the (unvisited) successors of a node before visiting its siblings (i.e., successors of a predecessor). Figure 1.22 provides a depth-first spanning tree (consisting of the solid arrows) for the control flow graph in Figure 1.2.

We define the following terms for a graph G with respect to a chosen depth-first spanning tree $T(G)$ of G. An edge $i \to k$ is a tree edge in G if k is a descendant of i in $T(G)$; it is a back edge if k is an ancestor of i in $T(G)$. As a special case, an edge from a node to itself is also considered a

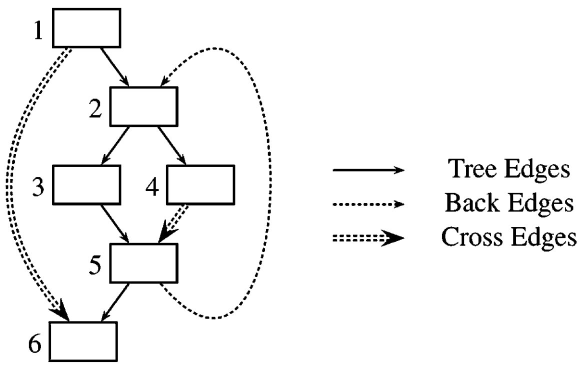


FIGURE 1.22 Classifications of edges in a graph based on a depth-first spanning tree.

back edge. An edge $i \rightarrow k$ is a cross edge in G if k is neither an ancestor, nor a descendant of i in in $T(G)$. Tree edges and cross edges are clubbed together as forward edges. A forward, or reverse postorder, traversal of a graph implies traversing G such that a node is traversed before any of its descendants in $T(G)$. For our graph in Figure 1.22, two possible forward traversals are 1, 2, 3, 4, 5, 6 and 1, 2, 4, 3, 5, 6. A backward, or postorder, traversal of a graph implies traversing G such that a node is traversed after all its descendants in $T(G)$. For our graph in Figure 1.22, two possible backward traversals are 6, 5, 3, 4, 2, 1 and 6, 5, 4, 3, 2, 1.

To estimate the total number of traversals required in the case of a round robin iterative method, consider an *ifp* (`exit`$_2$, `entry`$_3$, `exit`$_3$, `entry`$_5$, `exit`$_5$, `entry`$_2$). We assume that visiting node i leaves the values of $\texttt{IN}_i$ and $\texttt{OUT}_i$ in a mutually consistent state. This *ifp* is covered by the sequence of edge traversals ($2 \rightarrow 3 \rightarrow 5 \rightarrow 2$). Let us choose forward traversal over the graph. Let the edge traversals that are along the direction of the graph traversal be indicated by $\surd$ and the edge traversals against the direction of the graph traversal be indicated by $\times$. Then, our *ifp* can be represented by ($2 \overset{\surd}{\rightarrow} 3 \overset{\surd}{\rightarrow} 5 \overset{\times}{\rightarrow} 2$). The edge traversals marked by $\surd$ are called *conforming* traversals whereas the ones marked by $\times$ are called *nonconforming* traversals. A nonconforming edge traversal implies that its target node is visited before the source node in the chosen graph traversal; in the case of ($5 \overset{\times}{\rightarrow} 2$), node 2 has already been visited before node 5 in a forward traversal over the graph. Clearly, to propagate the data flow information from `exit`$_5$ to `entry`$_2$, another traversal over the graph is required. This can be generalized as follows:

Every nonconforming edge traversal requires an additional iteration in round robin iterative data flow analysis.

We define the *width of a graph* for a data flow framework as the maximum number of nonconfirming edge traversals in any *ifp*.[21] If the width of a graph for a data flow framework is w, then it is easy to show that the round robin iterative method of data flow analysis can converge in $w + 1$ iterations for two-bounded problems. If the data flow framework is k bounded, then the bound on the number of iterations becomes $(k - 1) \cdot w + 1$.

It is interesting to note that if we choose a backward traversal over the graph, then node 5 would be visited before node 2 and the edge traversal ($5 \rightarrow 2$) would not need an additional iteration; consequently it becomes a conforming traversal. On the other hand, the edge traversal ($2 \rightarrow 3$), which is a conforming traversal with respect to forward traversal over the graph, becomes a nonconfirming

[21]In the formal statement, a qualifying clause says, "no part of which has been bypassed."

traversal with respect to the backward traversal over the graph because node 3 is visited before 2; this requires an additional iteration to visit 3 after 2. In the general situation, a nonconforming edge traversal may be associated with a forward edge or a back edge. We show examples of all possibilities in the following:

			Type of Edge Traversal	
Direction of Edge	Direction of Data Flow	An Example Edge Traversal	Forward Graph Traversal	Backward Graph Traversal
Forward	Forward	$(2 \to 3)$	√	×
Forward	Backward	$(3 \to 2)$	×	√
Back	Forward	$(5 \to 2)$	×	√
Back	Backward	$(2 \to 5)$	√	×

We observe that:

- For forward data flow problems, the edge traversals of the kind $(3 \to 2)$ and $(2 \to 5)$ do not exist because both of them are backward flows. If we choose the forward graph traversal, then the nonconfirming edge traversals are only those that are traversals along a back edge. On the other hand, if we choose the backward graph traversal, then the nonconfirming edge traversals are those that are associated with forward edges. Because the number of forward edges usually is larger than the number of back edges, the width is very high in the case of backward graph traversal for problems involving forward data flows. This explains, quantitatively, why the number of traversal is much larger for available expressions analysis in the case of backward graph traversal.
- For backward data flow problems, the edge traversals of the kind $(2 \to 3)$ and $(5 \to 2)$ do not exist because both of them are forward flows. Thus, the nonconfirming edge traversals with respect to the backward graph traversal are only those that are traversals along a back edge. Clearly, a forward graph traversal can have many more nonconforming edge traversals requiring a much larger number of iterations.

Thus for unidirectional data flow problems, if the direction of graph traversal is the same as the direction of the data flows, the width of a graph reduces to the *depth of a graph*, which is defined as the maximum number of back edges in an acyclic path in a graph. However, depth is applicable to unidirectional data flow problems only. Width is not only more general than depth but also more precise than depth because it considers only those *ifp* segments that are not short-circuited by shorter *ifp* segments.

Though the round robin method converges in $w + 1$ iterations, in practice we do not know the width of a flow graph and the method terminates after discovering that there is no further change. Thus, practically, $w + 2$ iterations are required; if n nodes and r data items exist (i.e., r data flow properties), the total work is $O((w + 2) \cdot n \cdot r)$.

1.6.3.2 Work List Based Iterative Method of Data Flow Analysis

Work list based methods work in a demand-driven manner, and computations are made only where a change is likely to happen. In the worst case, a data flow property may have to be computed everywhere in a graph. For the bit vector problems, the effective height of the lattice is one and functions are monotonic, implying that a value can change only once. If the entire graph is traversed for this data flow property it implies $O(e)$ computations where e is the number of edges. If r properties exist, in the worst case the entire graph may have to be visited for each one of them, implying the worst-case complexity of $O(e \cdot r)$; because $e = O(n)$ for practical control flow graphs, the bound is $O(n \cdot r)$. If the framework is k bounded, a multiplication factor is $(k - 1)$.

1.6.3.3 Elimination Methods of Data Flow Analysis

Let there be r data flow properties and let the total number of nodes in all the graphs in the sequence of derived graphs (including the original graph) be $\mathscr{N}$. Because r data flow properties are computed for each node once, the complexity of elimination methods is $O(\mathscr{N} \cdot r)$.

1.6.4 Comparison of Solution Methods and Some Practical Issues

Elimination methods require complicated control flow analysis[22] and are restricted to programs that are reducible. They work well for bit vector data flow frameworks but are difficult to adapt to different data flow problems and it is difficult to prove their correctness. They require different treatment for forward and backward data flow problems and are not applicable to general bidirectional data flow problems. They usually work on identifying loops to avoid repetitive computations of data flow properties. However, for bidirectional data flow problems, even forward edges can have a behavior similar to back edges (i.e., they could be nonconforming; see Section 1.6.3.1) and hence may require repetitive computations of data flow properties. Apart from the suitability of a structure to be considered a region, the requirement of the ability to construct flow functions for a given instance of a data flow framework puts some limitations on the adaptability of elimination-based methods. Function compositions and merging may not be possible for general flow functions, say for flow function such as (1.27) for constant propagation. Hence, elimination methods are usually not a part of contemporary practices.

If an optimizing compiler uses many data flow analyses, it is preferable to use iterative methods. The round robin method is quite simple to implement and is reasonably fast; practically the widths of common control flow graphs (which admittedly depend on the data flow frameworks also) are quite small (at most two or three for unidirectional data flows and may be four for data flows like MR-PRE).

For the problems that have high widths, it is preferable to use the work list based method because it processes only those data flow properties that need to be processed. However, its complexity does not take into account the overheads of list management. Our initial experiments indicated that though the number of bit vector operations were much less for work list methods, the round robin methods worked faster; the list management overheads had more than nullified the advantage of fewer bit vector operations. We discovered that the most practical list management is with the help of a bit string where each bit corresponds to a program point. Constant time fast operations for this list can be implemented. Besides, the list can be easily organized (into a fixed order) to take advantage of the direction of data flows. With this organization, our implementations of work list methods turned out to be faster than round robin methods.

For bit vector problems, m bits can be processed in one machine operation where the size of the machine word is m. Thus, actual cost of wordwise iterative analysis reduces considerably. This is applicable to both round robin as well as work list methods of data flow analysis.

As a final comment, the global data flow analysis depends on local data flow analysis for computing the local data flow properties (namely, `Gen` and `Kill`). In our experience, this is harder than getting the global data flow analyzer to work. The reason is simple: the global data flow analyzer assumes simple interface with the program in the form of a control flow graph, and the data flow semantics of the actual computations performed in the program are hidden behind the local data flow properties.

[22]The exception is if they are performed on abstract syntax trees, in which case data flow analysis becomes quite complicated.

Practical intermediate representations have to address the concerns of all the phases of a compiler, right from syntax analysis to code generation; thus, they are not simple and clean. The local data flow analyzer has to deal with the entire gamut of information included in the intermediate representation and as such warrants extra efforts.

1.7 Bibliographic Notes

Most of texts on compilers discuss data flow analysis in varying lengths [1, 3, 23, 38, 39, 58]. The ones that discuss details are [1, 3, 39]. Among other books, a more advanced treatment can be found in [24, 40, 41].

Historically, the practice of data flow analysis precedes the theory. The round robin data flow analysis can be traced back to [57]. The pioneering work on data flow analysis includes [2, 22, 25, 30]. The classical concepts in data flow analysis are based on the treatments in [1, 22, 24, 25, 30, 35, 45]. All of them were restricted to unidirectional data flows. The need for a more general theory was perceived after the advent of the partial redundancy elimination [37] and composite hoisting and strength reduction [15] optimizations. This was felt particularly due to the fact that the complexity measures of unidirectional analyses were not applicable to these analyses. The initial forays [12, 17, 18, 31] failed to address the concerns adequately for two reasons: they were mostly directed at solution methods or were restricted to MR-PRE class of problems. They were based on edge splitting and placement that became very popular and served the cause of many investigations. Meanwhile, a generalized theory for data flows was proposed [16, 27, 28]. The theoretical foundations of these works provided interesting insights into the process of data flow analysis and paved the way for simple generic algorithms and precise complexity bounds that are uniformly applicable to unidirectional and bidirectional data flows — most discussions in this chapter are based on these insights.

We believe that it is very important to distinguish between the theoretical foundations of data flow analysis (the what) and the solution methods of data flow analysis (the how). The initial distinctions were made by [22, 25, 30]; [22] used set theoretical foundations whereas [25, 30] pioneered the lattice theoretical foundations that have now become de facto standard. These distinctions were strengthened by [16, 28, 35, 45]. The third aspect of data flow analysis (i.e., the semantics of the information captured by data flow analysis) is covered very well by [41].

Monotonicity in data flow analysis was first observed in [25] and almost simultaneously and independently in [22]. The implication of monotonicity captured by [46] is also called GKUW property, which is basically named after the authors of [22, 25]. Separability of function spaces has been used only implicitly in the literature; it was first defined in [28]. Similarly, the bit vector frameworks were defined only intuitively, with the first formal definition provided by [28]. The example of combined may-availability and must-availability is from [5], where it is simply called availability analysis.

Elimination methods are also called *structural methods* because they use the structural properties of control flow graphs. The pioneering works in elimination methods of data flow analysis are [2, 22, 51, 55]. A good discussion of these methods can be found in [24, 48]; later notable works include [6, 53]. We have ignored some solution methods, namely, node listing based data flow analysis; a much wider range of solution methods can be found in [24, 26].

Finally, some aspects of data flow analysis that we have left out for want of space. Each one of them deserves a separate treatment in an independent chapter. We have not covered interprocedural data flow analysis. Two pioneering publications on interprocedural data flow analysis are [4, 52]. Other popular investigations include [6, 9, 10, 19, 21, 43]. A good discussion can be found in [39, 41]. The effectiveness of interprocedural data flow analysis was studied in [44] and more recently in [36]. We have also excluded the effect of aliasing, which is an important issue related to both interprocedural as well as intraprocedural data flow analysis. A partial list of some recent works along this direction includes [11, 32, 33, 46, 50, 59]. Further, we have not discussed incremental data flow analysis, which updates the previously computed data flow information to incorporate changes in the programs. Incremental data flow analysis algorithms based on elimination methods are found in [6, 8, 47, 49] whereas those based on iterative methods are reported in [20, 42]; [34] falls in both the categories whereas [60] does not fall in either one. An early comparison of various incremental methods is contained in [7]. Some recent works on incremental data flow analysis are included in [27, 53, 54].

Acknowledgments

This chapter has benefited from the feedback of many people. Those who went beyond the original request and reviewed multiple drafts are Aditya Kanade, Amitabha Sanyal, Bageshri Sathe and Vinay Kakade. Bageshri's critical reviews clarified Sections 1.5 and 1.6. Section 1.3 has become significantly better due to a series of intense discussions with Amitabha Sanyal; in the process I gained many profound insights. Remarks by an anonymous reviewer helped me to relate Tarski's lemma to much of what was already described in the chapter. Finally, I would like to thank Ajit Agrawal, Asit Verma, D.M. Dhamdhere, Mandar Sahasrabudde and Sandeep Pendharkar for providing useful feedback.

References

[1] A.V. Aho, R. Sethi and J.D. Ullman, *Compilers — Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA, 1986.

[2] F.E. Allen and J. Cocke, A program data flow analysis procedure, *Communications of ACM*, 19(3), 137–147, 1977.

[3] A.W. Appel and M. Ginsburg, *Modern Compiler Implementation in C*, Cambridge University Press, New York, 1998.

[4] J. Banning, An Efficient Way to Find the Side Effects of Procedure Calls and Aliases of Variables, in POPL '79, 1979, pp. 29–41.

[5] R. Bodik, R. Gupta and M.L. Soffa, Complete Removal of Redundant Computations, in PLDI '98, 1998, pp. 1–14.

[6] M. Burke, An interval analysis approach to exhaustive and incremental interprocedural data flow analysis, *ACM TOPLAS*, 12(3), 341–395, 1990.

[7] M.G. Burke and B.G. Ryder, A critical analysis of incremental iterative data flow analysis algorithms, *IEEE Trans. Software Eng.*, 16(7), 723–728, 1990.

[8] M. Carroll and B. Ryder, Incremental Data Flow Analysis via Dominator and Attribute Updates, in POPL '88, 1988, pp. 274–284.

[9] K.D. Cooper and K. Kennedy, Efficient computation of flow insensitive interprocedural summary information, *SIGPLAN Notices*, 19(6), 247–258, 1984.

[10] K.D. Cooper and K. Kennedy, Fast Interprocedural Alias Analysis, in POPL '89, 1989, pp. 49–59.

[11] A. Deutsch, Interprocedural May-Alias Analysis for Pointers: Beyond k-limiting, in PLDI '94, 1994, pp. 230–241.

[12] D.M. Dhamdhere, A fast algorithm for code movement optimization, *ACM SIGPLAN Notices*, 23(10), 172–180, 1988.

[13] D.M. Dhamdhere, Register assignment using code placement techniques, *Comput. Languages*, 13(2), 75–93, 1988.

[14] D.M. Dhamdhere, Practical adaptation of the global optimization algorithm by Morel & Renvoise, *ACM TOPLAS*, 13(2), 291–294, 1991.

[15] D.M. Dhamdhere and J.R. Issac, A composite algorithm for strength reduction and code movement optimization, *Int. J. Comput. Inf. Sci.,* 9(3), 243–273, 1980.

[16] D.M. Dhamdhere and U.P. Khedker, Complexity of Bidirectional Data Flow Analysis, in POPL '93, 1993, pp. 397–408.

[17] D.M. Dhamdhere and H. Patil, An elimination algorithm for bidirectional data flow analysis using edge placement technique, *ACM TOPLAS*, 15(2), 312–336, 1993.

[18] D.M. Dhamdhere, B.K. Rosen and F.K. Zadeck, How to Analyze Large Programs Efficiently and Informatively, in PLDI '92, 1992, pp. 212–223.

[19] E. Duesterwald, R. Gupta and M.L. Soffa, A practical framework for demand-driven interprocedural data flow analysis, *ACM TOPLAS*, 19(6), 992–1030, 1997.

[20] V. Ghodssi, Incremental Analysis of Programs, Ph.D. thesis, Department of Computer Science, University of Central Florida, Orlando, 1983.

[21] D.W. Goodwin, Interprocedural Data Flow Analysis in an Executable Optimizer, in PLDI '95, 1997, pp. 122–133.

[22] S. Graham and M. Wegman, A fast and usually linear algorithm for global data flow analysis, *J. ACM*, 23(1), 172–202, 1976.

[23] D. Grune, H.E. Bal, C.J.H. Jacobs and K.G. Langendoen, *Modern Compiler Design*, John Wiley & Sons, New York, 2000.

[24] M.S. Hecht, *Flow Analysis of Computer Programs*, Elsevier/North-Holland, Amsterdam, 1977.

[25] J.B. Kam and J.D. Ullman, Monotone data flow analysis framework, *Acta Inf.,* 7(3), 305–318, 1977.

[26] K. Kennedy, A survey of data flow analysis techniques, in *Program Flow Analysis: Theory and Applications*, Prentice Hall, New York, 1981.

[27] U.P. Khedker, A Generalized Theory of Bit Vector Data Flow Analysis, Ph.D. thesis, Department of Computer Science and Engineering, Indian Institute of Technology, Bombay, 1997.

[28] U.P. Khedker and D.M. Dhamdhere, A generalized theory of bit vector data flow analysis, *ACM TOPLAS*, 16(5), 1472–1511, 1994.

[29] U.P. Khedker, D.M. Dhamdhere and A. Mycroft, The power of bidirectional data flow analysis, Department of Computer Science and Engineering, Indian Institute of Technology, Bombay, in preparation.

[30] G. Kildall, A Unified Approach to Global Program Optimization, in POPL '73, 1973, pp. 194–206.

[31] J. Knoop, O. Rüthing and B. Steffen, Lazy Code Motion, in PLDI '92, 1992, pp. 224–234.

[32] W. Landi and B.G. Ryder, A Safe Approximate Algorithm for Interprocedural Pointer Aliasing, in PLDI '92, 1992, pp. 235–248.

[33] W. Landi, B.G. Ryder and S. Zhang, Interprocedural Side Effect Analysis with Pointer Analysis, in PLDI '93, 1993, pp. 235–248.

[34] T.J. Marlowe and B.G. Ryder, An Efficient Hybrid Algorithm for Incremental Data Flow Analysis, in POPL '90, 1990, pp. 184–196.

[35] T.J. Marlowe and B.G. Ryder, Properties of data flow frameworks, *Acta Inf.,* 28, 121–163, 1990.

[36] F. Martin, Experimental Comparison of Call String and Functional Approaches to Interprocedural Analysis, in *CC '99*, 1999, pp. 63–75.

[37] E. Morel and C. Renvoise, Global optimization by suppression of partial redundancies, *Commun. ACM*, 22(2), 96–103, 1979.

[38] R. Morgan, *Building an Optimizing Compiler*, Butterworth-Heinemann, Oxford, 1998.

[39] S.S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann, San Fransisco, 1997.

[40] S.S. Muchnick and N.D. Jones, *Program Flow Analysis: Theory and Applications*, Prentice Hall, New York, 1981.

[41] F. Nielson, H.R. Nielson and C. Hankin, *Principles of Program Analysis*, Springer-Verlag, Heidelberg, 1998.

[42] L.L. Pollock and M.L. Soffa, An incremental version of iterative data flow analysis, *IEEE Trans. Software Eng.,* 15(12), 1537–1549, 1989.

[43] T.W. Reps, S. Horwitz and S. Sagiv, Precise Interprocedural Data Flow Analysis via Graph Reachability, in POPL '95, 1995, pp. 49–61.

[44] S.E. Richardson and M. Ganapathi, Interprocedural optimizations: experimental results. *Software Pract. Exp.,* 19(2), 149–169, 1989.

[45] B.K. Rosen, Monoids for rapid data flow analysis, *SIAM J. Comput.,* 9(1), 159–196, 1980.

[46] E. Ruf, Context-Insensitive Alias Analysis Reconsidered, in PLDI '95, 1995, pp. 13–22.

[47] B.G. Ryder, Incremental data flow analysis, in POPL '83, 1983, pp. 167–176.

[48] B.G. Ryder and M.C. Paull, Elimination algorithms for data flow analysis, *ACM Comput. Surv.,* 18, 277–316, 1986.

[49] B.G. Ryder and M.C. Paull, Incremental data flow analysis algorithms, *ACM TOPLAS*, 10(1), 1–50, 1988.

[50] M. Shapiro and S. Horwitz, Fast and Accurate Flow-Insensitive Points-to Analysis, in POPL '97, 1997, pp. 1–14.

[51] M. Sharir, Structural analysis: a new approach to data flow analysis in optimizing compilers, *Comput. Languages*, 5, 141–153, 1980.

[52] M. Sharir and A. Pnueli, Two approaches to interprocedural data flow analysis, in S.S. Muchnick and N.D. Jones, *Program Flow Analysis: Theory and Applications*, Prentice Hall, New York, 1981.

[53] V.C. Shreedhar, G.R. Gao and Y.-F. Lee, A new framework for exhaustive and incremental data flow analysis using D-J graphs, in PLDI '96, 1996, pp. 278–290.

[54] A. Sreeniwas, Program Transformations and Representations, Ph.D. thesis, Department of Computer Science and Engineering, Indian Institute of Technology, Bombay, 1998.

[55] R.E. Tarjan, Fast algorithms for solving path problems, *J. ACM*, 28(3), 594–614, 1981.

[56] A.M. Tennenbaum, Type determination in very high level languages, Technical report NSO-3, Courant Institute of Mathematical Sciences, New York University, 1974.

[57] V. Vyssotsky and P. Wegner, A graph theoretical FORTRAN source language analyzer (manuscript), AT & T Bell Laboratories, Murray Hill, NJ, 1963.

[58] R. Wilhelm and D. Maurer, *Compiler Design*, Addison-Wesley, Reading, MA, 1995.

[59] R.P. Wilson and M.S. Lam, Efficient context-sensitive pointer analysis for c programs, in PLDI '95, 1995, pp. 1–12.

[60] F.K. Zadeck, Incremental data flow analysis in a structured program editor, in SIGPLAN '84 Symposium on Compiler Construction, 1984, pp. 132–143.

1.A Appendix: Some Examples of Advanced Analyses

This section presents the data flow equations of some advanced analyses that stretch the limits of the classical theory of data flow analysis. We specify them in their original form in terms of bit vectors instead of sets. The confluence operation $\cap$ is now represented by the bitwise AND, which is denoted by "." when it is used as a binary operation and by Π when it is used to range over many operands. Similarly, $\cup$ is indicated by the bitwise OR, which is denoted by "$+$" when it is used as a binary operation and by Σ when it is used to range over many operands.

1.A.1 Data Flow Equations and Brief Descriptions

MR-PRE [37] performs global program optimization by suppression of partial redundancies. Partial redundancy of an expression is used to motivate its hoisting. Because common subexpression elimination and loop invariant movement are special cases of partial redundancy elimination, the algorithm unifies several traditional optimizations within a single framework. Equations (1.61) and (1.62) represent the data flows of MR-PRE:

$$\texttt{PPIN}_i = \texttt{Const}_i \cdot (\texttt{Antloc}_i + \texttt{Transp}_i \cdot \texttt{PPOUT}_i) \cdot \prod_{p \in pred(i)} (\texttt{AVOUT}_p + \texttt{PPOUT}_p) \tag{1.61}$$

$$\texttt{PPOUT}_i = \prod_{s \in succ(i)} (\texttt{PPIN}_s) \tag{1.62}$$

Local property $\texttt{Antloc}_i$ represents information concerning upward exposed occurrences of an expression *e* in node *i* of the program flow graph, whereas $\texttt{Transp}_i$ reflects the absence of definitions of *e*'s operands in the node. Partial redundancy is represented by the data flow property of partial availability $\texttt{PAVIN}_i/\texttt{PAVOUT}_i$ (which is a part of $\texttt{Const}_i$). It is defined using available expressions analysis with the confluence operator changed to Σ. Safety of hoisting is incorporated by considering only those expressions that are very busy at the entry or exit of a node. This is incorporated by the $\prod$ term of the $\texttt{PPOUT}_i$ (Equation 1.62). A property $\texttt{Insert}_i$ (not defined here) identifies nodes where occurrences of expression *e* should be inserted. $\texttt{PPIN}_i \cdot \neg\texttt{Antloc}_i$ identifies occurrences that become redundant following the insertion.

In the load store insertion algorithm [13], the problem of placing `Load` and `Store` instructions of a variable to characterize its live range for register assignment is modeled as redundancy elimination of the `Load` and `Store` instructions. Here, we present the data flow equations of the problem of the placement of `Store` instructions Equations (1.63) and (1.64), which is a dual of MR-PRE in that this analysis tries to identify suitable program points to *sink* a `Store` instruction as much as possible:

$$\texttt{SPPIN}_i = \prod_{p \in pred(i)} (\texttt{SPPOUT}_p) \tag{1.63}$$

$$\begin{aligned} \texttt{SPPOUT}_i = {} & \texttt{DPANTOUT}_i \cdot (\texttt{Dcomp}_i + \texttt{Dtransp}_i \cdot \texttt{SPPIN}_i) \\ & \cdot \prod_{s \in succ(i)} (\texttt{DANTIN}_s + \texttt{SPPIN}_s) \end{aligned} \tag{1.64}$$

In the MMRA [14], an additional term with $\sum$ as the confluence, is added to the $\texttt{PPIN}_i$ equation to suppress redundant hoisting of an expression (Equation (1.65)). Other terms in the equations are the same as in MR-PRE:

$$\begin{aligned} \texttt{PPIN}_i = {} & \texttt{Const}_i \cdot (\texttt{Antloc}_i + \texttt{Transp}_i \cdot \texttt{PPOUT}_i) \\ & \cdot \prod_{p \in pred(i)} (\texttt{AVOUT}_p + \texttt{PPOUT}_p) \\ & \cdot \sum_{p \in pred(i)} (\texttt{PPIN}_p \cdot \neg\texttt{Antloc}_p + \texttt{AVOUT}_p) \end{aligned} \tag{1.65}$$

$$\texttt{PPOUT}_i = \prod_{s \in succ(i)} (\texttt{PPIN}_s) \tag{1.66}$$

The technique of edge placement aims at total elimination of partial redundancies by placing an expression that is partially redundant in node i, but cannot be safely placed in a predecessor node j, into a synthetic node placed along edge $j \to i$. This simplifies the data flow dependencies by eliminating the $\prod$ term of the MR-PRE $\texttt{PPIN}_i$ equation. The edge placement algorithm (EPA) [12] uses the technique of edge placement, and also incorporates the suppression of redundant hoisting by adding a Σ term in the $\texttt{PPIN}_i$ equation (Equation (1.67)).

$$\begin{aligned} \texttt{PPIN}_i &= \texttt{Const}_i \cdot (\texttt{Antloc}_i + \texttt{Transp}_i \cdot \texttt{PPOUT}_i) \\ &\quad \cdot \sum_{p \in pred(i)} (\texttt{PPIN}_p \cdot \neg\texttt{Antloc}_p + \texttt{AVOUT}_p) \end{aligned} \tag{1.67}$$

$$\texttt{PPOUT}_i = \prod_{s \in succ(i)} (\texttt{PPIN}_s) \tag{1.68}$$

Data flow analysis based type inferencing for dynamically typed languages was proposed by Tennenbaum [56]; however, its formulation presented in [1] fails to qualify as a formal data flow framework, suffers from imprecision and is not amenable to complexity analysis. A formulation that eliminates these drawbacks is proposed in [29]. This is what we refer to as KDM type inferencing. We merely reproduce its data flow equations without defining the flow functions and the confluence operation:

$$\texttt{IN}_n = \begin{cases} Outside\_Info \sqcap f_n^B(\texttt{OUT}_n), & n \equiv n_0 \\ (\bigsqcap_{p \in Pred(n)} g_{p\to n}^F(\texttt{OUT}_p)) \sqcap f_n^B(\texttt{OUT}_n), & n \not\equiv n_0 \end{cases} \tag{1.69}$$

$$\texttt{OUT}_n = \begin{cases} Outside\_Info \sqcap f_n^E(\texttt{IN}_n), & n \equiv n_\infty \\ (\bigsqcap_{s \in Succ(n)} g_{n\to s}^B(\texttt{IN}_s)) \sqcap f_n^F(\texttt{IN}_n) \sqcap \texttt{OUT}_n, & n \not\equiv n_\infty \end{cases} \tag{1.70}$$

1.A.2 Taxonomies of Data Flow Analysis

In this section we refine the taxonomy in Section 1.3.8 to include advanced analyses. Figure 1.23 presents a larger canvas by laying out the possible combinations of confluences and directions of flows when the confluences are all paths or any paths (and data flow values are Boolean). It places some data flow problems at appropriate coordinates on the grid:

- The horizontal sides of the grid represent the possibilities for the direction of flows. The left corner indicates forward flows whereas the right corner indicates backward flows. The center point indicates that the number of forward flows is the same as the number of backward flows.

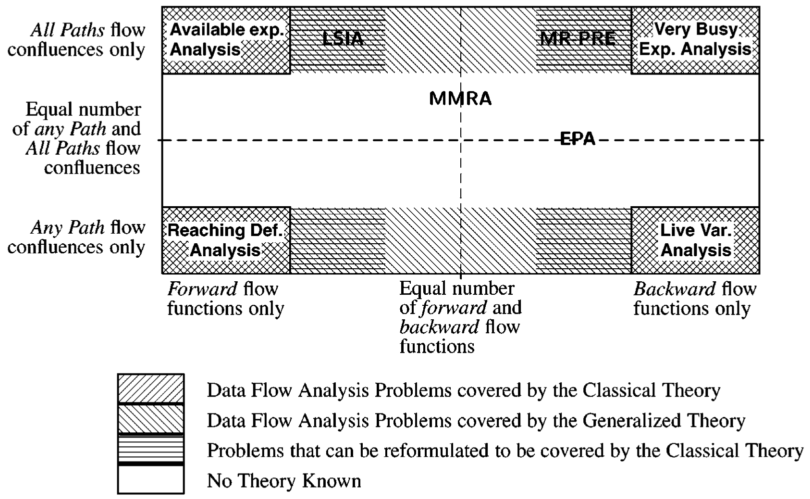

All Paths flow confluences only
Available exp. Analysis
LSIA
MR-PRE
Very Busy Exp. Analysis
Equal number of *any Path* and *All Paths* flow confluences
MMRA
EPA
Any Path flow confluences only
Reaching Def. Analysis
Live Var. Analysis
Forward flow functions only
Equal number of *forward* and *backward* flow functions
Backward flow functions only
Data Flow Analysis Problems covered by the Classical Theory
Data Flow Analysis Problems covered by the Generalized Theory
Problems that can be reformulated to be covered by the Classical Theory
No Theory Known

FIGURE 1.23 Taxonomy of data flow analysis problems with ⊓ and ⊔ confluences.

As we move away from the center toward the left, we have a larger number of forward flows than backward flows. Exactly opposite happens when we move from the center to the right.

- The vertical sides represent the possibilities for the confluence of flows. The bottom corner indicates any path confluences whereas the top corner indicates the all paths confluences. The center point indicates that the number of all paths confluences is the same as the number of any path confluences. As we move away from the center toward the top, we have a larger number of all paths confluences than any path confluences. Exactly opposite happens when we move from the center to the bottom.

It is easy to see that the four classical data flow problems appear in the four corners of the grid. We explain the placement of the other data flow problems in the following:

- *Partial redundancy elimination (MR-PRE).* MR-PRE uses backward node flows as well as edge flows. However, forward edge flows exist but forward node flows do not exist. Thus, it is only partially bidirectional, and the technique of edge splitting can be used to make this data flow problem amenable to the classical theory of data flow analysis. Both forward edge flows and backward edge flows are combined using $\sqcap$, which is an all paths confluence.
- *Load Store Insertion Algorithm (LSIA).* The `Store`-sinking equations of LSIA are duals or MR-PRE as far as the direction is concerned.
- *Edge placement algorithm (EPA).* In this case, forward edge flows use any path confluence, whereas the backward edge flows use all paths confluence. Replacing the all path forward confluence of MR-PRE by an any path forward confluence reduces the complexity of EPA but fails to make it amenable to the classical theory because it still has two confluences.
- *Modified Morel–Renvoise algorithm (MMRA).* In this case there is an additional any path forward edge flow component. Thus, it has two forward edge flows (one uses any path confluence whereas the other uses all paths confluence), one backward edge flow (which uses all paths confluence) and one backward node flow. MMRA does not have forward node flow.

Note that the generalized theory covers a larger ground than the classical theory in Figure 1.23, and yet a large ground remains uncovered. As of now very few data flow problems occur in the uncovered area. However, we believe that the development of theory for these problems may encourage researchers to experiment and find out many real applications where such data flow analyses can prove to be useful.

Now we present another taxonomy in which the variation is not in terms of the number of confluences but in terms of the type of confluence. This discussion allows us to compare constant

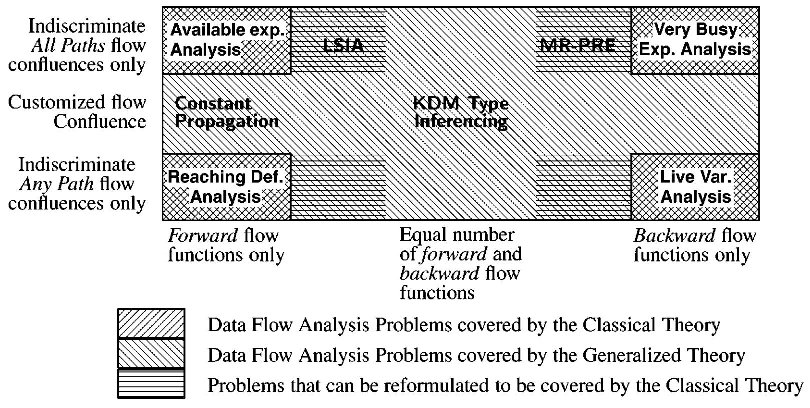

Indiscriminate *All Paths* flow confluences only
Customized flow Confluence
Indiscriminate *Any Path* flow confluences only
Available exp. Analysis
LSIA
MR-PRE
Very Busy Exp. Analysis
Constant Propagation
KDM Type Inferencing
Reaching Def. Analysis
Live Var. Analysis
Forward flow functions only
Equal number of *forward* and *backward* flow functions
Backward flow functions only
Data Flow Analysis Problems covered by the Classical Theory
Data Flow Analysis Problems covered by the Generalized Theory
Problems that can be reformulated to be covered by the Classical Theory

FIGURE 1.24 Taxonomy of data flow analysis problems with general (single) confluence.

propagation and KDM type inferencing with other data flow problems by placing them on a grid in Figure 1.24. In this, the vertical sides of the grid indicate whether the confluence is indiscriminate any path or all paths or special confluence:

- *Constant propagation*. Constant propagation uses a forward node flow and a forward edge flow that is combined using a special confluence operation.
- *KDM type inferencing*. This analysis uses information from successors as well as predecessors. Thus, it is a bidirectional data flow problem that uses forward node as well as edge flows and backward node as well as edge flows. Both forward and backward edge flows are combined using a special confluence operation.

2

Automatic Generation of Code Optimizers from Formal Specifications

Vineeth Kumar Paleri
Regional Engineering College, Calicut

2.1 Introduction

Code transformations form an integral part of high-performance computing systems. Optimizing compilers try to reduce the execution time or the size of a program by analyzing it and applying various transformations. Code transformations are broadly classified into scalar transformations and parallel transformations; scalar transformations reduce the number of instructions executed by the program and parallel transformations, in general, maximize parallelism and memory locality.

Different levels of representation of the source code exist at which different transformations are applied: high-level intermediate language, low-level intermediate language and object code. All semantic information of the source program is available at the high-level intermediate language level to apply high-level transformations. Low-level transformations are applied at the low-level intermediate language level and machine-specific transformations are performed at the object code level.

Code transformation is a complex function of a compiler involving data flow analysis and modification over the entire program. In spite of its complexity, hardly any tool exists to support this function of the compiler, unlike the functions lexical analysis, parsing and code generation; we have well-established tools, such as Lex and Yacc, for lexical analysis and parsing, and tools are now available for the code generation phase also.

Automatic generation of code transformers involves identification of a framework for the specification of code transformations, design of a specification language based on the framework

0-8493-1240-7/03/$0.00+$1.50

identified, specification of the transformations in the specification language and development of a generator to produce code for compiler transformers from their specifications.

Code transformations have been discussed extensively in the literature [1, 4, 17, 18, 26]. Aho, Sethi and Ullman [1] describe all traditional scalar transformations in detail, along with different approaches to data flow analysis. Bacon, Graham and Sharp [4] give an extensive, but brief, coverage of all code transformations — both scalar and parallel. Muchnick [17] describes all code transformations, in detail. Padua and Wolfe [18] and Wolfe [26] deal more with parallel transformations. Banerjee [2, 3] deals with transformations specifically for loops. Hecht [14], and Muchnick and Jones [16] deal with the data flow analysis for code transformations. Ryder and Paull [21] discuss different elimination algorithms for data flow analysis. Click and Cooper [8] describe combining analyses and optimizations.

We start with the formal specification of code transformations using dependence relations. Bacon, Graham and Sharp [4], Padua and Wolfe [18], Whitfield and Soffa [31] and Wolfe [26, 27] give informal definitions of dependence relations. Because dependence relations form the basis of our specifications, we need formal definitions for them, as well. Cytron et al. [6], Ferrante, Ottenstein and Warren [11], and Sarkar [22] give formal definitions of control dependence. Whereas Ferrante, Ottenstein and Warren [11], and Sarkar [22] use the concept of postdominance as the basis of their definition, the definition by Cytron et al. is based on the concept of dominance frontiers. Bilardi and Pingali [5] give a generalized notion of control dependence. Of the two formal definitions for data dependence — one by Banerjee [2] and the other by Sarkar [22] — Banerjee's definition is specifically for loop nests. Sarkar's definition is suitable in a more general setting.

Formal specifications for some of the scalar and parallel transformations, using dependence relations, were given by Whitfield and Soffa in their work on ordering optimizing transformations [28], automatic generation of global optimizers [29] and study of properties of code improving transformations [31].

In the later part we present the development of a system to generate code for compiler transformers from their specifications and to conduct experiments with the code thus generated. Early attempts to build tools for code transformations, including the analyses needed, were restricted to localized transformations. Davidson and Fraser [9], Fraser and Wendt [12], and Kessler [15] describe automatic generation of peephole optimizers. Tjiang and Hennessy [24], in a more recent work, describe a tool to generate global data flow analyzers.

Whitfield and Soffa [29–31] have developed a tool, called Genesis, for generating code transformers from formal specifications written in the specification language Gospel, designed by them. For each optimization, Gospel requires the specification of the preconditions and the actions to optimize the code. The preconditions consist of the code patterns and the global dependence information needed for the optimization. The code patterns express the format of the code, and the global information determines the control and data dependences that are required for the specified optimization. The actions take the form of primitive operations that occur when applying code transformations. The primitive actions are combined to express the total effect of applying a specific optimization. Genesis assumes a high-level intermediate representation that retains the loop structures from the source program. The high-level intermediate representation allows the user to interact at the source level for loop transformations. They have generated code for many of the scalar and parallel transformations from a prototype implementation of the generator.

Paleri [19] describes a code transformation system that provides an environment in which one can specify a transformation — using dependence relations — in the specification language designed for the purpose, generate code for a transformer from its specification using the transformer generator and experiment with the transformers generated on real-world programs.

In this work all traditional scalar transformations and some of the parallel transformations have been specified in the framework using dependence relations, but the implementation is restricted to scalar transformations. The specification language is powerful enough to express formal specifications for all traditional scalar transformations. The system takes a program to be transformed — in C or FORTRAN — as input, translates it to intermediate code, requests the user to choose the transformation to be performed, computes the required dependence relations using the dependence analyzer, applies the specified transformation and converts the transformed intermediate code back to high level. Code for all traditional scalar transformations have been generated from their specifications using the system and applied them on LINPACK benchmark programs. The system can be used either as a tool for the generation of code transformers or as an environment for experimentation with code transformations.

This article presents the design and implementation of the system for automatic generation of code optimizers from formal specifications developed by Paleri [19].

2.2 Formal Specification of Code Transformations

Even though code transformations — both scalar and parallel — have been discussed extensively in the literature [1, 4, 17, 18, 26, 27], only a few attempts have been made in formalizing them [20, 28–31]. Formal specification of code transformations not only helps in the precise understanding of those transformations but also finds many applications such as ordering of transformations [28, 30, 31] and code transformer generators [29–31].

In dependence relations, we have a uniform framework for the specification of both scalar and parallel transformations [2, 4, 17, 26, 27]. A dependence is a relationship between two computations that places constraints on their execution order. Dependence relations, representing these constraints, are used to determine whether a particular transformation can be applied without changing the semantics of the computations.

Different transformations are applied at different levels of representation of the source code. In general, scalar transformations use the low-level intermediate representation and parallel transformation the high-level intermediate representation. We require an intermediate representation satisfying the requirements of both scalar and parallel transformations for their specification.

All traditional scalar transformations and some of the parallel transformations have been specified in the dependence relations framework. The specification for each transformation constitutes a precondition part and an action part. The precondition part specifies the conditions to be satisfied for the application of the transformation and the action part specifies the steps for performing the transformation using some primitive actions. The specifications are expressed in a notation similar to predicate logic.

We first discuss about the framework used for the specification before giving the specifications for various transformations.

2.2.1 Framework for Specification

2.2.1.1 Dependence Relations

A dependence is a relationship between two computations that places constraints on their execution order. Dependence analysis identifies these constraints, which are then used to determine whether a particular transformation can be applied without changing the semantics of the computation. The two kinds of dependences are control dependence and data dependence. Whereas control dependence is a consequence of the flow of control in a program, data dependence is a consequence of the flow of data.

2.2.1.1.1 *Control Dependence.*

There is a control dependence [4, 6, 11, 22, 27] between statement S_i and statement S_j, denoted $S_i\delta^c S_j$, when statement S_i determines whether S_j can be executed. In the following example, S_j is control dependent on S_i (i.e., $S_i\delta^c S_j$):

$$\begin{array}{ll} S_i: & \text{if}(x\ =\ c) \\ S_j: & \quad y\ :=\ 1 \end{array}$$

To give a formal definition for control dependence we first define the terms control flow graph, dominance, dominance frontier, postdominance and postdominance frontier. We assume that a program is represented as a control flow graph (CFG) with each node as a single instruction or a basic block.

Definition 2.1 (control flow graph). *A CFG is a tuple (N, E, s) such that:*

- *(N, E) is a directed graph (or multigraph), with the set of nodes N and the set of edges E*
- *$s \in N$ is a unique entry node*
- *All nodes are reachable from s, that is, $(\forall v \in N)[s \overset{*}{\to} v]$*

Note that $s \overset{*}{\to} v$ means v is reachable from s by traversing zero or more edges.

Definition 2.2 (Dominance). *A node m in a CFG dominates a node n if every path from s to n includes m, denoted m dom n.*

By this definition, every node dominates itself, and s dominates every node in the CFG, including itself.

Definition 2.3 (dominance frontier). *The dominance frontier of a node m is the set of nodes r such that m dominates some predecessor of r, but not all; that is:*

$$df(m) = \{r \mid \exists p, q\ \in\ pred(r)[m\ dom\ p \wedge m\ \overline{dom}\ q]\}$$

where, $pred(r)$ is the set of predecessors of r, and $m\ \overline{dom}\ q$ means m does not dominate q.

Note that nodes in the dominance frontier are nodes with more than one predecessor in the CFG.

To define postdominance we must also have a unique exit node e, which is reachable from all other nodes. We define a single-exit CFG as $CFG = (N, E, s, e)$, where N, E, and s are as before and $e\ \in\ N$ is reachable from all other nodes; i.e.:

$$(\forall v\ \in\ N)[v \overset{*}{\to} e]$$

Definition 2.4 (postdominance). *A node n in a single-exit CFG postdominates a node m, denoted n pdom m, if every path from m to e includes n.*

By this definition, every node postdominates itself, and e postdominates every node in the graph, including itself.

Definition 2.5 (postdominance frontier). *The postdominance frontier of a node n is the set of nodes r such that n postdominates some successor of r, but not all; that is:*

$$pdf(n) = \{r \mid (\exists m\ \in\ succ(r))[n\ pdom\ m \wedge n\ \overline{spdom}\ r\}]$$

where, $succ(r)$ is the set of successors of r, and $spdom$ means strict postdominance; that is:

$$n\ spdom\ m \equiv (n\ pdom\ m) \wedge (n \neq m)$$

With a flow graph $G = (N_G, E_G, s_G, e_G)$, we can define the reverse CFG RCFG, $G^R = (N_G, E_G^R, e_G, s_G)$, with the same set of nodes, but with an edge $(n \rightarrow m) \in E_G^R$ for each edge $(m \rightarrow n) \in E_G$. Note that s_G and e_G switch roles. The dominator relationship in the RCFG is the same as the postdominator relationship in the original CFG.

Definition 2.6 (control dependence). *A node n in a CFG is control dependent on node m if:*

1. *n postdominates some successor of m.*
2. *n does not postdominate all successors of m.*

A less formal definition is to say that n is control dependent on m if:

1. Following one edge out of m eventually always executes n.
2. Choosing some other edge out of m may avoid n.

Note that n is control dependent on m if and only if $m \in pdf(n)$.

2.2.1.1.2 ***Data Dependence.***

Two statements have a data dependence [2, 4, 22, 26, 27] between them if they cannot be executed simultaneously due to conflicting uses of a variable. The three types of data dependence relations are flow dependence, antidependence and output dependence. If S_i and S_j are two statements, we say S_j is flow dependent on S_i, denoted $S_i \delta S_j$, if a variable is defined in S_i and a subsequently executed statement S_j uses the definition in S_i. An antidependence occurs between two statements S_i and S_j, denoted $S_i \bar{\delta} S_j$, when a variable is used in S_i and redefined in a subsequently executed statement S_j. When a variable is defined in a statement S_i and redefined in a subsequently executed statement S_j, we say an output dependence exists between S_i and S_j, denoted $S_i \delta^o S_j$.

The following examples illustrate the three dependences:

$$S_i: x := a + b$$
$$S_j: y := x + c$$

Here, $S_i \delta S_j$ due to the variable x;

$$S_i: x := y + a$$
$$S_j: y := a + b$$

Here, $S_i \bar{\delta} S_j$ due to the variable y;

$$S_i: z := x + a$$
$$S_j: z := y + b$$

Here, $S_i \delta^o S_j$ due to the variable z.

A fourth possible data dependence relation exists between two statements S_i and S_j called input dependence, denoted $S_i \delta^i S_j$, when two accesses to the same variable in these statements are reads. In the following example, $S_i \delta^i S_j$ due to the variable x:

$$S_i: y := x + a$$
$$S_j: z := x + b$$

In general, no ordering is implied in this case, because it does not matter which statement, S_i or S_j, reads from the variable first. However, there are cases when this relation do matter. For this reason and for completeness we include input dependence in the set of dependence relations.

DEFINITION 2.7 (data dependence). *Consider two statements S_i and S_j in a CFG. A data dependence exists between statements S_i and S_j with respect to variable x if and only if:*

1. *A path $P : S_i \overset{*}{\rightarrow} S_j$ exists in the CFG.*
2. *Variable x is written by S_i and later read by S_j (flow dependence), x is read by S_i and later written by S_j (antidependence), x is written by S_i and later rewritten by S_j (output dependence) or x is read by S_i and later reread by S_j (input dependence).*
3. *Variable x is not written in between the two references in S_i and S_j on the path from S_i to S_j.*

The third condition means that variable x should not be written in the time period between the two references in S_i and S_j during execution.

It is often convenient to be able to ignore the kind of the dependence relation. In such cases we can simply say that S_j is dependent on S_i and write it as $S_i \delta^* S_j$.

2.2.1.1.3 *Data Dependence in Loops.*

In a loop, each statement may be executed many times, and for many transformations it is necessary to describe dependences that exist between statements in different iterations, called *loop-carried* dependences. To discover whether a dependence exists in a loop nest, it is sufficient to determine whether any of the iterations can write a value that is read or written by any of the other iterations.

An iteration in a perfectly nested loop of d loops can be uniquely named by a vector of d elements $I = (i_1, \ldots, i_d)$, where each index falls within the iteration range of its corresponding loop in the nesting (i.e., $l_p \leq i_p \leq u_p$ where, l_p and u_p are the lower and upper bounds of loop p). The outermost loop corresponds to the leftmost index.

A reference in iteration J can depend on another reference in an earlier iteration I, but not on a reference in an iteration after iteration J. By using the $\prec$ relation to formalize the notion of *before*, we have:

$$I \prec J \quad \text{iff} \quad \exists p : (i_p < j_p \wedge \forall q : q < p : i_q = j_q)$$

A reference in some iteration J depends on a reference in iteration I if and only if at least one reference is a write and:

$$I \prec J \wedge \forall p : 1 \leq p \leq d : f_p(I) = g_p(J)$$

where the functions $f_p(i_1, \ldots, i_d)$ and $g_p(i_1, \ldots, i_d)$ map the current values of the loop iteration variables to integers that index the pth dimension of the array. In other words, a dependence exists when the values of the subscripts are the same in different iterations. By extending the notation for dependences to dependences between iterations, we write $I\delta^* J$ for a dependence from iteration I to iteration J.

When $I\delta^* J$, the *dependence distance* is defined as $J - I = (j_1 - i_1, \ldots, j_d - i_d)$. When all the dependence distances for a specific pair of references are the same, the potentially unbounded set of dependences can be represented by the dependence distance. When a dependence distance is used to describe the dependences for all iterations, it is called a *distance vector*.

A legal distance vector V must be lexicographically positive, meaning that $0 \prec V$ (the first nonzero element of the distance vector must be positive). A negative element in the distance vector means that the dependence in the corresponding loop is on the higher numbered iteration (i.e., a later iteration of that loop). If the first nonzero element was negative, this would indicate a dependence on a future iteration, which is impossible.

For the nested loop given next there is a loop-carried dependence $S_j \delta S_i$, with a distance vector (0, 1), due to the references $B[I_1, I_2 + 1]$ in S_j and $B[I_1, I_2]$ in S_i:

```
$do I_1 = LB_1, UB_1$
  $do I_2 = LB_2, UB_2$
    $S_i : A[I_1, I_2] = B[I_1, I_2] + C[I_1, I_2]$
    $S_j : B[I_1, I_2 + 1] = A[I_1, I_2] + B[I_1, I_2]$
  $enddo$
$enddo$
```

It should be noted that distance vectors describe dependences among iterations, not among array elements.

In some cases it is not possible to determine the exact dependence distance at compile time, or the dependence distance may vary between iterations. In such cases a direction vector is used to describe such dependences, where enough information is available to partially characterize the dependence.

For a dependence $I\delta^* J$, we define the direction vector $\Psi = (\Psi_1, \ldots, \Psi_d)$, where

$$\Psi_p = \begin{cases} < & \text{if } i_p < j_p \\ = & \text{if } i_p = j_p \\ > & \text{if } i_p > j_p \end{cases}$$

A particular dependence between statements S_i and S_j is denoted by writing $S_i \delta^*_\psi S_j$. For the nested loop given earlier, corresponding to the distance vector (0, 1) we have the direction vector (=, <) and the dependence is denoted by $S_j \delta_{(=,<)} S_i$.

The general term *dependence vector* is used to encompass both distance and direction vectors. Note that a direction vector entry of < corresponds to a distance vector entry that is greater than zero.

The dependence behavior of a loop is described by the set of dependence vectors for each pair of possibly conflicting references. These can be summarized into a single loop direction vector with $\Psi_k \in \{<, =, >, \leq, \geq, \neq, *\}$, where $*$ stands for *any* of the direction vector entries <, =, or >, at the expense of some loss of information. For example, the set of direction vectors (=, <), (<, =), (<, >) can be summarized as (≤,*).

2.2.1.2 Intermediate Representation

Different transformations are applied at different levels of representation of the source code: high-level intermediate representation, low-level intermediate representation and machine code. All semantic information of the source program is available at the high-level intermediate representation to apply high-level transformations. Low-level transformations are applied on the low-level intermediate representation, and machine-specific optimizations are performed on machine code.

Parallel transformations, in general, use the high-level intermediate representation in which references to source-level information such as loops and arrays can be made. Scalar transformations are applied on the machine-independent low-level intermediate representation. Statements at this level are assumed to be in the three-address code form:

$$dst := src1\ op\ src2$$

where, *op* represents the opcode, *src*1 and *src*2 are the source operands and *dst* is the destination operand of the statement.

Many possible intermediate representations may be useful in our context, namely, CFG, data dependence graph, program dependence graph [11] and static single assignment form. We assume an intermediate representation, such as an abstract syntax tree in Stanford University Intermediate Format [25], which satisfies the requirements of both scalar and parallel transformations.

2.2.1.3 Specification Format

The specification for each of the transformation has two parts — a precondition part and an action part. The precondition part specifies the condition as a combination of some basic conditions, which has to be satisfied for the application of the transformation; and the action part consists of a sequence of primitive actions to perform the transformation. The precondition is expressed in a format similar to the predicate form (Q:R:P), read as, "Q such that R for which P is true," where Q is the quantifier, R the range and P the predicate. The action part is expressed by enclosing the actions in curly braces.

Basic conditions in the precondition part include conditions to identify interested program elements such as statements and loops, and checks for dependence relations between pairs of statements.

The primitive actions used to perform the transformations include creation, insertion, deletion and modification of program elements. Following are some of the primitive actions used in the specifications:

delete_stmt(S_i)	:	delete statement S_i
delete_node(n)	:	delete basic block n
new_operand()	:	create a new operand using the given argument; the argument may be a symbol or a value
replace_operand(opr1,opr2)	:	replace the operand *opr*1 by the new operand *opr*2
new_stmt()	:	create a new statement using the given arguments; the arguments are the opcode and operands of the new statement
replace_stmt(S_i,S_j)	:	replace the statement S_i by the new statement S_j
insert_stmt(S_i,position)	:	insert the new statement S_i at the indicated position; possible positions are before and after a statement
move_stmt(S_i,position)	:	move the existing statement S_i to the indicated position; possible positions are before a statement, after a statement and loop preheader
eval(op,opr1,opr2)	:	return the value of the expression with *opr*1 and *opr*2 as operands of the operation *op*

The preceding actions are mainly with respect to the program element statement and the set of primitive actions include similar operations for other program elements.

For loops we assume a syntax similar to FORTRAN *do* loops:

```
do i = lb, ub
   loop − body
enddo
```

and denote the header, end, lower bound, upper bound, control variable and body of a loop L by L.header, L.end, L.initial, L.final, L.lcv and L.body, respectively.

2.2.2 Formal Specifications for Code Transformations

This section gives formal specifications for all traditional scalar transformations and some of the parallel transformations. The transformations considered are constant propagation, constant folding, useless code elimination, unreachable code elimination, copy propagation, common subexpression elimination, invariant code motion, induction variable elimination, loop interchange, loop fusion, loop reversal and loop skewing. All specifications given are for global transformations.

In the specifications given next $S_i.src1$, $S_i.src2$, $S_i.dst$ and $S_i.op$ represent the first source operand, the second source operand, the destination operand and the operation, respectively, of statement S_i. For a conditional branch statement S_i, $S_i.op$ can be any of the relational operators.

2.2.2.1 Constant Propagation

Constant propagation is a transformation that propagates values defined in a statement of the form $x := c$, for a variable x and a constant c, through the entire program. For the constant value, defined by a statement S_i, to be propagated the following conditions have to be satisfied:

1. There should be a statement S_j in which a use of the variable defined in S_i appears.
2. All definitions of the variable reaching the use at S_j must have the same value.

2.2.2.1.1 ***Formal Specification.***

```
$\exists S_i \exists S_j \exists opr$: $constant\_defn(S_i) \wedge S_i \delta S_j \wedge opr \in source\_operands(S_j) \wedge S_i.dst = S_j.opr$
        /* constant_defn($S_i$) ≡ $S_i$.opcode = ASSIGN ∧ type($S_i$.dst) = VAR
                                            ∧ type($S_i$.src1) = CONST. */
        : $(\forall S_k : S_k \delta S_j \wedge S_k \neq S_i \wedge S_k.dst = S_j.opr$
              : $constant\_defn(S_k) \wedge S_k.src1 = S_i.src1)$
        /* All definitions from $S_k$'s reaching the use at $S_j.opr$ have the same value. */
        $\{replace\_operand(S_j.opr, S_i.src1);\}$
```

The preceding specification is less conservative compared with other specifications available in the literature [28, 31]. Although other specifications do not consider the case where more than one definition are reaching the use, the specification given earlier considers this case also.

2.2.2.2 Constant Folding

Constant folding refers to the evaluation at compile time of expressions whose values are known to be constants. It involves determining that all operands in an expression are constants, performing the evaluation of the expression and replacing the expression by its value. Constant folding may not be legal if the operations performed at compile time are not identical to those that would have been performed at run time. In the following specification, if the expression folded is a Boolean expression in the condition part of a conditional branch statement, then the conditional branch statement is either deleted or replaced by a jump statement, depending on the truth-value of the condition.

2.2.2.2.1 ***Formal Specification.***

```
$\exists S_i :: foldable\_stmt(S_i)$
    /* foldable_stmt($S_i$) ≡ type($S_i$.src1) = CONST ∧ type($S_i$.src2) = CONST. */
    {
      $if(conditional\_branch\_stmt(S_i))$
      {
        $if(eval(S_i.op, S_i.src1, S_i.src2))$
          $replace\_stmt(S_i, new\_stmt(JMP, S_i.dst));$
          /* Replace the conditional branch statement by a jump.
             $S_i.dst$ is the target of jump. */
        $else$
          $delete\_stmt(S_i);$
      }
      $else$
        $replace\_stmt(S_i, new\_stmt(ASSIGN, S_i.dst,$
                                        $new\_operand(eval(S_i.op, S_i.src1, S_i.src2))));$
        /* Replace the constant expression by its value. */
    }
```

When a conditional branch statement is folded, it may leave unreachable code. Removal of such unreachable code is not considered as part of constant folding transformation with the assumption that this code is removed later using unreachable code elimination.

2.2.2.3 Useless Code Elimination

A statement is useless if it computes only values that are not used on any executable path leading from the statement. Even though original programs may include useless code, it is much more likely to arise from other code transformations. Useless code elimination is a form of dead code elimination.

2.2.2.3.1 *Formal Specification.*

```
$\exists S_i : definition\_stmt(S_i)$
    /* A statement is a definition statement if it involves the definition of a variable. */
    $: (\nexists S_j : S_j \neq S_i : S_i \delta S_j)$
    $\{delete\_stmt(S_i); \}$
```

2.2.2.4 Unreachable Code Elimination

Unreachable code is code that cannot be executed regardless of the input data. Such code is likely to arise as a result of other transformations — constant folding, as specified earlier, for example. Whereas useless code elimination removes code that is executable but has no effect on the result of the computation, unreachable code elimination removes codes that are never executable. Unreachable code elimination is another form of dead code elimination.

2.2.2.4.1 *Formal Specification.*

```
$\exists n : n \in N$
    /* N is the set of nodes in the CFG, each node representing a basic block. */
    $: (\nexists m : m \in N : m \delta^c n)$
    $\{delete\_node(n); \}$
```

The preceding specification assumes an augmented CFG in which every reachable node is control dependent on some node in the graph [11]. The augmented CFG is constructed by adding two additional nodes, namely, Entry and Exit, to the CFG, such that an edge exists from Entry to any basic block at which the program can be entered, and an edge exists to Exit from any basic block that is an exit point for the program. An edge from Entry to Exit is also added as part of the construction of the augmented CFG.

2.2.2.5 Copy Propagation

Copy propagation is a transformation that, given an assignment $x := y$ for some variables x and y, replaces later uses of x with uses of y, as long as intervening instructions have not changed the value of either x or y. Although this transformation by itself does not improve the code, this may leave the original assignment statement useless, and removal of this useless statement improves the code.

For a copy statement, $S_i : x := y$, we can replace any later use u of the variable x, where this definition is used, by the variable y if the following conditions are satisfied:

1. All definitions S_k of x reaching u must be a copy statement with variable y as the source operand.
2. On every path from definition S_k to use u, including paths that go through u several times (but do not go through S_k a second time), there are no assignments to the source variable y.

2.2.2.5.1 ***Formal Specification.***

```
$\exists S_i \exists S_j \exists opr : copy\_stmt(S_i) \wedge S_i \delta S_j \wedge opr \in source\_operands(S_j) \wedge S_i.dst = S_j.opr$
            /* copy_stmt($S_i$) ≡ $S_i$.opcode = COPY ∧ type($S_i$.dst) = VAR
               ∧ type($S_i$.src1) = VAR. */
            : $(\forall p : p \in Paths[S_{begin}, S_j]$
                : $(\exists S_k : S_k \in p \wedge S_k \delta S_j \wedge S_k.dst = S_j.opr$
                    : $copy\_stmt(S_k) \wedge S_k.src1 = S_i.src1$
                    $\wedge (\nexists S_l : S_l \in p : S_k \bar{\delta} S_l)))$
                      /* There are no redefinitions of the source operand of $S_k$
                         between $S_k$ and $S_j$ on path $p$. */
                      $\{replace\_operand(S_j.opr, S_i.src1);\}$
```

As in constant propagation, the preceding specification is less conservative compared with other specifications available in the literature [31]. This specification includes the case where more than one definition reaches the use, which is not considered by other specifications.

2.2.2.6 Common Subexpression Elimination

An occurrence of an expression in a program is a common subexpression if another occurrence of the expression exists whose evaluation always precedes this one in execution order and if the operands of the expression remain unchanged between these two executions. Common subexpression elimination (CSE) is a transformation that removes the recomputations of common subexpressions and replaces them with uses of saved values.

2.2.2.6.1 ***Formal Specification.***

```
$\exists S_i \exists S_j : S_i \neq S_j \wedge valid\_cse\_expr(S_i.expr) \wedge valid\_cse\_expr(S_j.expr)$
      /* valid_cse_expr() checks whether the expression              */
         is a possible candidate for CSE.
    : $S_i.expr = S_j.expr \wedge same\_control\_dep(S_i, S_j)$
      /* same_control_dep($S_i$,$S_j$) checks whether the set of statements to which $S_i$
         and $S_j$ are control dependent are the same and also that both $S_i$ and $S_j$
         occur on a single execution path. */
      $\wedge (\nexists p : p \in Paths[S_i, S_j] : (\exists S_l : S_l \in p : S_i \bar{\delta} S_l))$
      /* There are no redefinitions of the source operands of $S_i$ between $S_i$ and $S_j$. */
      {
        $replace\_operand(S_i.dst, new\_operand(h));$
        /* Replace the destination operand of $S_i$ by the new variable $h$. */
        $insert\_stmt(new\_stmt(COPY, S_i.dst, new\_operand(h)), S_i.next);$
        /* Insert a new statement $S_i.dst := h$ after statement $S_i$. */
        $replace\_stmt(S_j, new\_stmt(COPY, S_j.dst, new\_operand(h)));$
        /* Replace $S_j$ by $S_j.dst := h$. */
      }
```

where:

```
$same\_control\_dep(S_i, S_j) \equiv (\forall S_k : S_k \delta^c S_i : S_k \delta^c S_j) \wedge (\forall S_k : S_k \delta^c S_j : S_k \delta^c S_i)$
                               $\wedge\ same\_side\_of\_branch(S_i, S_j)$
                               /* $same\_side\_of\_branch(S_i, S_j)$ checks whether $S_i$ and $S_j$
                                  are on the same side of the branch instruction to which
                                  $S_i$ and $S_j$ are control dependent. */
```

Like other specifications in the literature, the preceding specification also is conservative (i.e., it does not capture all possible common subexpressions). The reason for the conservativeness is due to the difficulty in capturing the concept of available expressions in the dependence relations framework.

The specification can be easily modified to capture more than two common subexpressions existing on a single execution path simultaneously [19].

2.2.2.7 Invariant Code Motion

A computation in a loop is invariant if its value does not change within the loop. Loop invariant code motion recognizes invariant computations in loops and moves them to the loop preheader. For an invariant computation $S_i : x := y + z$, in a loop L, the following three conditions ensure that code motion does not change the semantics of the program:

1. The basic block containing S_i dominates all exit nodes of the loop, where an exit of a loop is a node with successor not in the loop.
2. No other statement is in the loop that assigns to x.
3. No use of x in the loop is reached by any definition of x from outside the loop.

2.2.2.7.1 *Formal Specification.*

```
$\exists S_i \exists L : S_i \in L.body$
    $: (\nexists S_j : S_j \in L.body$
        $: (S_i \bar{\delta} S_j \vee S_j \delta S_i) \vee$
          /* There are no redefinitions of the operands of $S_i$ in L. */
        $(S_j \neq S_i \wedge (S_i \delta^o S_j \vee S_j \delta^o S_i)) \vee$
          /* There are no redefinitions of the destination variable of $S_i$ in L. */
        $(S_j \bar{\delta} S_i \wedge (\exists S_k : S_k \notin L : S_k \delta S_j \wedge S_k \delta^o S_i)) \vee$
          /* No definition of the destination variable of $S_i$, from outside the loop,
             reaches any use of it in the loop. */
        $(S_j \delta^c S_i))$
          /* $S_i$ is not control dependent on any statement in the loop body. */
        $\{move\_stmt(S_i, L.pre);\}$
          /* Move statement $S_i$ to loop preheader. */
```

The specification given here is less conservative compared with other specifications in the literature [28, 31]; see the difference in the specification of the condition stating that no definition of the destination variable of S_i from outside the loop reaches any use of it in the loop.

2.2.2.8 Induction Variable Elimination

Induction variable elimination tries to get rid of all induction variables, except one, when two or more induction variables of a family are in a loop. Induction variable elimination may be enabled by strength reduction on induction variables; strength reduction replaces expensive operations, such as multiplication and division, by less expensive addition and subtraction operations. After strength reduction on induction variables the only use of some induction variables is in branch tests. We can replace a test of such an induction variable by that of another, making the original induction variable useless.

A variable x, in a loop L, is called an induction variable if every time the variable x changes its value, it is incremented or decremented by some constant. *Basic induction variables* (*BIVs*) are those variables x whose only assignments within the loop are of the form $x := x + c$, where c is a constant. *Other induction variables* (*OIVs*) are variables y defined only once within the loop and whose value is a linear function of some BIV or of some OIV.

Associated with each other induction variable y is a triple $\langle x, c, d \rangle$, where c and d are constants, and x is a basic induction variable such that the value of y is given by $c * x + d$. Each basic induction variable x has a triple $\langle x, 1, 0 \rangle$ associated with it.

The process of induction variable elimination may be divided into three subtasks:

1. Detection of induction variables
2. Strength reduction on induction variables
3. Elimination of induction variables

After detecting the induction variables, strength reduction is applied on these induction variables followed by induction variable elimination transformation.

We now give the formal specifications for induction variables, induction variable strength reduction transformation and induction variable elimination transformation:

2.2.2.8.1 ***Formal Specification for Induction Variables.***

```
$BIV(x, L) \equiv$ /* $x$ is a basic induction variable in loop L. */
  $\exists S_i : S_i \in L \wedge same\_symbol(S_i.dst.symbol, x) \wedge same\_symbol(S_i.src1.symbol, x)$
      $\wedge\ type(S_i.src2) = CONST \wedge S_i.opcode \in \{+, -\}$
    $: (\forall S_p : S_p \in L \wedge S_p \neq S_i \wedge (S_p \delta^o S_i \vee S_i \delta^o S_p)$
          $: same\_symbol(S_p.src1.symbol, x)$
            $\wedge\ type(S_p.src2) = CONST \wedge S_p.opcode \in \{+, -\})$

$OIV(z, x, L) \equiv$ /* $z$ is an other induction variable with $x$ as the BIV in loop L. */
  /* Case(a): $z$ is defined in terms of basic induction variable $x$. */
  $\exists S_k : S_k \in L \wedge same\_symbol(S_k.dst.symbol, z) \wedge \neg same\_symbol(z, x)$
      $\wedge\ same\_symbol(S_k.src1.symbol, x)$
      $\wedge\ type(S_k.src2) = CONST \wedge S_k.opcode \in \{+, -, *, /\}$
    $: (\nexists S_p : S_p \in L \wedge S_p \neq S_k : S_p \delta^o S_k \vee S_k \delta^o S_p)$
      /* There are no redefinitions of $z$ in L. */

 /* Case(b): $z$ is defined in terms of other induction variable $y$. */
 $\exists S_k \exists S_j : S_k \in L \wedge same\_symbol(S_k.dst.symbol, z) \wedge \neg same\_symbol(z, x)$
        $\wedge\ same\_symbol(S_j.dst.symbol, y) \wedge \neg same\_symbol(y, z) \wedge OIV(y, x, L)$
        $\wedge\ same\_symbol(S_k.src1.symbol, y) \wedge type(S_k.src2) = CONST$
        $\wedge\ S_k.opcode \in \{+, -, *, /\}$
      $: (\nexists S_p : S_p \in L \wedge S_p \neq S_k : S_p \delta^o S_k \vee S_k \delta^o S_p)$
        /* There are no redefinitions of $z$ in L. */
        $\wedge\ (\nexists S_q : S_q \notin L : S_q \delta S_k)$
         /* No definition of $y$ from outside L reaches $S_k$. */
        $\wedge\ (\nexists p : p \in Paths[S_j, S_k] : (\exists S_r : S_r \in p : same\_symbol(S_r.dst.symbol, x)))$
         /* There are no assignments to $x$ between $S_j$ and $S_k$. */
```

2.2.2.8.2 ***Formal Specification for Induction Variable Strength Reduction.***

```
$\exists x \exists L :: BIV(x, L)$
{
  $\forall z :: OIV(z, x, L)$
  /* Note that $OIV(z, x, L) \Rightarrow \exists S_k : S_k \in L : same\_symbol(S_k.dst.symbol, z)$.
     Let the triple associated with $z$ be $\langle x, c, d \rangle$. */
  {
    $replace\_stmt(S_k, new\_stmt(COPY, new\_operand(z), new\_operand(temp1)));$
     /* Replace assignment to $z$ by $z := temp1$, where $temp1$ is a new unique variable. */

    /* Immediately after each assignment $x := x + n$ in L, where $n$ is a constant, append
       $temp1 := temp1 + c * n$. The case $x := x - n$ has to be treated analogously. */
    $\forall S_l : S_l \in L$
        $: same\_symbol(S_l.dst.symbol, x) \wedge same\_symbol(S_l.src1.symbol, x)$
          $\wedge\ type(S_l.src2) = CONST \wedge S_l.opcode = ADD$
          $\{\ insert\_stmt(new\_stmt(ADD, new\_operand(temp1), new\_operand(temp1),$
                              $new\_operand(eval(MUL, c, S_l.src2))), S_l.next);\ \}$
    /* At this point, $temp1$ is placed in the family of $x$ with triple $\langle x, c, d\rangle$,
       and $z$ is removed from the family of $x$. */

    /* Initialize $temp1$ to $c * x + d$ in the loop preheader, $L.pre$. */
    $insert\_stmt(new\_stmt(MUL, new\_operand(temp1),$
                          $new\_operand(c), new\_operand(x)), L.pre);$
    $if\,(d \neq 0)$
      $insert\_stmt(new\_stmt(ADD, new\_operand(temp1),$
                          $new\_operand(temp1), new\_operand(d)), L.pre);$
  }
}
```

Note that insertion at $L.pre$ means insertion at the end of the preheader block of loop L.

2.2.2.8.3 ***Formal Specification for Induction Variable Elimination.***

```
    $\exists x \exists L : BIV(x, L)$
          $: (\forall S_l : S_l \in L \wedge (\exists S_l.src :: same\_symbol(S_l.src.symbol, x))$
                $: conditional\_branch\_stmt(S_l) \vee OIV(S_l.dst.symbol, x, L))$
            $\wedge\ (\exists S_m : S_m \in L \wedge (\exists S_m.src :: same\_symbol(S_m.src.symbol, x))$
                  $: conditional\_branch\_stmt(S_m))$
            $\wedge\ (\exists z :: OIV(z, x, L))$
              /* Let the triple associated with $z$ be $\langle x, c, d\rangle$.
                 The specification given here assumes $c$ to be positive. */
    {
      /* Replace each statement of the form, "if $x$ relop $w$ goto L",
         where, $w$ is not an induction variable by
         " $temp := c * x;\ temp := temp + d;\ if\ (z\ relop\ temp)\ goto\ L;$",
         where, $temp$ is a new unique variable.
         The case, "$if\ (w\ relop\ x)\ goto\ L$" has to be treated analogously.
         The special case, "$if\ (x_1\ relop\ x_2)\ goto\ L$",
         where, both $x_1$ and $x_2$ are basic induction variables is not considered here. */
      $\forall S_t : S_t \in L \wedge conditional\_branch\_stmt(S_t)$
          $: same\_symbol(S_t.src1.symbol, x) \wedge \neg induction\_variable(S_t.src2, L)$
          /* where, induction_variable $\equiv$ BIV $\vee$ OIV. */
      {
        $insert\_stmt(new\_stmt(MUL, new\_operand(temp),$
                          $new\_operand(c), new\_operand(S_t.src2)), S_t.prev);$
        $if\,(d \neq 0)$
          $insert\_stmt(new\_stmt(ADD, new\_operand(temp),$
                          $new\_operand(temp), new\_operand(d)), S_t.prev);$
        $replace\_stmt(S_t, new\_stmt(\ S_t.opcode, new\_operand(S_t.dst),$
                              $new\_operand(z), new\_operand(temp)));$
      }
    }
```

2.2.2.9 Loop Interchange

Loop interchange exchange the position of two loops in a tightly nested loop. Interchange is one of the most powerful transformations and can improve performance in many ways.

For example, loop interchange may be performed to enable vectorization by interchanging an inner, dependent loop with an outer, independent loop.

The requirements for loop interchanging can be stated as:

1. The loops L_1 and L_2 must be tightly nested (L_1 surrounds L_2, but contains no other executable statements).
2. The loop limits of L_2 are invariant in L_1.
3. There are no statements S_i and S_j (not necessarily distinct) in L_2 with a dependence relation $S_i \delta^*_{(\langle,\rangle)} S_j$.

Also, if input or output (I/O) statements exist, then interchanging the loops can change the order in which the I/O occurs. Even though this may be allowable in some cases, in general it is not allowable for the compiler to change the order of I/O operations of a program.

2.2.2.9.1 *Formal Specification.*

```
$\exists L_1 \exists L_2 : tightly\_nested\_loops(L_1, L_2)$
        /* $L_1$ surrounds $L_2$, but contains no other executable statements. */
        $\wedge (\nexists S_i : S_i \in L_2.body : io\_stmt(S_i))$
        /* There are no I/O statements in the loop. */
        $: \neg(\exists S_j \exists S_k : S_j \in L_1.head \wedge S_k \in L_2.head : S_j \delta S_k \vee S_j \bar{\delta} S_k \vee S_j \delta^o S_k)$
        /* Loop limits of $L_2$ are invariant in $L_1$. */
        $\wedge \neg(\exists S_m \exists S_n : S_m \in L_2.body \wedge S_n \in L_2.body$
                $: S_m \delta_{(\langle,\rangle)} S_n \vee S_m \bar{\delta}_{(\langle,\rangle)} S_n \vee S_m \delta^o_{(\langle,\rangle)} S_n)$
        /* There are no dependences with direction vector $(\langle,\rangle)$ in the loop. */
        $\{move(L_2.head, L_1.head.prev);\}$
        /* Interchange heads of loops $L_1$ and $L_2$. */
```

2.2.2.10 Loop Fusion

Loop fusion transformation fuses two adjacent loops into one. Reducing the loop overhead and increasing instruction parallelism are some of the performance improvements that can be achieved by loop fusion.

The requirements for fusing two adjacent loops L_1 and L_2 are:

1. The loop control variables should be the same and the loop limits must be identical.
2. There should not be any statement S_v in L_1 and S_w in L_2 with a dependence relation $S_v \delta^*_{(\rangle)} S_w$.
3. Both loops do not have a conditional branch that exits the loop.
4. Loops L_1 and L_2 both should not have I/O statements.

Note that in condition 2 we have used data dependence direction vector augmented to add a direction for the adjacent loop [26].

2.2.2.10.1 *Formal Specification.*

```
$\exists L_1 \exists L_2 : adjacent\_loops(L_1, L_2)$
        /* There are no statements between the end of $L_1$ and the header of $L_2$. */
        $\wedge\ identical\_loop\_headers(L_1, L_2)$
        /* Loop control variables are the same and the loop bounds are identical. */
        $\wedge\ \neg(\exists S_i : S_i \in L_1 : conditional\_branch(S_i) \wedge branch\_target(S_i) \notin L_1)$
        /* There is no conditional branch statement in $L_1$ which exits the loop. */
```

```
    $\wedge \neg\ (\exists S_j : S_j \in L_2 : conditional\_branch(S_j) \wedge branch\_target(S_j) \notin L_2)$
    /* There is no conditional branch statement in $L_2$ which exits the loop. */
    $\wedge \neg\ (\exists S_m \exists S_n : S_m \in L_1 \wedge S_n \in L_2 : io\_stmt(S_m) \wedge io\_stmt(S_n))$
    /* Loops $L_1$ and $L_2$ both do not have I/O statements. */
  $: \neg(\exists S_v \exists S_w : S_v \in L_1 \wedge S_w \in L_2 : S_v \delta_{(())} S_w \vee S_v \bar{\delta}_{(())} S_w \vee S_v \delta^o_{(())} S_w)$
    /* There are no statements $S_v$ and $S_w$ such that $S_v \in L_1$ and $S_w \in L_2$ which will
       cause a dependence $S_v \delta^*_{(()} S_w$ in the fused loop. */
    $\{delete(L_1.end);\ delete(L_2.head);\ \}$
    /* Fuse loops $L_1$ and $L_2$. */
```

2.2.2.11 Loop Reversal

Reversal changes the direction in which the loop traverses its iteration range. It is often used in conjunction with other iteration space reordering transformations because it changes the dependence vectors; for example, loop reversal may enable loop interchange.

If loop p in a nest of d loops is reversed, then for each dependence vector V, the entry v_p is negated. The reversal is legal if each resulting vector V' is lexicographically positive, i.e., when $v_p = 0$ or $\exists q \langle p : v_q \rangle 0$.

2.2.2.11.1 *Formal Specification.*

```
$\exists NL \exists L_i : L_i \in NL$
        /* $L_i$ is a loop in the loop nest $NL$. */
      $: (\forall V : V \in reversed\_direction\_vectors(NL, L_i)$
            /* $V$ is a direction vector in $NL$ with $L_i$ reversed. */
          $: lexicographically\_positive(V))$
          /* The first nonzero element of the direction vector is positive. */
          {
            $swap(L_i.initial, L_i.final);$
            /* Interchange the initial and final iteration values of loop $L_i$. */
            $replace(L_i.step, eval(-L_i.step));$
            /* Reverse the direction of iteration of loop $L_i$. */
          }
```

2.2.2.12 Loop Skewing

Loop skewing is an enabling transformation that is primarily useful in combination with loop interchange. Skewing handles wavefront computations where updates to the array propagates like a wave across the iteration space.

Skewing is performed by adding the outer loop index multiplied by a skew factor, f, to the bounds of the inner iteration variable, and then subtracting the same quantity from every use of the inner iteration variable inside the loop. Because it alters the loop bounds but then alters the uses of the corresponding index variables to compensate, skewing does not change the meaning of the program and is always legal.

2.2.2.12.1 *Formal Specification.*

```
$\exists L_1 \exists L_2 : tightly\_nested\_loops(L_1, L_2) \wedge forward\_loop(L_1) \wedge forward\_loop(L_2)$
    /* $L_1$ surround $L_2$, but contains no other executable statements.
       Both $L_1$ and $L_2$ traverses the iteration range in the forward direction;
       i.e., Both loops have positive step values. */
    {
      $replace(L_2.initial, eval(L_2.initial + f * L_1.lcv));$
      /* Add loop index of $L_1$ multiplied by skew factor $f$ to the lower bound
```

```
        of the iteration variable of $L_2$. */
    $replace(L_2.final,\ eval(L_2.final + f * L_1.lcv));$
    /* Add loop index of $L_1$ multiplied by skew factor $f$ to the upper bound
        of the iteration variable of $L_2$. */
    $\forall x : x \in L_2.body : x = L_2.lcv$
        $\{\ replace(x,\ eval(x - f * L_1.lcv));\ \}$
}
```

In this work most of the scalar transformations and also some of the parallel transformations have been specified in the dependence relations framework. Each specification has two parts — a precondition part and an action part. Whereas the precondition part specifies the condition that has to be satisfied for the application of the transformation, the action part specifies the steps to perform the transformation using some primitive actions. The work assumes an intermediate representation that supports both low-level and high-level transformations. We did not succeed in specifying some of the transformations in the framework used, for instance, partial redundancy elimination. Further study is required on the limitations of the current framework to come up with better alternatives for the specification of code transformations.

2.3 Specification Language

We need a specification language to express formal specifications of code transformations; these specifications, expressed in the specification language, are used to generate code for the corresponding transformations. Ideally, one would like to have a language in which all code transformations can be specified. Whitfield and Soffa [31] have designed a specification language, called Gospel, in which they have specified many of the traditional and parallelizing transformations, using dependence relations.

The specification language we describe here is powerful enough to specify all traditional scalar transformations in the dependence relations framework. Each specification has a precondition part and an action part. The precondition part, which checks the conditions to be satisfied for the application of the transformation, is expressed in a notation similar to predicate logic. The action part, which specifies the transformation to be performed, is expressed using a set of primitive actions provided by the language. These primitive actions assume an intermediate representation of the program to be transformed, in three-address code form.

The specifications for all traditional scalar transformations have been expressed in this language. The transformations expressed in the specification language are constant propagation (CTP), constant folding (CFD), useless code elimination (UCE), unreachable code elimination (URCE), copy propagation (CPP), common subexpression elimination (CSE), invariant code motion (ICM) and induction variable elimination (IVE). The language is more powerful, compared to Gospel, for the class of transformations for which it is intended (i.e., scalar transformations). The expressive power of the language is demonstrated by the specification of a complex transformation, such as induction variable elimination. The language permits modularity in expressing complex specifications. The language design is restricted to the class of scalar transformations, but permits extensions to the language, if needed.

2.3.1 Description of the Language

The language is designed with the intention of specifying all scalar transformations, having dependence relations as the basis of the specification. The program elements used in the specification are operands, statements, loops and basic blocks. A program, which has to be transformed, is represented as a CFG in which each node is a basic block. Each basic black contains a sequence of statements without a branch, except for the last statement. A loop in the program is represented as a sequence of basic blocks. The specification of a transformation assumes an intermediate representation in three-address code form on which the transformations are applied. In general, a statement in three-address code form has a destination operand, two source operands and an opcode: $dst := scr1\ op\ src2$.

In the following part of this section we describe the major syntactic categories of the language: *specification*, *subspecification*, *conditional_expr*, *action* and *main_specification*. The complete syntax of the language is given in Appendix 2.A.

2.3.1.1 Complete Specification

The complete specification for a transformation consists of a sequence of optional subspecifications followed by the main specification, the production for which is given as follows:

$$\begin{array}{l} \mathit{specification} :: \\ \quad \mathit{sub\_specification\_list}_{opt}\ \mathit{main\_specification} \end{array}$$

The subspecification facility can be of help when writing complex specifications; see the specification for constant propagation in Section 2.3.3.

2.3.1.2 Subspecification

The subspecification can be either a condition that has to hold for the application of the transformation or an action for transforming the code:

$$\begin{array}{l} \mathit{sub\_specification} :: \\ \quad \mathit{condition\_sub\_specification} \\ \quad \mathit{action\_sub\_specification} \end{array}$$

Each subspecification has a header and a body; the header contains a name for identification and a list of parameters used in the subspecification. Examples of parameters include program elements such as operands and statements. The body of condition subspecification is a predicate and the body of action subspecification is a list of actions to transform the code:

$$\begin{array}{l} \mathit{condition\_sub\_specification} :: \\ \quad \mathit{sub\_specification\_header}\ \mathit{condition\_predicate} \\ \mathit{action\_sub\_specification} :: \\ \quad \mathit{sub\_specification\_header}\ \{\mathit{action\_list}\} \\ \mathit{sub\_specification\_header} :: \\ \quad \mathit{identifier}(\mathit{parameter\_list}) : \end{array}$$

The syntactic category *condition_predicate* is a subpart of the precondition in the main specification and is expressed in a notation similar to predicate logic: (Q:R:P) where Q is the quantifier, R the range of the quantified element and P the predicate. The elements that can be quantified are the program elements — operands, statements, loops and basic blocks. The range part identifies the program elements satisfying the conditions mentioned and the predicate part checks whether the selected program elements satisfy the conditions for applying the transformation:

$$\begin{array}{l} \mathit{condition\_predicate} :: \\ \quad (\mathit{quantifier} : \mathit{conditional\_expr} : \mathit{conditional\_expr}) \\ \mathit{quantifier} :: \\ \quad \mathit{quantifier\_type}\ \mathit{program\_element} \end{array}$$

$$
\begin{array}{l}
\textit{quantifier\_type} :: \textit{one of} \\
\quad \textit{FORALL EXISTS NOT\_EXISTS} \\
\textit{program\_element} :: \textit{one of} \\
\quad \textit{OPERAND STMT LOOP NODE}
\end{array}
$$

When alternative categories are listed on one line, it is marked by the phrase "one of." Words in upper case letters represent terminals in the grammar. *FORALL* and *EXISTS* represent the universal and existential quantifiers and *NOT_EXISTS* represents the negation of existential quantifier. The lexemes corresponding to terminals *FORALL*, *EXISTS* and *NOT_EXISTS* are the same character sequence in those words, in lowercase letters.

2.3.1.3 Conditional Expressions

Conditional expressions are built from basic conditions using Boolean operators *not*, *and* and *or*:

$$
\begin{array}{l}
\textit{conditional\_expr} :: \\
\quad \textit{condition} \\
\quad \textit{NOT conditional\_expr} \\
\quad \textit{conditional\_expr AND conditional\_expr} \\
\quad \textit{conditional\_expr OR conditional\_expr}
\end{array}
$$

Basic conditions include checks for dependence relations between pairs of statements. The following productions give some examples:

$$
\begin{array}{l}
\textit{conditions} :: \\
\quad \textit{FLOW} \; (\textit{STMT}, \textit{STMT}) \\
\quad \textit{CTRL} \; (\textit{NODE}, \textit{NODE})
\end{array}
$$

The condition *FLOW*($STMT$, $STMT$) checks for a flow dependence between the two statements and *CTRL*($NODE$, $NODE$) checks for a control dependence between the two nodes (basic blocks). The terminals representing the dependences, such as *FLOW*, have their lexemes as the same character sequence in those words, in lowercase letters. For example, a specification may contain a condition, such as *flow (stmt_i, stmt_j)*.

Apart from the conditions involving dependence relations, the language also provides some general conditions useful for the precondition part. For example, consider the following conditions:

$$
\begin{array}{l}
\textit{condition} :: \\
\quad \textit{VAR\_OPERAND} \; (\textit{OPERAND}) \\
\quad \textit{COND\_BRANCH} \; (\textit{STMT})
\end{array}
$$

VAR_OPERAND checks whether the given operand is a variable, and *COND_BRANCH* checks whether the given statement is a conditional branch statement.

Provision to add new conditions is made through the production:

$$
\begin{array}{l}
\textit{condition} :: \\
\quad \textit{identifier} \; (\textit{parameter\_list})
\end{array}
$$

By using this provision, one can write new conditions within the constraints of parameters permitted by the language, and can use these conditions in a specification.

2.3.1.4 Primitive Actions

The language provides a set of primitive actions in which all code transformations are specified. For example, consider the following primitive actions provided by the language:

$$\begin{array}{l} action :: \\ \quad DELETE\_STMT\ (STMT); \\ \quad CREATE\_NEW\_STMT\ (STMT\_CLASS, \mathit{parameter\_list}, NEW\_STMT); \\ \quad INSERT\_STMT\ (NEW\_STMT, POSITION); \\ \quad CREATE\_NEW\_OPERAND\ (\mathit{parameter}, NEW\_OPERAND); \\ \quad REPLACE\_OPERAND\ (OPERAND, \mathit{parameter}); \end{array}$$

DELETE_STMT deletes the specified statement from the program. *CREATE_NEW_STMT* returns a new statement knowing the statement class and using the parameters provided in the parameter list. *STMT_CLASS* represents a different class of statements, like branch or jump statements. The parameters in the parameter list are the opcode and operands of the statement. *INSERT_STMT* inserts a newly created statement at the indicated position in the program. The position in a program is indicated with reference to a statement — either before or after a statement. For example, $stmt\_i.next$ and $stmt\_i.prev$ indicate the positions after and before statement i, respectively. Another position used in the language is the loop preheader; for example, $loop\_i.pre$ represents the preheader of loop i. *CREATE_NEW_OPERAND* creates a new operand from the parameter given; the parameter can be a new symbol, an existing operand symbol, an existing operand, an existing statement or a new statement. *REPLACE_OPERAND* replaces an operand of a statement by the parameter supplied; the parameter can be either another existing operand or a newly created operand.

Facility is also provided to add new actions through the production:

$$\begin{array}{l} action :: \\ \quad identifier\ (parameter\_list); \end{array}$$

The provision allows one to write new actions, with the parameters permitted by the language, and to use them in a specification.

2.3.1.5 Main Specification

The main specification of a transformation constitutes a name to identify the transformation followed by the precondition and action parts:

$$\begin{array}{l} main\_specification :: \\ \quad transformation\_name : action\_predicate \\ action\_predicate :: \\ \quad [action\_quantifier : conditional\_expr : conditional\_expr\{action\_list\}] \\ action\_quantifier :: \\ \quad FOREACH\ program\_element \end{array}$$

The syntactic category *action_predicate* identifies all program elements specified by *FOREACH program_element*, satisfying the conditions mentioned in the range part — specified by the first *conditional_expr*, checks the conditions mentioned in the predicate part — specified by the second *conditional_expr* and makes the transformations mentioned in the action part if the predicate is true.

Experiments with new specifications can be done by the provision made in the language with the production:

$$\begin{array}{l} \mathit{transformation\_name} :: \\ \quad \mathit{NEW} \end{array}$$

This provision allows the user to write new specifications with the transformation name, *new_transformation*.

2.3.2 Lexical Conventions

The specification language has tokens like identifiers, keywords, constants, operators and punctuation symbols [19]. White spaces like blanks, tabs and new lines are ignored except for the case when they separate tokens. Comments can appear inside the specification between characters /* and */.

An identifier is a sequence of letters and digits with a letter at the beginning; the underscore counts as a letter. Many identifiers are reserved for use as keywords and may not be used otherwise. The keywords include names used for denoting dependences — such as *flow*, names used for quantifiers — such as *forall*, names used for conditions — such as *var_operand*, names used for actions — such as *delete_stmt*, and names used for transformations — such as *constant_propagation*. Only integer constants are permitted and each integer constant is a sequence of digits. The operators used are relational operators and Boolean operators. The punctuation symbols include comma, colon and semicolon.

The token *OPERAND* can represent any operand of a statement. *STMT_CLASS* represents the class of statements; the same classes as in Stanford University Intermediate Format (SUIF) have been used: *ldc* — the load constant statements, *rrr* — three address statements and *bj* — branch and jump statements. The token *NULL_VALUE* — with lexeme null — is used in place of a parameter when it is not required.

2.3.3 Example: Specification for Constant Propagation in the Language

All traditional scalar transformations have been expressed in the specification language [19]. Here, we give an example of a specification, for constant propagation, expressed in the language.

. . . .

Subspecification to check whether all definitions of the variable reaching the point of use have same constant values. If more than one definition reaches the point of use, then all definitions must have the same constant value for applying constant propagation.

. . . .

```
same_const_value_from_all_candidate_stmts(stmt_i, stmt_j, stmt_j.src) :
( forall stmt_k : flow(stmt_k, stmt_j)
                      and not (same_stmt(stmt_k, stmt_i))
                      and same_operand_symbol(stmt_k.dst, stmt_j.src)
                     : same_expr(stmt_k.expr, stmt_i.expr))
```

. . . .

Main specification for constant propagation. Condition “constant_defn(stmt_i)” checks whether stmt_i is a constant definitions statement (i.e., statement of the form $x := 5$).

. . . .

```
constant_propagation :
[ foreach stmt_i : constant_defn(stmt_i) :
  {
    [ foreach stmt_j : flow(stmt_i, stmt_j) :
      {
        [ foreach stmt_j.src : same_operand_symbol(stmt_i.dst, stmt_j.src)
            : same_const_value_from_all_candidate_stmts(stmt_i, stmt_j, stmt_j.src)
          {
            replace_operand(stmt_j.src, stmt_i.src1);
          }
        ]
      }
    ]
  }
]
```

2.4 Automatic Generation of Code Optimizers

We describe the transformation system to generate code for scalar optimizers from their specifications. The system provides a complete environment for code transformations where one can specify a transformation in the specification language, generate code for it and experiment the effect of the transformation on real-world programs. Code for all traditional scalar transformations have been generated, and those transformers were applied on LINPACK benchmark programs. The use of an intermediate representation such as SUIF makes the system portable and extendable.

2.4.1 Code Transformation System

The system provides a complete environment for code transformations where one can specify a transformation in the specification language, generate code for it and experiment with the transformer generated on real-world programs. The system is built using the infrastructure provided by SUIF [25]. Figure 2.1 gives the organization of the system.

The transformer generator takes specifications — written in the specification language — as input, analyzes it using Lex and Yacc and generates code — the transformer — for the specified transformations. The generated code uses the transformer library, which contains a collection of routines for checking the preconditions and performing the actions for different transformations. The program to be transformed, in C or FORTRAN, is first converted to the intermediate form by the component HLL-to-SUIF, supported by SUIF. The dependence analyzer analyzes the code in the intermediate form and annotates it with control and data dependences. The code transformer interface then applies the transformation, requested by the user, on the intermediate code using the transformer generated from the specification. The transformer uses the annotations on the intermediate code for dependence checks. Finally, the transformed intermediate code is converted back to high level by the component SUIF-to-HLL, again a function supported by SUIF. The transformer generator, transformer library, dependence analyzer and code transformer interface (blocks drawn using bold lines in the figure) are the components developed in this work, on top of SUIF, to build the transformation system.

2.4.1.1 Stanford University Intermediate Format

SUIF consists of a small kernel and a tool kit of compiler passes built on top of the kernel [25]. The kernel defines the intermediate representation, provides functions to access and manipulate the intermediate representation, and structures the interface between compiler passes [23]. The tool

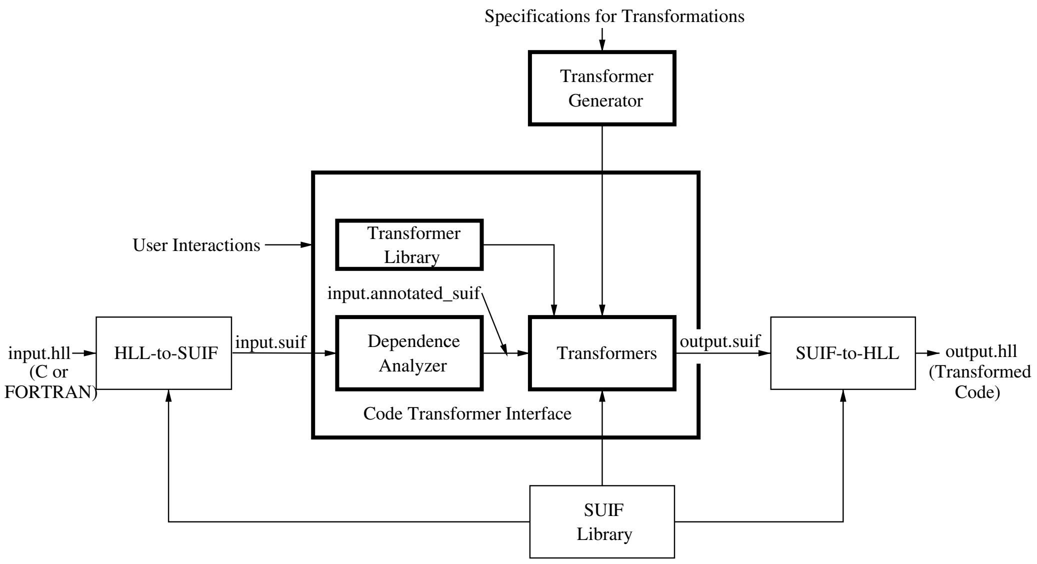


FIGURE 2.1 The code transformation system.

kit includes C and FORTRAN front ends, an optimizing MIPS back end and a set of compiler development tools.

The intermediate representation supports both high-level and low-level information. The high-level structure is represented by a language-independent form of abstract syntax trees and the low-level structure by sequential lists of instructions. The low-level structure is better suited for scalar transformations and the high-level structure is required for parallel transformations.

Low-level instruction in SUIF performs a single operation specified by its opcode. With a few exceptions, the opcodes are simple and straightforward, similar to the instruction set of a typical reduced instruction set computing (RISC) processor. The instructions include arithmetic, data transfer and control flow operations; and are divided into several classes, including three-operand instructions, branch and jump instructions, load constant instructions, call instructions and array instructions. Most SUIF instructions use a quadruple format with a destination operand and two source operands. However, some instructions require more specialized formats. For example, load constant instructions have an immediate value field in place of the source operands.

The symbol tables in a SUIF program hold detailed symbol and type information. A symbol table is attached to each element of the hierarchical structure of SUIF — such as procedures and blocks — that defines a new scope. Symbols record information about variables, labels and procedures. The symbol tables are organized in a tree structure that form a hierarchy parallel to the SUIF hierarchy.

SUIF tool kit consists of a set of compiler passes implemented as separate programs. Each pass typically performs a single analysis or transformation and then writes the results out to a file. Compiler passes interact with one another either by updating the SUIF representation directly or by adding annotations to various elements of the program.

2.4.1.2 Transformer Generator

The transformer generator takes specifications for code transformations — written in the specification language — as input, analyzes it using Lex and Yacc and generates C language code for performing those transformations. The generated code — the transformer — relies on the transformer library, which contains routines for all conditions and actions supported by the specification language, for checking the precondition and performing the actions corresponding to the transformation. The transformer is generated under the assumption that it can be applied on low-level SUIF intermediate representation.

Lexical and syntax analysis of the specification are done by Lex and Yacc using the lexical and syntactical specifications of the language. The method of syntax-directed translation has been used for generating the transformer; semantic actions are associated with the productions of the grammar, which emits the code for the transformer. Generated code for each transformer is put into separate files, and the code transformer interface uses them appropriately.

The transformer generator first generates code for identifying the program elements of interest such as loops and instructions, followed by code to check the conditions to be satisfied for the application of the transformation, using the precondition part of the specifications. The action part of the specification is used next to generate code that can perform the corresponding transformation.

In the productions for the main specification:

$$\begin{array}{l} \textit{main\_specification} :: \\ \quad \textit{transformation\_name} : \textit{action\_predicate} \\ \textit{action\_predicate} :: \\ \quad [\textit{action\_quantifier} : \textit{conditional\_expr}_1 : \textit{conditional\_expr}_2 \{\textit{action\_list}\}] \end{array}$$

the term *action_quantifier : conditional_expr*$_1$ *: conditional_expr*$_2$ forms the precondition part and the term *action_list* forms the action part.

The semantic actions associated with the production for the syntactic component *action_quantifier* generate a loop construct for iteration over the instances of the program element in the program to be transformed:

```
action_quantifier ::
  FOREACH program_element
    {
      Semantic actions to generate a loop construct for
      iteration over the instances of the program element.
    }
```

As an example, if the program element is a statement, then the semantic actions can emit code for a loop construct to iterate over the list of instructions of the program.

The syntactic component *conditional_expr*$_1$, in the precondition part, is used to generate code for checking the conditions to select the instances of the program element of interest. Semantic actions associated with the productions for *conditional_expr* generates code for complex conditional expressions. For example, semantic actions associated with the production:

$$\begin{array}{l} \textit{conditional\_expr} :: \\ \quad \textit{conditional\_expr}_a \ \textit{AND} \ \textit{conditional\_expr}_b \end{array}$$

generate code for *conditional_expr* using the code for *conditional_expr*$_a$ and *conditional_expr*$_b$ with the Boolean operator *and* in between. The entire code generated for *conditional_expr*$_1$, in the precondition part, is enclosed in parentheses, preceded by an "if," to get the code form:

if('code for $\textit{conditional\_expr}_1$')

Semantic actions associated with the syntactic component *conditional_expr*$_2$, in the precondition part, generate code to check the conditions that have to be satisfied for the application of the transformation; the generation of code is done in the same way as in the case of *conditional_expr*$_1$.

The action part of the specification constitutes a sequence of actions supported by the specification language. Semantic actions associated with the production for each action can emit code for performing the corresponding function. For example, the action *DELETE_STMT* can emit code to delete the specified statement from the program:

```
action ::
  DELETE_STMT (STMT);
  {
    Semantic actions to generate code for deleting the specified statement.
  }
```

The structure of the code generated for the syntactic component *action_predicate* (i.e., *action_quantifier : conditional_expr*$_1$ *: conditional_expr*$_2$ $\{$*action_list*$\}$) is of the form:

```
while(! end_of_program_element_instances)
{
    if('code for conditional_expr1')
      if('code for conditional_expr2')
        {'code for action_list'}
}
```

For example, consider the specification for useless code elimination:

```
useless_code_elimination :
[
  foreach stmt_i :: useless_stmt (stmt_i)
    {delete_stmt(stmt_i); }
]
```

The code generated by the transformer generator for the preceding specification is given next, with some finer details removed:

```
void useless_code_elimination(...)
{...
  while(! end_of_instruction_list)
  {...
    stmt_i = next_instruction;
    if(useless_stmt(stmt_i))
      {delete_stmt(stmt_i); }
  }
}
```

Note that the condition *useless_stmt* and the action *delete_stmt* are routines provided by the transformer library. The construct *while(!end_of_instruction_list)* is the loop header of the loop generated for iteration over the list of instructions of the program.

By using the transformer generator, codes for all traditional scalar transformations have been generated. The major effort for implementing the code generator was for creation of the transformer library, which contains routines for all conditions and actions supported by the specification language. The current implementation did not take efficiency as a major concern and, hence, possibilities exist for improvement in this regard.

2.4.1.3 Dependence Analyzer

The dependence analyzer takes the SUIF intermediate representation of the program as input, does control and data dependence analyses on it and gives annotated intermediate code — intermediate code with control and data dependence information attached — as output. Computations of control dependence information involve construction of an augmented control flow graph (ACFG), reversal of ACFG to get the reverse augmented control flow graph (RACFG), computation of dominators of the RACFG and computation of dominance frontiers [6, 27] of each node in the RACFG [6]. Data dependences — flow, anti, output and input — are computed by the conventional iterative data flow analysis method [1]. Local information for each basic block is computed first, and then it is propagated over the CFG to get the global information at each basic block — through an iterative procedure. Finally, this global information is used to compute the data dependences for each instruction in a basic block.

2.4.1.4 Computation of Control Dependence

Control dependences of a node are essentially the dominance frontiers of that node in the reverse CFG [6]. In a reverse CFG, node n is control dependent on node m if and only if m is an element of the dominance frontier of n, that is:

$$m\delta^c n \;\equiv\; m \;\in\; df(n)$$

where, $df(n)$ is the dominance frontier of n, in the reverse CFG. The formal definition of control dependence is given in Section 2.2.1.1.

The algorithm to compute control dependence in a program is given as follows [6]:

```
begin
    construct the control flow graph, CFG;
    construct the augmented control flow graph, ACFG;
    construct the reverse augmented control flow graph, RACFG;
    compute dominators of each node in RACFG;
    compute the dominance frontier of each node in RACFG;
    compute control dependences, from the dominance frontier information;
end
```

CFG is a directed graph whose nodes represent the basic blocks of a program, and edges the flow of control. A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without a branch, except at the end. First, the statements in the intermediate representation of the given program are partioned into basic blocks [1]. The CFG is built using these basic blocks as nodes, and the flow of control as edges. Figure 2.2 shows a CFG as an example.

The ACFG is constructed by adding two additional nodes Entry and Exit to the CFG, such that there is an edge from Entry to any basic block at which the program can be entered, and there is an edge to Exit from any basic block that is an exit point for the program. An edge from Entry to Exit is added as part of the requirement for the computation of control dependences [6]; see Figure 2.3.

From ACFG the RACFG can be constructed with the same set of vertices, but with all arcs reversed. Note that the Entry and Exit nodes of ACFG become the Exit and Entry nodes of the RACFG respectively; see Figure 2.4.

A node m, in an augmented CFG, dominates a node n if every path from Entry to n includes m. The dominators of m include m and any node that also dominates all its predecessors [27], that is:

$$dom(m) \;=\; \{m\} \bigcup \bigcap_{p \in pred(m)} dom(p)$$

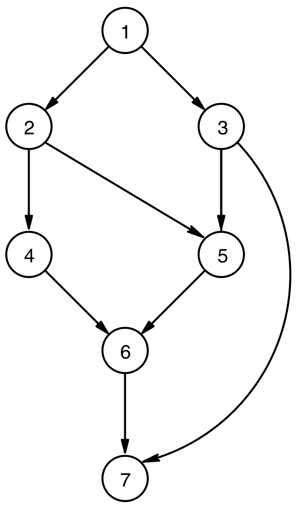


FIGURE 2.2 A control flow graph.

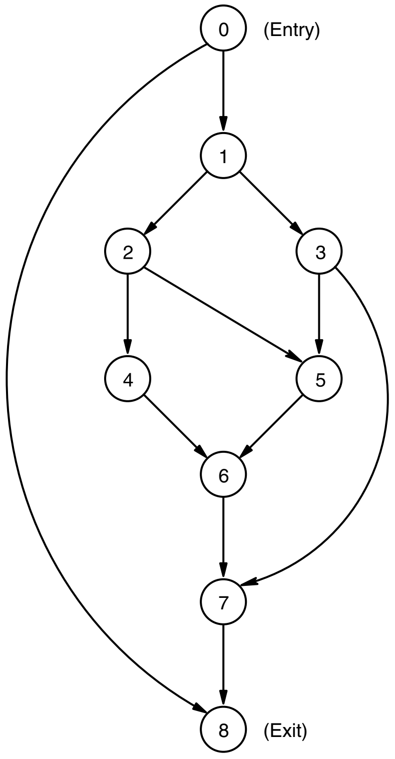


FIGURE 2.3 Augmented control flow graph.

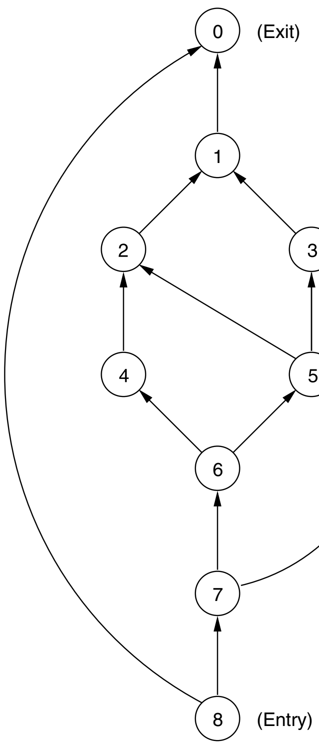


FIGURE 2.4 Reverse augmented control flow graph.

Based on the preceding relation, dominators for all nodes of RACFG are computed by the following algorithm:

```
begin
   dom(Entry) = $\{$Entry$\}$ ;
   for $x \in V - \{$Entry$\}$ do /* V is the set of vertices of RACFG */
      dom(x) = V;
   changed = TRUE;
   while(changed) do
   $\{$
      changed = FALSE;
      for $x \in V - \{$Entry$\}$ do
         old_dom = dom(x);
               dom(x) = $\{$x$\} \bigcup \bigcap_{p \in pred(x)}$ dom(p);
         if(dom(x) $\neq$ old_dom)
            changed = TRUE;
   $\}$
end
```

The dominators computed for the example in Figure 2.2 are given in Table 2.1.

The dominance frontier of a node *m* is the set of nodes *r* such that *m* dominates some predecessor of *r*, but not all, that is:

$$df(m) = \{r \mid \exists p, q \in pred(r)[m\ dom\ p \wedge m\ \overline{dom}\ q]\}$$

TABLE 2.1 Computation of Control Dependence

Node in RACFG n	Dominators of n $dom(n)$	Dominance frontiers of n $df(n)$	Nodes Control Dependent on n $cd(n)$
0	$\{0, 8\}$	ϕ	—
1	$\{1, 7, 8\}$	$\{0\}$	$\{2, 3, 6\}$
2	$\{2, 6, 7, 8\}$	$\{1\}$	$\{4, 5\}$
3	$\{3, 7, 8\}$	$\{1\}$	$\{5, 6\}$
4	$\{4, 6, 7, 8\}$	$\{2\}$	ϕ
5	$\{5, 6, 7, 8\}$	$\{2, 3\}$	ϕ
6	$\{6, 7, 8\}$	$\{1, 3\}$	ϕ
7	$\{7, 8\}$	$\{0\}$	ϕ
8	$\{8\}$	ϕ	—

With the dominator information available, dominance frontiers of all nodes in RACFG are computed, using the preceding relation; see Table 2.1.

In a reverse CFG, node n is control dependent on node m if and only if $m \in df(n)$ [6, 27]. Based on this knowledge, the control dependences for all nodes in RACFG is computed using the algorithm given next; see Table 2.1.

```
begin
  for each node m do
    cd(m) = $\phi$;  /* cd(m) represents nodes that are control dependent on m */
  for each node n do
    for each $m \in df(n)$ do
      cd(m) = cd(m) $\bigcup$ {n};
end
```

Note that, $df(n)$ in the preceding algorithm represents the dominance frontier of node n in RACFG.

The procedure outlined earlier computes the control dependences for each basic block in a program. The control dependence information computed is marked as annotations on instructions in the intermediate code. In the implementation, the information is attached to instructions that are leaders of basic blocks; the first instruction in a basic block is the leader of the basic block. When dealing with control dependences between instructions, instead of nodes, the control dependence information attached to the leader of a basic block is considered to be the representation for all instructions in the basic block. The specification language supports control dependence checks between either nodes or instructions.

2.4.1.5 Computation of Data Dependences

Data dependence analysis involves identification of all data dependences — flow, anti, output and input, as well as the direction vectors, between statements in a program. Because the implementation is restricted to scalar transformations, data dependences are computed without the computation of direction vectors; direction vectors are required, in general, for performing parallel transformations. Computation of data dependences is done by the conventional iterative data flow analysis method [1]. Data dependence computation problems have been formulated as data flow analysis problems by setting up the corresponding data flow equations. The analysis is a conservative approximation to the actual dependence relations adding all possible relations that might occur during execution. The implementation adopts this conservative approach to deal with pointers and procedure calls, without performing a detailed analysis in their presence.

Data flow information can be collected by setting up and solving systems of equations that relate information at various points in a program. A typical equation has the form:

$$out[s] \ = \ gen[s] \bigcup (in[s] - kill[s])$$

and can be read as, "the information at the end of a statement is either generated within the statement, or enters at the beginning and is not killed as control flows through the statement." Such equations are called data flow equations [1].

For some problems, instead of proceeding along the flow of control and defining $out[s]$ in terms of $in[s]$, we need to proceed backward and define $in[s]$ in terms of $out[s]$. In general, equations are set up at the level of basic blocks, instead of statements. Data flow analysis gives information that does not misinterpret what the program under analysis does, in the sense that it does not tell us that a transformation of the code is safe to perform, when, in fact, it is not safe.

Computation of data dependences essentially boils down to the problem of identifying definitions and uses of all variables in a program, and the relationships among them. Based on this observation, data dependence computations have been formulated as data flow analysis problems by setting up the corresponding data flow equations. Note that computation of flow dependence is a backward analysis problem and computation of anti, output and input dependences are forwarded analysis problems.

2.4.1.5.1 *Flow Dependence.*

Flow dependence computation is essentially the identification of all reachable uses of each definition in a program, termed as *reachable uses* problem, with the following data flow equations:

$$in[B] = gen[B] \bigcup (out[B] - kill[B])$$

$$in[B] \ = \bigcup_{s \in succ(B)} in[s]$$

where, $in[B]$ is the set of uses reachable from the beginning of basic block B, $out[B]$ the set of uses reachable from the end of B, $gen[B]$ the set of upward exposed uses in B and $kill[B]$ the set of uses reachable from the end of B, of all variables defined in B.

2.4.1.5.2 *Antidependence.*

The computation of antidependence is essentially the identification of uses of variables before its redefinition on an execution path, termed as *uses before redefinition* problem, with the following data flow equations:

$$out[B] = gen[B] \bigcup (in[B] - kill[B])$$

$$in[B] \ = \bigcup_{p \in pred(B)} out[p]$$

where $in\ [B]$ is the set of uses reaching the beginning of B, without a definition of the variable in between, $out\ [B]$ is the set of uses reaching the end of B, without a definition of the variable in between, $gen\ [B]$ is the set of uses in B, without a definition of the variable later in the block and $kill\ [B]$ is the set of uses reaching the beginning of B, of all variables defined in B.

2.4.1.5.3 ***Output Dependence.***
Output dependence computation is essentially the well-known reaching definitions problem with the following data flow equations:

$$out[B] = gen[B] \bigcup (in[B] - kill[B])$$

$$in[B] = \bigcup_{p \in pred(B)} out[p]$$

where *in* $[B]$ is the set of definitions reaching the beginning of B, *out* $[B]$ is the set of definitions reaching the end of B, *gen* $[B]$ is the set of definitions in B reaching the end of B and *kill* $[B]$ is the set of definitions reaching the beginning of B, of all variables defined in B.

2.4.1.5.4 ***Input Dependence.***
For the computation of input dependence, a variant of its definition given in Section 2.2.1.1 has been used. Consider the following example:

$$\begin{aligned} s_1 : x &:= w + a \\ s_2 : y &:= w + b \\ s_3 : y &:= w + c \end{aligned}$$

Here, according to the definition, we have three input dependences due to the variable w : $s_1 \delta^i \delta_2$, $s_2 \delta^i s_3$ and $s_1 \delta^i s_3$. The implementation considered only input dependences between statements having uses of a variable without another reference — definition or use — of the same variable in between (i.e., only $s_1 \delta^i \delta_2$ and $s_2 \delta^i s_3$ in the preceding example). It is assumed that other input dependences, such as $s_1 \delta^i s_3$ cited earlier, can be derived from the dependences computed, if needed. With this notion of input dependence, the problem essentially becomes identification of last uses of variables before its redefinition on an execution path, termed as *last uses before redefinition* problem, with the following data flow equations:

$$out[B] = gen[B] \bigcup (in[B] - kill[B])$$

$$in[B] = \bigcup_{p \in pred(B)} out[p]$$

where *in*$[B]$ is the uses reaching the beginning of B, without a reference — definition or use — of the same variable, in between; *out*$[B]$ is the set of uses reaching the end of B, without a reference in between; *gen*$[B]$ is the set of uses in B, without a reference later in the block; and *kill*$[B]$ is the set of uses reaching the beginning of B, without a reference in between, of all variables referenced in B.

For each problem, *gen* and *kill* for each basic block is computed first, followed by computation of global information at the entry and exit points of each basic block by solving the set of data flow equations using an iterative process. Finally, the global data flow information is computed at points inside basic blocks using the global information available at the entry and exit points of each basic block. The iterative process for solving the systems of equations can give several solutions depending on the initialization of unknowns, and the initializations are done appropriately to get safe solutions.

The implementation adopts a conservative approach in dealing with pointers and procedure calls. Arrays cause no problems because of the way in which they are handled in low-level SUIF. Each array reference is an instruction in low-level SUIF, which essentially gives a pointer to the location of the array element, and this pointer is used later for either a read or a write. In the case of pointers, it is assumed that an assignment through a pointer can define any variable, and a use through a pointer can cause a reference to any variable. Also, it is assumed, conservatively, that no definitions or uses are killed by any definition or use through a pointer. For procedure calls, it is assumed that all variables passed as parameters and all global variables are defined and used inside the called procedure. Also, it is assumed, conservatively, that no definitions or uses reaching the call site are killed by the procedure call.

The implementation used linked list as the data structure to store the information concerning definitions and uses of variables in a program. Each element of the list is a pair, consisting of a variable name and a statement number. The linked list representation makes the implementation inefficient, but the main feature of the implementation is that it permits real-world programs as input.

2.4.1.6 Code Transformer Interface

The code transformer interface provides a convenient environment for the user to interact with the transformation system (see Figure 2.1). The system has to be initialized first, with the generation of all transformers from their respective specifications. After the initialization, a program to be transformed — in C or FORTRAN — is translated to SUIF intermediate code using the component HLL-to-SUIF. The code transformer interface takes SUIF intermediate code as input and requests the user to choose the transformation to be performed on the code. Based on the transformation specified by the user, the dependence analyzer does the required dependence analyses; for example, only flow dependence information is required for useless code elimination. The intermediate code, with the dependence information attached as annotations, is given as input to the transformer, which performs the specified transformation on the intermediate code. The output from the interface — the transformed intermediate code — is then passed to the component SUIF-to-HLL to get the transformed code back in high-level language.

2.4.2 Experiments

Routines from LINPACK benchmarks were taken as input and all the transformations generated by the system were applied on these routines. The number of applications of each transformation in each routine was counted to see the effect. The correctness of the transformations was verified by comparing the output of the transformed code with the original code before transformation. The data from the experiments conducted on the routines taken from LINPACK benchmarks are given in Table 2.2 that follows. The last two entries in Table 2.2 are sample programs created to test copy propagation and induction variable elimination, which had no effect on LINPACK benchmark programs. The experiments were conducted mainly to demonstrate the completeness and correctness of the system implementation.

2.5 Conclusion

The code transformation system described earlier provides a complete environment for automatic generation of code transformers from formal specifications. The development of the code transformation system involved formal specification of transformations, design of a specification language and the implementation of the transformation system.

The main issue concerning the specification of code transformations is the choice of a framework for the specification. This work used a framework with dependence relations as the basis for the specification of both scalar and parallel transformations.

Formal specification of code transformations not only helps in their precise understanding but also has many applications. Formal specifications of code transformations have found use in the study of ordering of transformations and transformer generators. In the work described here the whole class of

TABLE 2.2 Experimental Results

Name of the Routine	Number of Nodes	Number of SUIF Statements	Number of Applications of the Transformation						
			CTP	UCE	CFD	CPP	CSE	ICM	IVE
daxpy	19	116	0	1	0	0	0	4	0
ddot	17	100	0	1	1	0	0	3	0
dgefa	26	249	0	3	0	0	6	25	0
dgesl	36	318	0	4	0	0	6	46	0
dmxpy	35	735	0	6	0	0	26	308	0
dscal	10	59	0	1	0	0	0	2	0
epslon	7	46	1	0	1	0	0	10	0
idamax	32	159	1	2	0	0	0	4	0
matgen	29	200	0	5	0	0	1	20	0
main	202	1408	86	8	1	0	1	49	0
cpp	4	18	0	1	0	2	0	0	0
ive	5	33	0	0	0	0	0	3	1

traditional scalar transformations has been specified, using dependence relations. The specifications given in the work are, in general, improved versions — in terms of conservativeness — compared with the specifications available in the literature. Not all transformations could be specified in the chosen framework, for instance, partial redundancy elimination. Further study is required to find a better alternative to the current framework.

Generation of code for compiler transformations involves the design of a specification language for specifying those transformations. A specification language for the class of scalar transformations has been designed and the power of the language is demonstrated by the specification of a complex transformation such as induction variable elimination. The design permits extensions to the language, if needed.

The development of the code transformation system involves the design and implementation of a transformer generator and a dependence analyzer. The method of syntax-directed translation is used to translate the specification of a transformation, written in the specification language, to code for the corresponding transformation. Implementation of the dependence analyzer involves the computation of control and data dependences. In the implementation, a conservative approach is adopted to deal with pointers and procedure calls, without going for a detailed analysis in their presence. This, of course, is a limitation that should be addressed to make the system more useful. Experiments have been conducted using the system with LINPACK benchmark programs as input, transforming it using the transformers generated and verifying the output of the transformed code with that of the original one. The use of an intermediate representation like SUIF makes the system extendable and portable. The implementation of the code transformation system is restricted to scalar transformations, but the system provides a working model for automatic generation of code transformers from formal specifications.

Efficiency is not taken as a major concern in the implementation and there is scope for improvement of the system in this regard, especially in the implementation of the transformer generator and the dependence analyzer. The system requires the computation of dependence relations each time a transformation is applied. One may think of improving this by incremental updating of dependence relations instead of recomputing them each time. The system can be extended to include parallel transformations also, because dependence relations provide a uniform framework for the specification of both scalar and parallel transformations.

References

[1] A.V. Aho, R. Sethi and J.D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA, 1986.

[2] U. Banerjee, *Loop Transformations for Restructuring Compilers: The Foundations*. Kluwer Academic, Dordrecht, 1993.

[3] U. Banerjee, *Loop Parallelization*. Kluwer Academic, Dordrecht, 1994.

[4] D.F. Bacon, S.L. Graham and O.J. Sharp, Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4), 345, 1994.

[5] G. Bilardi and K. Pingali, A framework for generalized control dependence, in Proceedings of the SIGPLAN 1996 Conference on Programming Language Design and Implementation, 1996, p. 291.

[6] R. Cytron et al., Efficiently computing static single assignment form and the control dependence graph, *ACM TOPLAS*, 13(4), 451, 1991.

[7] F. Chow, A Portable Machine Independent Optimizer — Design and Implementation, Ph.D. dissertation, Department of Electrical Engineering, Stanford University, Stanford, CA; and Technical Report 83-254, Computer Systems Laboratory, Stanford University, Stanford, CA, 1983.

[8] C. Click and K.D. Cooper, Combining analyses, combining optimizations, *ACM TOPLAS*, 17(2), 181, 1995.

[9] J.W. Davidson and C.W. Fraser, Automatic generation of peephole transformations, in Proceedings of the ACM SIGPLAN 1984 Symposium on Compiler Construction, 1984, p. 111.

[10] D.M. Dhamdhere, B.K. Rosen and F.K. Zadeck, How to analyze large programs efficiently and informatively, *ACM SIGPLAN Notices*, 27(7), 212, 1992.

[11] J. Ferrante, K.J. Ottenstein and J.D. Warren, The program dependence graph and its use in optimization, *ACM TOPLAS*, 9(3), 319, 1987.

[12] C.W. Fraser and A.L. Wendt, Automatic generation of fast optimizing code generators, in Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, 1988, p. 79.

[13] M. Ganapathi and C.N. Fischer, Integrating code generation and peephole optimization, *ACTA Inf.*, 25(1), 85, 1988.

[14] M.S. Hecht, *Flow Analysis of Computer Programs*, North-Holland, Amsterdam, 1977.

[15] R.R. Kessler, Peep — An architectural description driven peephole transformer, in *SIGPLAN Notices*, 19(6), 106, 1984.

[16] S.S. Muchnick and N.D. Jones, *Program Flow Analysis: Theory and Applications*. Prentice Hall, New York, 1981.

[17] S.S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann, San Francisco, 1997.

[18] D.A. Padua and M.J. Wolfe, Advanced compiler optimizations for supercomputers, *Commun. ACM*, 29(12), 1184, 1986.

[19] V.K. Paleri, An Environment for Automatic Generation of Code Optimizers, Ph.D. thesis, Department of Computer Science and Automation, Indian Institute of Science, Bangalore, 1999.

[20] A. Podgurski and L.A. Clarke, A formal model of program dependences and its implications for software testing, debugging, and maintenance, *IEEE Trans. Software Eng.*, 16(9), 965, 1990.

[21] B.G. Ryder and M.C. Paull, Elimination algorithms for data flow analysis, *ACM Comput. Surv.*, 18(3), 277, 1986.

[22] V. Sarkar, The PTRAN parallel programming system, in *Parallel Functional Programming Languages and Compilers*, Szymanski, B., Ed., ACM Press, New York, 1991, p. 309.

[23] Stanford Compiler Group, SUIF library, Version 1.0, Stanford University, Stanford, CA, 1994.

[24] S.W.K. Tjiang and J.L. Hennessy, Sharlit — A tool for building optimizers, *ACM SIGPLAN Notices*, 27(7), 82, 1992.

[25] Wilson et al., SUIF: An infrastructure for research on parallelizing and optimizing compiler, *ACM SIGPLAN Notices*, 29(12), 31, 1994.

[26] M.J. Wolfe, *Optimizing Supercompilers for Supercomputers*, Research Monographs in Parallel and Distributed Computing, MIT Press, Cambridge, MA, 1989.

[27] M.J. Wolfe, *High Performance Compilers for Parallel Computing*, Addison-Wesley, Reading, MA, 1996.

[28] D. Whitfield and M.L. Soffa, An approach to ordering optimizing transformations, *ACM SIGPLAN Notices*, 25(3), 137, 1990.

[29] D. Whitfield and M.L. Soffa, Automatic generation of global optimizers, *ACM SIGPLAN Notices*, 26(6), 120, 1991.

[30] D. Whitfield and M.L. Soffa, The design and implementation of Genesis, *Software-Pract. Exp.*, 24(3), 307, 1994.

[31] D. Whitfield and M.L. Soffa, An approach for exploring code improving transformations, *ACM TOPLAS*, 19(6), 1053, 1997.

2.A Appendix: Syntax of the Specification Language

The full grammar for the specification language is given in this appendix. The nonterminals are indicated by italic type and the terminals, in upper case letters. Alternative categories are usually listed on separated lines; in some cases, the alternatives are presented on one line marked by the phrase "one of." An optional element is represented with a subscript "opt."

specification ::
  sub_specification_list$_{opt}$ *main_specification*

sub_specification_list ::
  sub_specification
  sub_specification_list sub_specification

sub_specification ::
  condition_sub_specification
  action_sub_specification

condition_sub_specification ::
    sub_specification_header condition_predicate

action_sub_specification ::
  sub_specification_header {*action_list*}

sub_specification_header ::
  identifier (parameter_list):

parameter_list ::
  parameter
  parameter_list, parameter

parameter :: one of
  INT_CONST OPCODE OPERAND_SYMBOL OPERAND_TYPE
  NEW_SYMBOL NEW_OPERAND NEW_STMT NULL_VALUE
  program_element identifier

condition_predicate ::
  (quantifier: conditional_expr$_{opt}$ *: conditional_expr)*

quantifier ::
  quantifier_type program_element

quantifier_type :: one of
  FORALL EXISTS NOT_EXISTS

main_specification ::
  transformation_name : action_predicate

transformation_name :: one of
  CTP CFD UCE URCE CPP CSE ICM IVE NEW

action_predicate ::
  [*action_quantifier : conditional_expr*$_{opt}$ *: conditional_expr*$_{opt}$ {*action_list*}]

action_quantifier ::
  FOREACH program_element

program_element :: one of
  OPERAND STMT LOOP NODE

conditional_expr ::
  condition
  NOT conditional_expr
  conditional_expr AND conditional_expr
  conditional_expr OR conditional_expr

condition ::
  VAR_OPERAND (OPERAND)
  INT_CONST_OPERAND (OPERAND)
  SAME_OPCODE (STMT, OPCODE)
  SAME_OPERAND_SYMBOL (OPERAND, OPERAND)
  SAME_OPERAND_VALUE (OPERAND, OPERAND)
  SAME_EXPR (EXPR, EXPR)
  SAME_STMT (STMT, STMT)
  VALID_CSE_EXPR (EXPR)
  STMT_IN_LOOP (STMT, LOOP)
  STMT_IN_LOOP_BODY (STMT, LOOP)
  CTRL (NODE, NODE)
  CTRL_DEP_STMT (STMT, STMT)
  SAME_CTRL_DEP (STMT, STMT)
  COND_BRANCH_TRUTH_VALUE (STMT)
  OPERAND_REDEFN_BETWEEN_STMTS (STMT, STMT)
  parameter RELOP parameter
  stmt_type (STMT)
  dep_type (STMT)
  identifier (parameter_list)

stmt type :: one of
  LABEL_STMT CONST_DEFN COPY_STMT COND_BRANCH
  COND_BRANCH_TEST USELESS_STMT FOLDABLE_STMT MOVABLE_STMT

dep_type :: one of
  FLOW ANTI OUTPUT INPUT

action_list ::
  action
  action_list action

```
action ::
   DELETE_STMT (STMT);
   DELETE_NODE (NODE);
   REPLACE_OPERAND (OPERAND, parameter);
   REPLACE_STMT (STMT, NEW_STMT);
   INSERT_STMT (NEW_STMT, POSITION);
   MOVE_STMT (STMT, POSITION);
   FOLD_STMT (STMT);
   SET_OPCODE (STMT, OPCODE);
   CREATE_NEW_SYMBOL (OPERAND_TYPE, SCOPE, NEW_SYMBOL);
   CREATE_NEW_OPERAND (parameter, NEW_OPERAND);
   CREATE_NEW_STMT (STMT_CLASS, parameter_list, NEW_STMT);
   IF (conditional_expr) action_list
   IF (conditional_expr) action_list ELSE action_list
   BREAK;
   action_predicate
   {action_list}
   identifier (parameter_list);

identifier ::
   IDENTIFIER
```

3

Scalar Compiler Optimizations on the Static Single Assignment Form and the Flow Graph

Y.N. Srikant
Indian Institute of Science

3.1 Introduction

The topic of scalar compiler optimizations is definitely not new. It has been studied, researched and criticized for over 40 years. Several books [3, 7, 8, 41, 44] and articles contain detailed descriptions of important optimizations that are performed by compilers. However, the last word has not yet been said on this topic. The emergence of newer intermediate forms and extremely sophisticated microprocessors have always managed to keep the subject alive. Some of the important classical optimization algorithms have been made simpler, more precise and more efficient due to the emergence of the static single assignment (SSA) form as an important intermediate representation used in compilers [9]. In this chapter we present some of the important optimization algorithms as applied to the traditional flow graph and the SSA form. We hope that this helps in gaining an insight into the working of SSA-based algorithms.

0-8493-1240-7/03/$0.00+$1.50

We discuss partial redundancy elimination [7,10–13,44], global value numbering [7, 15, 16, 19, 44] and conditional constant propagation [2, 7, 44]. Out of these three optimizations, we discuss the latter two on both the flow graph and the SSA form, but we discuss partial redundancy elimination (PRE) only on the flow graph. PRE has emerged as one of the most important scalar optimizations of today and is still one of the most intensely researched topics. Whereas the concepts behind the flow graph version of PRE have stabilized, the SSA version is just emerging [13], and has not established its superiority over the flow graph version (unlike the other two algorithms mentioned earlier). Several other algorithms such as dead code elimination [9, 41, 44], strength reduction [1, 8, 17, 33–35, 44] and array-bound check elimination [18, 21, 22] have not been included due to space constraints.

Section 3.2 discusses the SSA form with several examples, its construction, its advantages and its disadvantages. Section 3.3 contains a detailed presentation of the conditional constant propagation algorithms, both on the flow graph and the SSA form. This is followed by a description of the PRE algorithm in Section 3.4, and the value-numbering algorithms in Section 3.5. Conclusions and future directions are presented in Section 3.6.

In our discussion, we assume that the reader is familiar with the terminology of control flow graphs, basic blocks, paths, etc. These definitions are available in [3, 9, 41, 44].

3.2 Static Single Assignment Form

The SSA form has emerged in recent years as an important intermediate representation used in compilers. A program is in SSA form if each of its variables has exactly one definition that implies each use of a variable is reached by exactly one definition. The control flow remains the same as in a traditional (non-SSA) program. A special merge operator, denoted ϕ, is used for the selection of values in join nodes. The SSA form is usually augmented with use–definition or definition–use chains in its data structure representation to facilitate design of faster algorithms.

Figures 3.1 and 3.4 show two non-SSA form programs, Figures 3.2 and 3.5 show their SSA forms and Figures 3.3 and 3.6 show the flowcharts of SSA forms.

```
Read $A, B, C$
if $(A > B)$
   if $(A > C)$ $max = A$
   else $max = C$
else if $(B > C)$ $max = B$
      else $max = C$
Print $max$
```

FIGURE 3.1 Program in non-SSA form.

```
Read $A, B, C$
if $(A > B)$
   if $(A > C) max_1 = A$
   else $max_2 = C$
else if $(B > C)$ $max_3 = B$
      else $max_4 = C$
$max_5 = \phi(max_1, max_2, max_3, max_4)$
Print $max_5$
```

FIGURE 3.2 Program in Figure 3.1 in SSA form.

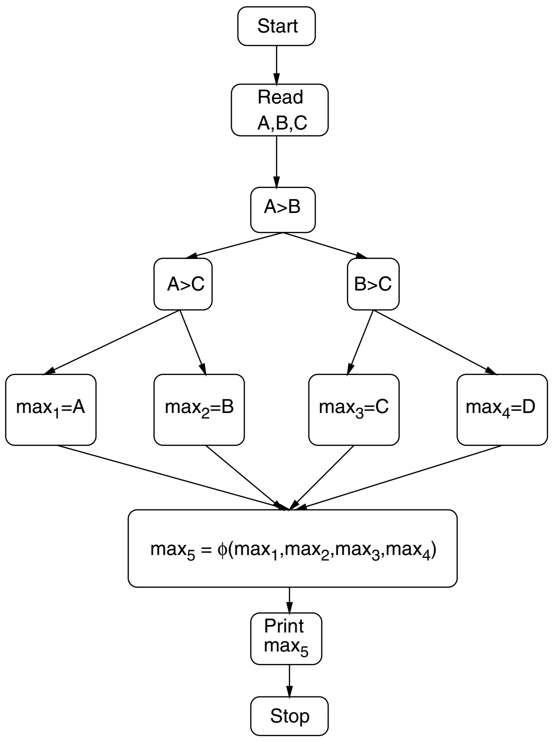


FIGURE 3.3 Flowchart of program in Figure 3.2.

```
Read $A$; $LSR = 1$; $RSR = A$; $SR = (LSR + RSR)/2$;
Repeat
    $T = SR * SR$;
    if $(T > A) RSR = SR$
    else if $(T < A) LSR = SR$
            else begin $LSR = SR$; $RSR = SR$; end
    $SR = (LSR + RSR)/2$;
until $(LSR \neq RSR)$
Print $SR$
```

FIGURE 3.4 Another program in non-SSA form.

The program in Figure 3.1 is not in SSA form because there are several assignments to the variable *max*. In the program in Figure 3.2 (see Figure 3.3 also), each assignment is made to a different variable, max_i. Variable max_5 is assigned the correct value by the ϕ-function, which takes the value max_i, if the control reaches it via the ith incoming branch from left to right.

The ϕ-functions in the two blocks B_1 and B_5 in Figure 3.6 are meant to choose the appropriate value based on the control flow. For example, the ϕ-assignment to RSR_5 in the block B_5 in Figure 3.6 selects one of RSR_3, RSR_2, or RSR_4 based on the execution following the arc $B2 \to B5$, $B3 \to B5$, or $B4 \to B5$, respectively.

Usually, compilers construct a control flow graph representation of a program first, and then convert it to SSA form. The conversion process involves introduction of statements with assignment to ϕ-functions in several join nodes and renaming of variables that are targets of more than one definition. Of course, the usages of such variables can also be changed appropriately. Not every join

```
Read $A$; $LSR_1 = 1$; $RSR_1 = A$; $SR_1 = (LSR_1 + RSR_1)/2$;
Repeat
        $LSR_2 = \phi(LSR_5, LSR_1)$;
        $RSR_2 = \phi(RSR_5, RSR_1)$;
        $SR_2 = \phi(SR_3, SR_1)$;
        $T = SR_2 * SR_2$;
        if $(T > A) RSR_3 = SR_2$
        else if $(T < A) LSR_3 = SR_2$
            else begin $LSR_4 = SR_2$; $RSR_4 = SR_2$; end
        $LSR_5 = \phi(LSR_2, LSR_3, LSR_4)$;
        $RSR_5 = \phi(RSR_3, RSR_2, RSR_4)$;
        $SR_3 = (LSR_5 + RSR_5)/2$;
until $(LSR_5 \neq RSR_5)$
Print $SR_3$
```

FIGURE 3.5 Program in Figure 3.4 in SSA form.

node needs a ϕ function for every variable in the program. Suitable placement of ϕ functions so as to ensure a minimal SSA form and the renaming of variables are topics covered in the next section and are also available in [9, 14, 41, 44].

A reader unfamiliar with SSA forms may wonder about the role of ϕ-functions in the final machine code of the program. No direct translation of a ϕ-function to machine code is possible for general purpose processors of today. A copy instruction needs to be inserted at the end of each predecessor block of the block containing a ϕ-function (the temporary variable t is essential in some cases to maintain correctness, but not in this case; see [44] for a full discussion). This introduces some inefficiency into machine code, which can be annulled to some extent by good register allocation [9]. Carrying out dead code elimination before ϕ-conversion is also required to remove redundant assignment statements. It is not possible to replace all subscripted instances of a variable x by x itself due to possible movement of code during optimizations. For example, dropping the subscripts for a would lead to wrong code shown later in Figure 3.16.

Figure 3.7 shows a simple example of the effect of ϕ-conversion. During register allocation, the same register would be assigned to t, max_1, max_2, max_3, max_4 and max_5, and thereby eliminating copying.

3.2.1 Construction of the Static Single Assignment Form

We now discuss the construction of an SSA form from a flow graph. We consider only scalars and refer the reader to [9, 37] for details on how structures, pointers and arrays are handled. Our presentation is based on the material in [9].

3.2.1.1 Conditions on the SSA Form

After a program has been translated to SSA form, the new form should satisfy the following conditions for every variable v in the original program:

1. If two paths from nodes having a definition of v converge at a node p, then p contains a trivial ϕ-function of the form $v = \phi(v, v, \ldots, v)$, with a fanout equal to the in-degree of v.
2. Each appearance of v in the original program or a ϕ-function in the new program has been replaced by a new variable v_i, leaving the new program in SSA form.
3. Any use of a variable v along any control path in the original program and the corresponding use of v_i in the new program yield the same value for both v and v_i.

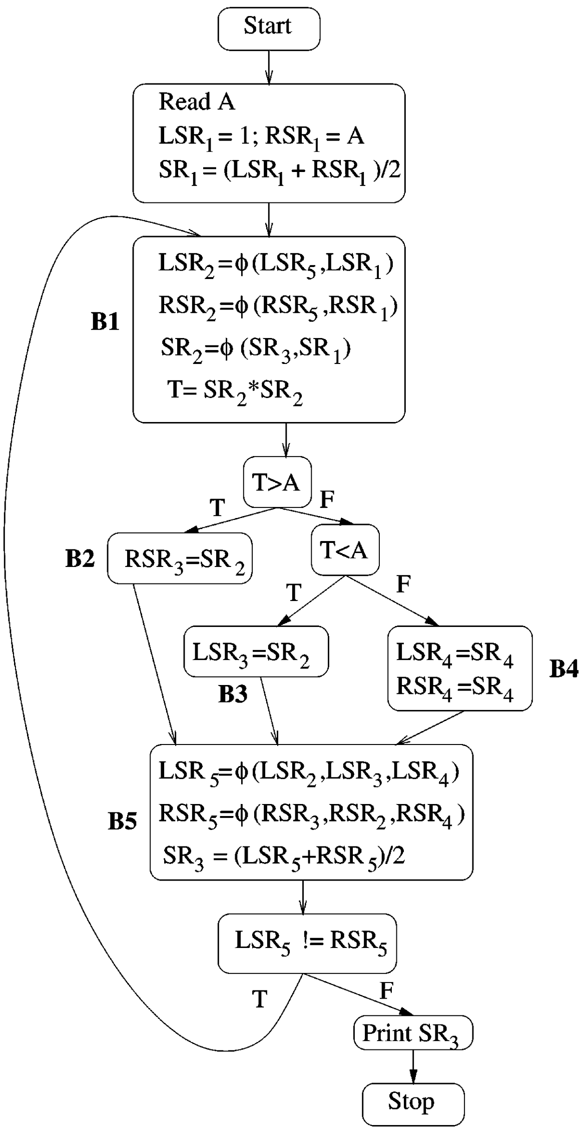


FIGURE 3.6 Flowchart of program in Figure 3.5.

Preceding condition 1 is recursive. It implies that ϕ-assignments introduced by the translation procedure can also qualify as assignments to v and this in turn may lead to introduction of more ϕ-assignments at other nodes.

3.2.1.2 The Join Set and ϕ-Nodes

Given a set $\mathscr{S}$ of flow graph nodes, we define the set $JOIN(\mathscr{S})$ of nodes from the flow graph to be the set of all nodes n, such that there are two nonnull paths in the flow graph that start at two distinct nodes in $\mathscr{S}$ and converge at n. The iterated join set, $JOIN^{+}(\mathscr{S})$, is the limit of the monotonic nondecreasing sequence of sets of nodes:

$$JOIN^{(1)}(\mathscr{S}) = JOIN(\mathscr{S})$$

$$JOIN^{(i+1)}(\mathscr{S}) = JOIN(\mathscr{S} \cup JOIN^{(i)}(\mathscr{S}))$$

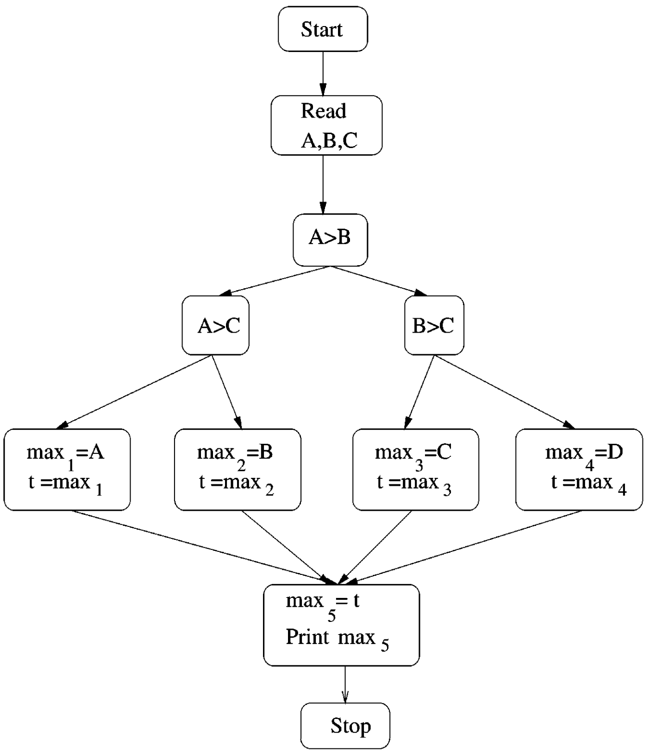


FIGURE 3.7 ϕ-Conversion on program in Figure 3.3.

We note that if $\mathcal{S}$ is defined to be the set of assignment nodes for a variable v, then $JOIN^+(\mathcal{S})$ is precisely the set of flow graph nodes, where ϕ-functions are needed (for v). See [9] for proofs. $JOIN^+(\mathcal{S})$ is precisely the *iterated dominance frontier*, $DF^+(\mathcal{S})$ [9], which can be computed efficiently in a manner to be described shortly.

3.2.1.3 Dominator Tree

Given two nodes x and y in a flow graph, x dominates y, if x appears in all paths from the *Start* node to y; x strictly dominates y, if x dominates y and $x \neq y$; and x is the immediate dominator of y (denoted $idom(y)$), if x is the closest strict dominator of y. A dominator tree shows all the immediate dominator relationships. Figure 3.8 shows a flow graph and its dominator tree. Dominator trees can be constructed in time almost linear in the number of edges of a flow graph [39] (see [40] for a linear time but more difficult algorithm).

3.2.1.4 Dominance Frontier

For a flow graph node x, the set of all flow graph nodes y, such that x dominates a predecessor of y, but does not strictly dominate y is called the dominance frontier of x and is denoted by $DF(x)$. The following redefinition of $DF(x)$ makes it simple to compute it in linear time:

$$DF(x) = DF_{local}(x) \cup \bigcup_{z \in children(x)} DF_{up}(y)$$

$$DF_{local}(x) = \{y \in successor(x) \mid idom(y) \neq x\}$$

$$DF_{up}(x) = \{y \in DF(x) \mid idom(y) \neq parent(x)\}$$

Here, $children(x)$ and $parent(x)$ are defined over the dominator tree and $successor(x)$ is defined over the flow graph. The algorithm in Figure 3.9 computes the dominance frontier based on the definitions given earlier. It is called on the root of the dominator tree and the tree is traversed in postorder. Figure 3.8 shows the *DF* sets as decorations on the nodes of the dominator tree.

We now extend the definition of *DF* to act on sets and also define the iterated dominance frontier, on lines similar to $JOIN^{+}(\mathscr{S})$.

$$DF(\mathscr{S}) = \bigcup_{x \in \mathscr{S}} DF(x)$$

$$DF^{(1)}(\mathscr{S}) = DF(\mathscr{S})$$

$$DF^{(i+1)}(\mathscr{S}) = DF(\mathscr{S} \cup DF^{(i)}(\mathscr{S}))$$

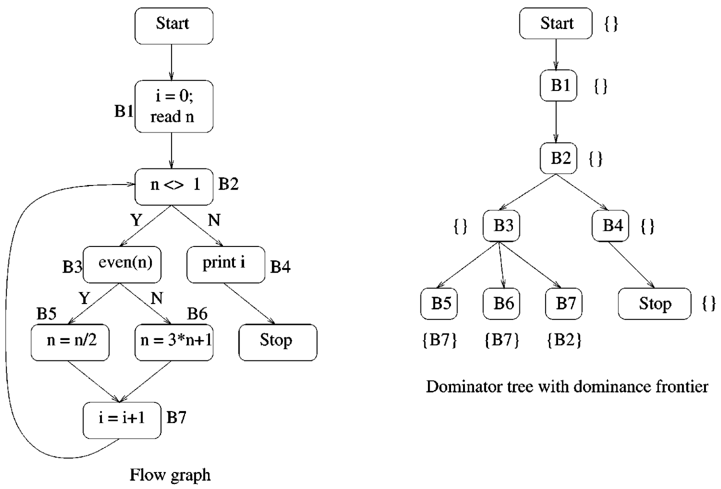


FIGURE 3.8 Example flow graph for SSA form construction.

```
function $dominance\_frontier(n)$ // $n$ is a node in the dominator tree
begin
   for all children $c$ of $n$ in the dominator tree do
      $dominance\_frontier(n.c)$;
   end for
   $DF(n) = \emptyset$;
   $//DF_{local}$ computation
   for all successors of $n$ in the flow graph do
      if $(idom(s) \neq n)$ then $DF(n) = DF(n) \cup \{s\}$;
   end for
   // $DF_{up}$ computation
   for all children $c$ of $n$ in the dominator tree do
      for all $p \in DF(c)$ do
         if $(idom(p) \neq n)$ then $DF(n) = DF(n) \cup \{p\}$;
      end for
   end for
end
```

FIGURE 3.9 Dominance frontier computation.

As mentioned before, for each variable v, the set of flow graph nodes that need ϕ-functions is $DF^+(\mathscr{S})$, where $\mathscr{S}$ is the set of nodes containing assignments to v. We do not construct $DF^+(\mathscr{S})$ explicitly. It is computed implicitly during the placement of ϕ-functions.

3.2.1.5 Minimal SSA Form Construction

The three steps in the construction of the minimal SSA form are:

1. Compute *DF* sets for each node of the flow graph using the algorithm in Figure 3.9.
2. Place trivial ϕ-functions for each variable in the nodes of the flow graph using the algorithm in Figure 3.10.
3. Rename variables using the algorithm in Figure 3.11.

The function place Phi-Function (v) is called once for each variable v. It can be made more efficient by using integer flags instead of Boolean flags as described in [9]. Our presentation uses Boolean flags to make the algorithm simpler to understand.

The renaming algorithm in Figure 3.11 performs a top-down traversal of the dominator tree. It maintains a version stack V, whose top element is always the version to be used for a variable usage encountered (in the appropriate range, of course). A counter v is used to generate a new version number. It is possible to use a separate stack for each variable as in [9], so as to reduce the overheads a bit. However, we believe that our presentation is simpler to comprehend.

```
function Place-phi-function($v$) // $v$ is a variable
// This function is executed once for each variable in the flow graph
begin
  // has-phi ($B$) is true if a $\phi$-function has already
  // been placed in $B$
  // processed($B$) is true if $B$ has already been processed once
  // for variable $v$
  for all nodes $B$ in the flow graph do
    has-phi($B$) = false; processed($B$) = false;
  end for
  W = $\emptyset$; // $W$ is the work list
  // Assignment-nodes($v$) is the set of nodes containing
  // statements assigning to $v$
  for all nodes $B \in$ Assignment-nodes($v$) do
     processed ($B$) = true; Add($W$, $B$);
  end for
  while $W \neq \emptyset$; do
  begin
    $B$ = Remove($W$);
    for all nodes $y \in DF(B)$ do
      if (not has-phi($y$)) then
      begin
           place $< v = \phi(v, v, \ldots, v) >$ in $y$;
           has-phi($y$) = true;
           if (not processed($y$)) then
           being processed($y$) = true; Add($W$, $y$); end
      end
    end for
  end
end
```

FIGURE 3.10 Minimal SSA construction.

```
function $Rename\text{-}variables(x, B)$//$x$ is a variable and $B$ is a block
begin
  $v_e = Top(V)$; //$V$ is the version stack of $x$
  for all statements $s \in B$ do
     if $s$ is a non-$\phi$ statement then
        replace all uses of $x$ in the $RHS(s)$ with $Top(V)$;
     if $s$ defines $x$ then
     begin
       replace $x$ with $x_v$ in its definition; push $x_v$ onto $V$;
       // $x_v$ is the renamed version of $x$ in this definition
       $v = v + 1$; $v$ is the version number counter
     end
  end for
  for all successors $s$ of $B$ in the flow graph do
     $j$ = predecessor index of $B$ with respect to $s$
     for all $\phi$-functions $f$ in $s$ which define $x$ do
        replace the $j^{th}$ operand of $f$ with $Top(V)$;
     end for
  end for
  for all children $c$ of $B$ in the dominator tree do
     $Rename\text{-}variables\ (x, c)$;
     end for
  repeat $Pop(V)$; until ($Top(V) == v_e$);
end

being // calling program
  for all variables $x$ in the flow graph do
    $V = \emptyset$; $v = 1$; push 0 onto $V$; // end-of-stack marker
    $Rename\text{-}variables\ (x, Start)$;
    end for
end
```

FIGURE 3.11 Renaming variables.

Let us trace the steps of the SSA construction algorithm with the help of the example in Figure 3.12. Let us concentrate on the variable n. Blocks B_1, B_5 and B_6 have assignments to n (**read** n is also considered as an assignment). With the dominance frontier of B_1 as null, no ϕ-function is introduced while processing it. A ϕ-function is introduced for n in B_7 while processing B_5 (this is not repeated while processing B_6). B_2 gets a ϕ-function for n when B_7 is handled.

Let us now understand how different instances of n are renamed. The instruction **read** n in B_1 becomes **read** n_0 while processing B_1. At the same time, the second parameter of the ϕ-function for n in block B_2 is changed to n_0. Processing B_2 in the top-down order results in changing the statement $n = \phi(n, n_0)$ to $n_1 = \phi(n, n_0)$ and the new version number 1 is pushed onto the version stack V. This results in changing the comparisons $n \neq 1$ to $n_1 \neq 1$ in block B_2 and $even(n)$ to $even(n_1)$ in block B_3. The expressions $n/2$ in block B_5 and $3 * n + 1$ in block B_6 also change to $n_1/2$ and $3 * n_1 + 1$, respectively. A new version of n, namely, n_2 (n_3, respectively) is created while processing the assignment in block B_5 (B_6, respectively). The version number 2 (3, respectively) is pushed onto V. This results in changing the parameters of the ϕ-function in block B_7 as shown in Figure 3.12. After finishing with B_5 (B_6, respectively), V is popped to remove 2 (3, respectively), before processing B_7. A new version n_4 is created while processing the ϕ statement in B_7, which in turn changes the first parameter of the ϕ-function for n in block B_2, from n to n_4. V is then popped and recursion unwinds. The variable i is treated in a similar manner.

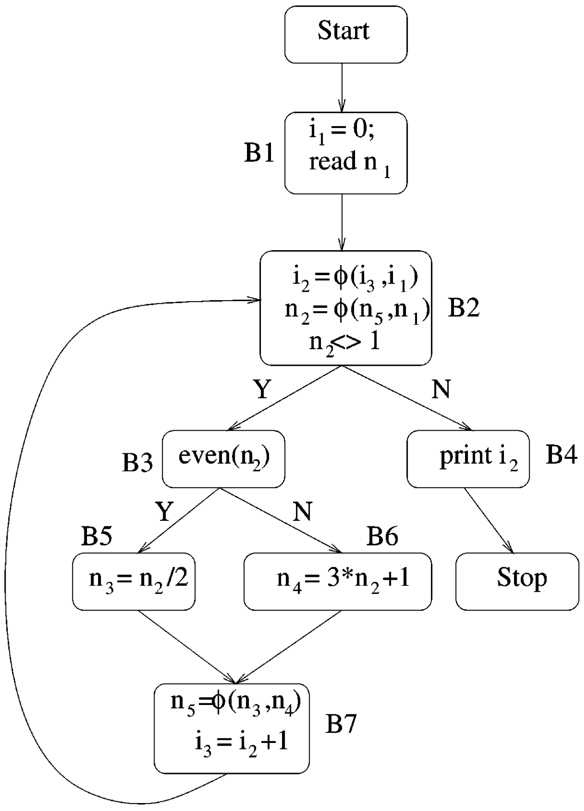


FIGURE 3.12 SSA form construction for the flow graph in 3.8.

3.2.1.6 Complexity of SSA Graph Construction

We define R, the size of a flow graph as follows:

$$R = max\{N, E, A, M\},$$

where N is the number of nodes in the flow graph, E is the number of edges in the flow graph, A is the number of assignments in the flow graph and M is the number of uses of variables in the flow graph.

The construction of the dominance frontier and the SSA form in theory take $O(R^2)$ and $O(R^3)$ time, respectively. However, according to [9], measurements on programs show that the size of dominance frontiers in practice is small and hence the entire construction process, including construction of dominance frontiers, takes only $O(R)$ time.

3.2.1.7 Note on the Size of SSA Graphs

An SSA form is usually augmented with links from every unique definition of a variable to its uses (corresponding to *d-u* information). Some algorithms need SSA forms augmented with links that go from every use of a variable to its unique definition (corresponding to *u-d* information). If n definitions are in a program, and each of these could reach n uses, then both *d-u* and *u-d* chains can have O(n^2) links. However, an SSA graph with *d-u* or *u-d* information can have only O(n) links due to the factoring carried out by ϕ-functions. See Figures 3.13 and 3.14 for an example.

In Figure 3.13, the *d-u* chain of each definition of i contains all the three uses of i in the second **switch** statement. However, in Figure 3.14, due to the factoring introduced by the ϕ-function, each definition reaches only one use.

```
switch (j)
    case 1: i = 10; break;
    case 2: i = 20; break;
    case 3: i = 30; break;
end
switch (k)
    case 1: x = i * 3 + 5; break;
    case 2: y = i * 6 + 15; break;
    case 3: z = i * 9 + 25; break;
end
```

FIGURE 3.13 Program in non-SSA form.

```
switch (j)
    case 1: i1 = 10; break;
    case 2: i2 = 20; break;
    case 3: i3 = 30; break;
end
i4 = φ(i1, i2, i3);
switch (k)
    case 1: x = i4 * 3 + 5; break;
    case 2: y = i4 * 6 + 15; break;
    case 3: z = i4 * 9 + 25; break;
end
```

FIGURE 3.14 Program in Figure 3.13 in SSA form.

```
      b = 5;
L1:   if (b > 100) goto L2;
      read a;
      b = a + b;
      a = 16;
      b = a + b;
      goto L1
L2:   stop
```

FIGURE 3.15 Loop invariant code motion not possible.

The renaming algorithm of Figure 3.11 can be augmented to establish the *d-u* and *u-d* links. This requires that every statement be a separate node in the flow graph. It also requires keeping a pointer to the node defining a variable on the version stack along with the name of the variable. The rest of the process is simple.

The SSA form increases the number of variables. If n variables are in a program and each of these has k definitions, then the SSA form would have nk variables to take care of the nk definitions. It may not be possible to map these nk variables back to n variables during machine code generation because of code movements that could have taken place during optimizations. Figures 3.15 and 3.16 show an example of such a code motion.

```
      b = 5; a_2 = 16;
L1:   b_2 = φ(b_1, b_4);
      if (b_2 > 100) goto L2;
      read a_1;
      b_3 = a_1 + b_2;
      b_4 = a_2 + b_3;
      goto L1
L2:   stop
```

FIGURE 3.16 Loop invariant code moved out.

3.2.2 Advantages of the Static Single Assignment Form

- Only one definition reaches every usage and this makes optimization algorithms simpler. For example, this enables efficient constant propagation along SSA edges.
- The number of edges in an SSA graph (including *d-u* and *u-d* information edges) is generally only $O(max\{N, E\})$, with N and E the number of nodes and edges in the flow graph, respectively. This makes SSA graphs sparse and hence optimization algorithms using edge traversals are efficient. For example, conditional constant propagation on SSA graphs is better and more efficient than the corresponding version on flow graphs [2].
- The single assignment property makes better code movement possible. For example, in the program in Figure 3.15, the assignment $a = 16$ cannot be moved out of the loop due to the presence of the statement **read** a. However, once the program is converted to SSA form, this is possible (see Figure 3.16). Note that other optimizations such as constant propagation have not been applied in either of the programs.
- The ϕ-functions are the only points where several definitions merge and this makes code movement during transformations easier. Further, ϕ-functions can also be used to indicate beginning and end of loops, conditionals, etc.
- Several optimization algorithms operating on SSA forms do not need iterative data flow analysis. They use the dominator tree and the dominance frontier instead. An example is global value numbering.
- Most global optimization algorithms are no more expensive than local versions. An example is partition-based value numbering.

3.2.3 Disadvantages of the Static Single Assignment Form

- Several instances of a single variable are created. Potentially, n^2 instances can be created out of n variables. This increases the memory requirement and hence SSA forms may not be suitable for use in compilers that perform aggressive space optimizations (e.g., compilation for embedded systems).
- Optimizations needing information that cannot be computed using dominator and *d-u* information are not efficiently performed on the SSA form. An example is liveness analysis [32, 38, 41, 44].
- Optimizations such as loop unrolling, procedure in-lining and loop tiling do not benefit from using the SSA form. Similarly, dependence analysis that needs array subscript intersection tests does not benefit significantly from the use of SSA forms.
- Register allocation needs an interference graph of some kind and SSA forms do not facilitate its construction easily (liveness analysis is needed).

- The effects of SSA forms on the quality of code generated have not been adequately studied. However, it is known that good register allocation is essential to make the quality of code even acceptable.
- Some of the optimization algorithms use SSA forms with *d-u edges* [2] and a few others use SSA forms with *u-d edges* [17]. There is no single *canonical SSA form* on which all the optimization algorithms operate.
- Some of the known optimization algorithms using SSA graphs are not as efficient as ones on flow graphs in practice. Examples are partition-based value numbering and partial redundancy elimination [13, 15].
- Because all optimizations cannot be carried out beneficially on the SSA form, conversions from flow graph to SSA and back to flow graph introduces some inefficiency in the compiler.
- The SSA form is a conservative mechanism of guaranteeing single definition property at runtime. Transformations require to know that when a variable v is used, exactly one definition of v can reach that use at runtime. This is sufficient. However, SSA forms (over-)guarantee this by enforcing it in program text (the static part).
- The ϕ-functions form a bottleneck for movement of code. In general, a ϕ-function cannot be moved across basic blocks, because it does not explicitly mention the control conditions under which an input is selected. Alternative SSA forms such as gated SSA forms [42] address this issue.

3.3 Conditional Constant Propagation

Conditional constant propagation (CCP) is one of the well-known compiler optimizations. Variables that can be determined to contain only constant values at runtime in all possible executions of the program are discovered during this optimization. These values are also propagated throughout the program, and expressions whose operands can be determined to be only constants are also discovered and evaluated. In effect, a symbolic execution of the program is with the limited purpose of discovering constant values. The following examples help in explaining the intricacies of constant propagation.

The simplest case is that of straight-line code without any branches, as in a basic block. This requires only one pass through the code with forward substitutions and no iteration. However, such a one-pass strategy cannot discover constants in conditionals and loops. For such cases we need to carry out data flow analysis involving work lists and iterations. A simple algorithm of this kind adds successor nodes to the work list as symbolic execution proceeds. Nodes are removed one at a time from the work list and executed. If the new value at a node is different from the old value, then all the successors of the node are added to the work list. The algorithm stops when the work list becomes empty. Such an algorithm can catch constants in programs such as program A but not in programs such as program B in the Figure **??**.

CCP handles programs such as program B in Figure **??** by evaluating all conditional branches with only constant operands. This uses a work list of edges (instead of nodes) from the flow graph. Further, both successor edges of a branch node are not added to the work list, when the branch condition evaluates to a constant value (true or false); only the relevant successor (true or false) is added.

CCP algorithms on SSA graphs are faster and more efficient than the ones on flow graphs. They find at least as many constants as the algorithms on flow graphs (even with *d-u* chains, this efficiency cannot be achieved with flow graphs, see [2]). In the following sections, we present CCP algorithms on flow graphs as well as SSA graphs. Our description is based on the algorithms presented in [2]. *We assume that each node contains exactly one instruction and that expressions at nodes can contain*

Program A	Program A after simple CP
$a = 10$; if $(i > j)$ $b = a$ else $c = a$;	$a = 10$; if $(i > j)$ $b = 10$ else $c = 10$;
Program B	Program B after simple CP
$a = 10$; $b = 20$; if $(a == 10)$ $x = b$ else $x = a$;	$a = 10$; $b = 20$; if $(a == 10)$ $x = 20$ else $x = 10$;
	Program B after CCP with flowgraph
	$a = 10$; $b = 20$; $x = 10$;
Program C	Program C after CCP with flowgraph
$a = 10$; $b = 20$; if $(b == 20)$ $a = 30$; $d = a$;	$a = 10$; $b = 20$; $a = 30$; $d = a$; (a cannot be determined to be a constant)
	Program C after CCP with SSA graph

	$a = 10$;// unused code, removable
	$b = 20$;
	$a = 30$;
	$d = 30$; (a has been determined to be a constant)

FIGURE 3.17 Limitations of various CP algorithms.

at most one operator and two operands (except for ϕ-functions). The graphs are supposed to contain N number of nodes, E_f number of flow edges, E_s number of SSA edges (corresponding to *d-u* information) and V number of variables.

3.3.1 Lattice of Constants

The infinite lattice of constants used in the constant propagation algorithms is a flat lattice as shown in Figure 3.18. $\top$ stands for an as yet undetermined constant $\perp$ for a nonconstant and $\mathscr{C}_i$ for a constant value. The meet operator $\sqcap$ is defined as in Figure 3.19, with **any** standing for any lattice value in Figure 3.19.

3.3.2 Conditional Constant Propagation Algorithm: Flow Graph Version

All variables are supposed to be used only after defining them and are initialized by the CCP algorithm to $\top$ at every node. This is a special feature of the Wegman–Zadeck algorithm that enables it to find more constants than other algorithms (in programs with loops). For details of this effect, the reader is referred to [2]. Each node is supposed to have two lattice cells per variable, one to store the incoming value and the other to store the computed value (exit value). Also two lattice cells are at each node to store the old and new values of the expression. All the edges going out of the start node are initially

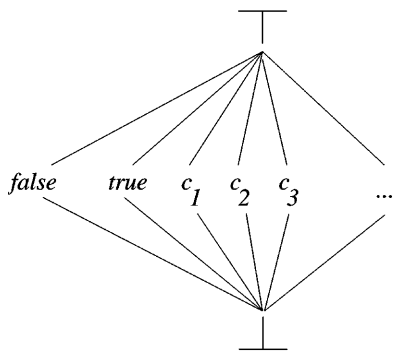


FIGURE 3.18 The lattice of constants.

any $\sqcap$ $\top = $ any
any $\sqcap$ $\bot = \bot$
$\mathscr{C}_i \sqcap \mathscr{C}_j = \bot$ if $i \neq j$
$\mathscr{C}_i \sqcap \mathscr{C}_j = \mathscr{C}_j$, if $i = j$

FIGURE 3.19 The meet operator.

val $(a\ op\ b)$ $= \bot$, if either or both of a and b is $\bot$
$= \mathscr{C}_i\ op\ \mathscr{C}_j$, if val (a) and val (b) are constants, $\mathscr{C}_i$ and $\mathscr{C}_j$, respectively
$= \top$, otherwise

Special rules for $\vee$ and $\wedge$
any stands for an element of {true, false, $\bot$, $\top$}
These rules indicate that if one of the operands is known in the shown cases, then the other operand is irrelevant

any $\vee$ true = true
true $\vee$ any = true
any $\wedge$ false = false
false $\wedge$ any = false

FIGURE 3.20 Expression evaluation.

put on the work list and marked as *executable*. The algorithm does not process any edges that are not so marked. A marked edge is removed from the work list and the node at the target end of the edge is symbolically executed. This execution involves computing the lattice value of all variables (not just the assigned variable). The incoming value of a variable x at a node y is computed as the *meet* of the exit values of x at the preceding nodes of y, and the values are stored in the incoming lattice cells of the respective variables at the node.

The expression at the node is evaluated by the rules given in Figure 3.20. If the node contains an assignment statement, and if the new value of the expression is lower (in lattice value) than the existing value of the assignment target variable, then all outgoing edges of the node are marked as executable and added to the work list. The new value is also stored in the exit lattice cell of the assignment target variable.

If the node contains a branch condition, and if the new value of the expression is lower than its existing value, then one of the following two actions is taken. If the new value is $\perp$, then both the successor edges of the node are marked and added to the work list. If the new value is a constant (*true* or *false*), then only the corresponding successor edge is marked and added to the work list. This step enables elimination of unreachable code. The new value of the expression is also stored in a lattice cell at the node.

In both these cases, note that no action is taken if the value of the expression does not change from one visit to the next. There is also a copying of all incoming lattice cells apart from the target variable of the assignment, to their exit cells, so as to enable the successor nodes to pick up values of variables. Figures 3.21 and 3.22 show the CCP algorithm in more detail.

3.3.2.1 Asymptotic Complexity

Because each variable can take only three lattice values ($\top$, $\perp$ and $\mathscr{C}_i$), starting from a value of $\top$, a variable can be lowered only twice. Each node can be visited through any of its predecessors, and

```
// $\mathscr{G} = (\mathscr{N}, \mathscr{E}_f)$ is the flowgraph, and $\mathscr{V}$ is the set of variables used in the flowgraph
begin
  $Pile = \{(Start \to n) | (Start \to n) \in \mathscr{E}_f\}$;
  Mark all edges $\{(Start \to n) | (Start \to n) \in \mathscr{E}_f\}$ as $executable$
  and all other edges as $unexecutable$;
  // $y.oldval$ and $y.newval$ store the lattice values of expressions at node $y$
  for all $y \in \mathscr{N}$ do
    $y.oldval = \top$; $y.newval = \top$;
    for all $u \in \mathscr{V}$ do
      $y.v.incell = \top$; $y.v.outcell = \top$;
    end for
  end for
  while $Pile \neq \in$ do
  begin
    $(x, y) =$ remove $(Pile)$;
    // $y.i.incell$ and $y.i.outcell$ are the lattice cells
    // associated with the variable $i$ at node $y$
    for all $i \in \mathscr{V}$ do
      $y.i.incell = \sqcap_{p \in Pred(y)} p.i.outcell$
    end for
    // evaluate expression as in Figure 3.20
    $y.newval = evaluate(y)$;
    switch $(y)$
      case $y$ is an assignment node:
        // $y.instruction.output.outcell$ is the $out$-lattice cell of the target variable
        // of the assignment at node $y$ and it contains the value from a previous visit to $y$
        if $(y.newval < y.instruction.output.outcell)$ then
        begin
          Mark all edges $\{(y \to n) | (y \to n) \in \mathscr{E}_f\}$ as $executable$
            and add them to $Pile$;
          for all $i \in V$ do $y.i.outcell = y.i.incell$; end for // copy cells
          $y.instruction.output.outcell = y.newval$;
          $y.oldval = y.newval$;
        end
      end case
```

FIGURE 3.21 CCP algorithm with flow graph.

```
          case $y$ is a branch node:
            // copy lattice cells
            for all $i \in \mathscr{V}$ do
              $y.i.outcell = y.i.incell$;
            end for
            if ($y.newval < y.oldval$) then
              begin
                switch ($y.newval$)
                  case $\perp$: // both $true$ and $false$ branches are equally likely
                    Mark all edges $\{(y \to n) | (y \to n) \in \mathscr{E}_f)\}$ as $excutable$
                      and add them to $Pile$;
                    $y.oldval = y.newval$;
                  end case
                  case $true$:
                    Mark the $true$ branch edge of $y$ as $executable$ and add it to $Pile$;
                    $y.oldval = y.newval$;
                  end case
                  case $false$:
                    Mark the $false$ branch edge of $y$ as $executable$ and add it to $Pile$;
                    $y.oldval = y.newval$;
                  end case
                  // $\top$ value results due to uninitialized variables which is a
                  // program error and hence is not considered here
                end switch
              end
          end case
        end switch
      end // while
    end
```

FIGURE 3.22 CCP algorithm with flow graph cont'd.

in each visit, the value of a variable can be either lowered or not at all. However, during each visit to a node, lattice values of all the variables are computed. Assuming that we have N nodes, each with an in-degree of at most d_{in}, V variables and E_f edges in the flow graph, each node is visited at most ($d_{in} \times 2 \times 2$) times, because each instruction has at most two operands, each of which can be a variable. The total number of visits to all the nodes is obtained by multiplying this quantity by N. During each visit, ($d_{in} \times V + k_1 \times V + k_2$) computations are performed. Here, the first additive term corresponds to the meet computation, and the second additive term accounts for the time for lattice value copy operations. The time for adding an edge to the pile is charged to the edge removal operation, because every edge is added from a unique node and because each edge that is added to a pile is also definitely removed. Thus, these are constant time operations and are taken care of by the last additive term. Expression evaluation as indicated in Figure 3.20 (note that expressions can contain at most one operator and two operands) and other constant time operations are also taken care of by the last additive term. This entire term can be seen to be $k_3 \times V$, for some constant k_3, if d_{in} is small in magnitude compared with V. This is a reasonable assumption because each node has one instruction and thus N and V are of the same order of magnitude. Because $d_{in} \times N$ is $k_4 \times E$, for some constant k_4, we get the total time needed by this algorithm to be $k_5 \times E \times V$, for some constant k_5, or $O(E \times V)$.

This algorithm is too slow in practice and hence is not used as it is in any compiler. It is used with *d-u* information, which helps in eliminating most of the lattice cell copy operations. The major source of inefficiency is that it uses the flow graph edges for both propagation of values and also for tracking reachable code. The SSA-based algorithm does this more efficiently because it uses different types of edges for each task. We present this version next.

3.3.3 Conditional Constant Propagation Algorithm: Static Single Assignment Version

SSA forms along with extra edges (SSA edges) corresponding to *d-u* information are more efficient for constant propagation than the flow graph. We add an edge from every definition to each of its uses in the SSA form. The new algorithm uses both flow graph and SSA edges and maintains two different work lists, one for each. Flow graph edges are used to keep track of reachable code and SSA edges help in propagation of values. Flow graph edges are added to the flow work list whenever a branch node is symbolically executed or whenever an assignment node has a single successor (all this is subject to value changes as before). SSA edges coming out of a node are added to the SSA work list whenever a change occurs in the value of the assigned variable at the node. This ensures that all uses of a definition are processed whenever a definition changes its lattice value. This algorithm needs only one lattice cell per variable (globally, not on a per node basis) and two lattice cells per node to store expression values. Conditional expressions at branch nodes are handled as before. However, at any join node, the meet operation considers only those predecessors that are marked executable. The SSA-based algorithm is presented in Figures 3.23 to 3.26.

Example 3.1
Consider the program C in Figure **??**. The flow graph and the SSA graph for this program are shown in Figures 3.27 and 3.28. It is clear that at node B_5, a cannot be determined to be a constant by the CCP algorithm using a flow graph, because of the two definitions of a reaching B_5. The problem is due to the meet operation executed at B_5 using both its predecessors while determining the lattice value of a. This problem is avoided by the CCP algorithm using the SSA graph. It uses only those predecessors that have edges to the current node marked as executable.

In this example, only C_4 would be considered while performing the meet operation to compute the new value for a_3, because the other edge (C_3, C_5) is not marked executable. As shown in Figure **??**, the SSA version of the algorithm determines a_3 to be a constant (of value same as that of a_2), and assigns its value to d.

3.3.3.1 Asymptotic Complexity

Each SSA edge can be examined at least once and at most twice. This is because the lattice value of each variable can be lowered only twice and thus each SSA edge can be added to the SSA pile only twice. Each flow graph edge can be examined only once. During a visit to a node, all operations take only constant time. As before, the time for adding an edge (either flow or SSA) to a pile is charged to the edge removal operation. Thus, the total time taken by the algorithm is $O(E_f + E_s)$. Theoretically, E_s can be as large as $O\big(max\{E_f, N\}\big)^2$ [9], and this algorithm can become $O(E_f{}^2)$. However, in practice, E_s is usually $O(max\{E_f, N\})$, and hence the time for constant propagation with the SSA graph is linear in the size of the program ($max\{E_f, N\}$).

3.4 Partial Redundancy Elimination

PRE is a powerful compiler optimization that subsumes global common subexpression elimination and loop-invariant code motion, and can be modified to perform additional code improvements such as strength reduction as well. PRE was originally proposed by Morel and Renvoise [11]. They showed that elimination of redundant computations and movement of invariant computations out of loops can be combined by solving a more general problem, (i.e., the elimination of computations

```
// $\mathcal{G} = (\mathcal{N}, \mathcal{E}_f, \mathcal{E}_s)$ is the SSA graph, with flow edges and SSA edges,
// and $\mathcal{V}$ is the set of variables used in the SSA graph
begin
  $Flowpile = \{(Start \rightarrow n) | (Start \rightarrow n) \in \mathcal{E}_f\}$;
  $SSApile = \emptyset$;
  for all $e \in \mathcal{E}_f$ do $e.executable = false$; end for
  $v.cell$ is the lattice cell associated with the variable $v$
  for all $v \in \mathcal{V}$ do $v.cell = \top$; end for
  // $y.oldval$ and $y.newval$ store the lattice values of expressions at node $y$
  for all $y \in \mathcal{N}$ do
    $y.oldval = \top$; $y.newval = \top$;
  end for
  while ($Flowpile \neq \phi$) or ($SSApile \neq \phi$) do
  begin
    if ($Flowpile \neq \phi$) then
    begin
      $(x, y).$ = remove ($Flowpile$);
      if (not $(x, y).executable$) then
      begin
        $(x, y).executable$ = true;
        if ($\phi$-$present(y)$) then $visit$-$\phi(y)$
          else if ($first$-$time$-$visit(y)$) then $visit$-$expr$ $(y)$;
        // $visit$-$expr$ is called on $y$ only on the first visit
        // to $y$ through a flow edge; subsequently, it is called
        // on $y$ on visits through SSA edges only
        if ($flow$-$outdegree(y)$ == 1) then
          // Only one successor flow edge for y
          $Flowpile = Flowpile \cup \{(y, z) | (y, z) \in \mathcal{E}_f\}$;
    end
    // if the edge is already marked, then do nothing
  end
```

FIGURE 3.23 CCP algorithm with SSA graphs.

performed twice on a given execution path). Such computations were termed as partially redundant. PRE performs insertions and deletions of computations on a flow graph in such a way that after the transformation, each path, in general, contains fewer occurrences of such computations than before. Most compilers of today perform PRE. It is regarded as one of the most important optimizations and it has generated substantial interest in the research community [7, 8, 10, 12, 13, 23 to 29, 33 to 35].

In spite of the benefits, the Morel and Renvoise algorithm has several serious shortcomings. It is not optimal in the sense that it does not eliminate all partial redundancies that exist in a program and it performs redundant code motion. It involves performing bidirectional data flow analysis, which some claim is in general more complex than unidirectional analysis [10]. Knoop, Rüthing and Steffen decomposed the bidirectional structure of the PRE algorithm into a sequence of unidirectional analyses and proposed an optimal solution to the problem with no redundant code motion [10, 28].

Next we present a simple algorithm for partial redundancy elimination [12, 29] based on well-known concepts, namely, availability, anticipability, partial availability and partial anticipability. The algorithm is computationally and lifetime optimal. The algorithm operates on a flow graph with a single statement per node. Modifications to the algorithm to handle basic blocks with multiple statements are reported in [29]. The essential feature of this algorithm is the integration of the notion of safety into the definition of partial availability and partial anticipability. It requires four

```
        if ($SSApile \neq \emptyset$) then
        begin
          $(x, y)$ = remove ($SSApile$);
          if ($\phi$-$present(y)$) then $visit$-$\phi(y)$
            else if ($already$-$visited(y)$) then $visit$-$expr(y)$;
            // A $false$ returned by $already$-$visited$ implies that $y$
            // is not yet reachable through flow edges
        end
    end // Both piles are empty
end

function $\phi$-$present(y)$ // $y \in \mathcal{N}$
begin
    if $y$ is a $\phi$-node then return $true$
      else return $false$
end

function $present$-$\phi(y)$ // $y \in \mathcal{N}$
begin
    $y.newval = \top$;
    // $\|y.instruction.inputs\|$ is the number of parameters of $\phi$-instruction at node $y$
    for $i$ = 1 to $\|y.instruction.inputs\|$ do
      Let $p_i$ be the $i^{th}$ predecessor of $y$;
      if (($p_i$, $y$).$executable$) then
      begin
         Let $a_i = y.instruction.inputs[i]$;
         // $a_i$ is the $i^{th}$ input and $a_i.cell$ is the lattice cell associated with that variable
         $y.newval = y.newval \sqcap a_i.cell$;
      end
    end for
    if ($y.newval < y.instruction.output.cell$) then
    begin
      $y.instruction.output.cell = y.newval$;
      $SSApile = SSApile \cup \{(y, z) \mid (y, z) \in \mathcal{E}_s\}$;
    end
end
```

FIGURE 3.24 CCP algorithm with SSA graphs cont'd.

unidirectional bit vector analyses. A special feature of this algorithm is that it does not require the edge-splitting transformation to be done before the application of the algorithm.

An informal description of the idea behind the algorithm follows. We say an expression is *available* at a point if it has been computed along all paths reaching this point with no changes to its operands since the computation. An expression is said to be *anticipatable* at a point if every path from this point has a computation of that expression with no changes to its operands in between. Partial availability and partial anticipability are weaker properties with the requirement of a computation along "at least one path" as against "all paths" in the case of availability and anticipability.

We say a point is *safe* for an expression if it is either available or anticipatable at that point. Safe partial availability (anticipability) at a point differs from partial availability (anticipability) in that it requires all points be safe on the path along which the computation is partially available (anticipatable). In the example given in Figure 3.29, partial availability at the entry of node 4 is true but safe partial availability at that point is false, because the entry and exit points of node 3 are not safe. In Figure 3.30, safe partial availability at the entry of node 4 is true. We say a computation is

```
function $visit\text{-}expr\ (y)$ // $y \in \mathcal{N}$
being
    Let $input_1 = y.instruction.inputs[1]$;
    Let $input_2 = y.instruction.inputs[2]$;
    If ($input_1.cell == \perp$ or $input_2.cell == \perp$) then
        $y.newval = \perp$
    else if ($input_1.cell == \top$ or $input_2.\ cell == \top$) then
            $y.newval = \top$
        else // evaluate expression at $y$ as in Figure 3.20
            $y.newval = evaluate(y)$;
            it is easy to handle instructions with one operand
    if $y$ is an assignment node then
        if ($y.newval < y.instruction.output.cell$) then
        begin
            $y.instruction.output.cell = y.newval$;
            $SS\,Apile = SS\,Apile \cup \{(y, z) | (y, z) \in \mathcal{E}_s\}$;
        end
    else if $y$ is a branch node then
        begin
            if ($y.newval < y.oldval$) then
            begin
                $y.oldval = y.newval$;
                switch ($y.newval$)
                    case $\perp$: // Both *true* and *false* branches are equally likely
                        $Flowpile = Flowpile \cup \{(y, z) | (y, z) \in \mathcal{E}_f\}$;
                    case true: $Flowpile = Flowpile \cup \{(y, z) | (y, z) \in \mathcal{E}_f$ and
                                        $(y, z)$ is the *true* branch edge at $y\}$;
                    case *false*: $Flowpile = Flowpile \cup \{(y, z) | (y, z) \in \mathcal{E}_f$ and
                        $(y, z)$ is the *false* branch edge at $y\}$;
                end switch
            end
        end
end
```

FIGURE 3.25 CCP algorithm with SSA graphs cont'd.

safe partially redundant in a node, if it is locally anticipatable and is safe partially available at the entry of the node. In Figure 3.30, the computation in node 4 is safe partially redundant.

The basis of the algorithm is to identify safe partially redundant computations and make them totally redundant by the insertion of new computations at proper points. The totally redundant computations after the insertions are then replaced. If $a + b$ is the expression of interest, then by insertion we mean insertion of the computation $h = a + b$, where h is a new variable; replacement means substitution of a computation, such as $x = a + b$, by $x = h$.

Given a control flow graph we compute availability, anticipability, safety, safe partial availability and safe partial anticipability at the entry and exit points of all the nodes in the graph. We then mark all points that satisfy both safe partial availability and safe partial anticipability. Next, we note that the points of insertion for the transformation are the entry points of all nodes containing the computation whose exit point is marked but whose entry point is not, and also all edges whose head is marked but whose tail is not. We also note that replacement points are the nodes containing the computation whose entry or exit point is marked.

Alternatively, if we consider the paths formed by connecting all the adjacent points that are marked, we observe that the points of insertion are the nodes corresponding to the starting points of such paths

```
function first-time-visit(y) // $y \in \mathcal{N}$
// This function is called when processing a flowgraph edge
begin // Check in-coming flowgraph edge of $y$
    for all $e \in \{(x, y) \mid (x, y) \in \mathcal{E}_f\}$
      if e.executable is true for more than one edge $e$
        then return false else return true
    end for
    // At least one in-coming edge will have executable true
    // because the edge through which node $y$ is entered is
    // marked as executable before calling this function
end

function already-visited (y) // $y \in \mathcal{N}$
// This function is called when processing an SSA edge
being // Check in-coming flowgraph edges of $y$
    for all $e \in \{(x, y) \mid (x, y) \in \mathcal{E}_f\}$
      if e.executable is true for at least one edge $e$
        then return true else return false
    end for
end
```

FIGURE 3.26 CCP algorithm with SSA graphs cont'd.

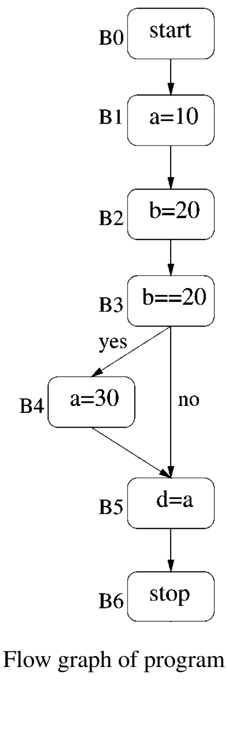


FIGURE 3.27 Flow graph of program C in Figure **??**.

and also the edges that enter junction nodes on these paths. The computations that are to be replaced are the ones appearing on these paths. For the example in Figure 3.31, small circles correspond to marked points. Based on the preceding observation, we see that node 1 and edge (2, 3) are the points of insertion and nodes 1 and 4 are the points of replacement. The graph after the transformation is shown in Figure 3.32.

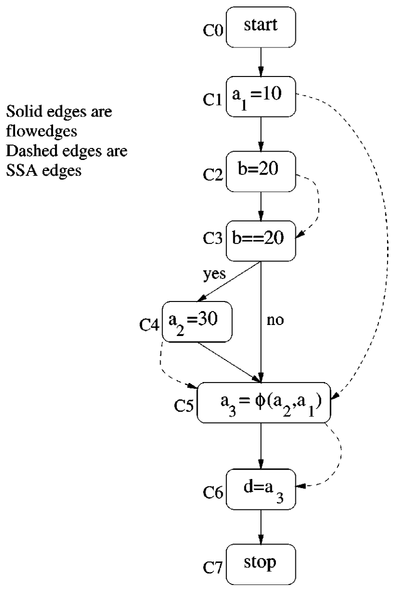


FIGURE 3.28 SSA graph of program C in Figure **??**.

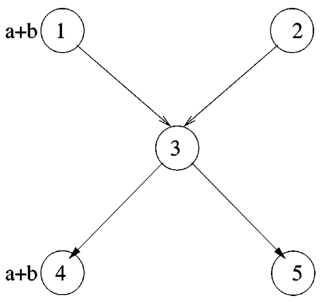


FIGURE 3.29 Node 4: a+b not safe partially available.

3.4.1 Boolean Properties Associated with the Expressions

We assume that the program representation is a flow graph with a single assignment statement or condition per node.

For each expression and each node, Boolean properties are defined. Some of these properties depend only on the statement in the node and are termed *local*. Other properties that depend on statements beyond a node are termed *global*. We develop our algorithm for an arbitrary and fixed expression, and a global algorithm dealing with all expressions simultaneously is the independent combination of all of them.

3.4.1.1 Local Properties

The local properties associated with a node are transparency, availability and anticipability. An expression is said to be transparent in a node i, denoted by $TRANSP_i$, if its operands are not modified by the execution of the statement in node i. The expression in node i is said to be locally

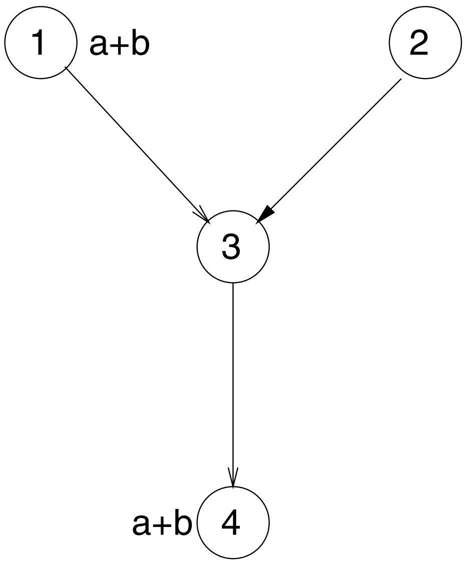


FIGURE 3.30 Node 4: a+b is safe partially redundant.

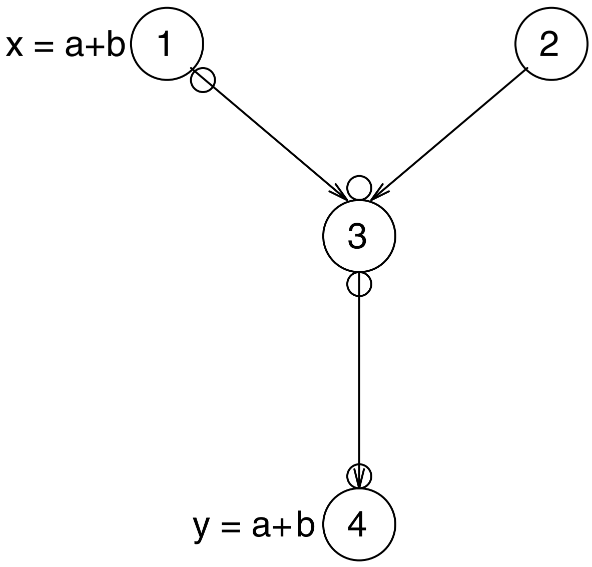


FIGURE 3.31 Before PRE.

anticipatable and is denoted by $ANTLOC_i$. We say the expression in node i is locally available, denoted by $COMP_i$, if the statement in node i does not modify any of its operands.

3.4.1.2 Global Properties

The global properties of our interest are availability anticipability, safe partial availability and safe partial anticipability. We use $AVIN_i$, $ANTIN_i$, $SPAVIN_i$ and $SPANTIN_i$ to denote global availability, anticipability, safe partial availability and safe partial anticipability, respectively, of the

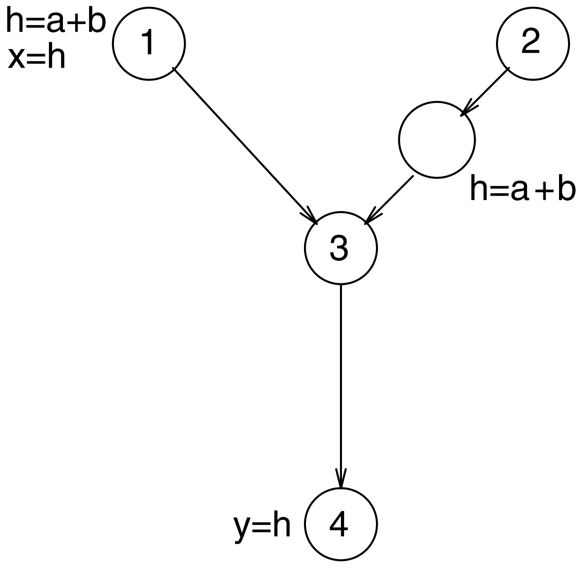


FIGURE 3.32 After PRE.

expression at the entry of node i. Similarly, $AVOUT_i$, $ANTOUT_i$, $SPAVOUT_i$ and $SPANTOUT_i$ are used to denote the same properties at the exit of node i.

We use $SAFEIN_i$ and $SAFEOUT_i$ to denote the fact that it is safe to insert a computation at the entry and exit, respectively, of node i. We say a path $p[m, n]$, from point m to point n in the flow graph, is safe if every point on the path is safe and denote it by $SAFE_{[m,n]}$. Also, we say a path $p[i, j]$, where i and j are nodes, is transparent if every node on the path is transparent and denote it by $TRANSP_{[i,j]}$.

The relation between global and local properties for all nodes of the graph are expressed in terms of systems of Boolean equations. Boolean conjunctions are denoted by . and Π; disjunctions, by + and Σ; and Boolean negation, by $\urcorner$.

3.4.1.2.1 ***Availability.***

An expression is said to be available at a point p if every path from the entry node s to p contains a computation of that expression, and after the last such computation prior to reaching p no modifications occur to its operands.

An expression is available at the entry of a node if it is available on exit from each predecessor of the node. An expression is available at the exit of a node if it is locally available or if it is available at the entry of the node and transparent in this node. That is:

$$AVIN_i = \begin{cases} FALSE & \text{if } i = s \\ \prod_{j \in pred(i)} AVOUT_j & \text{otherwise} \end{cases}$$

$$AVOUT_i = COMP_i + AVIN_i.TRANSP_i$$

3.4.1.2.2 *Anticipability.*

An expression is said to be anticipatable at a point p if every path from p to the exit node e contains a computation of that expression, and after p prior to reaching the first such computation no modifications occur to its operands.

An expression is anticipatable at the exit of a node if it is anticipatable at the entry of each successor of the node. An expression is anticipatable at the entry of a node if it is locally anticipatable or it is anticipatable at the exit of the node and transparent in this node. That is:

$$ANTOUT_i = \begin{cases} FALSE & \text{if } i = e \\ \prod_{j \in succ(i)} ANTIN_j & \text{otherwise} \end{cases}$$

$$ANTIN_i = ANTLOC_i + ANTOUT_i . TRANSP_i$$

3.4.1.2.3 *Safety.*

A point p is considered to be safe for an expression if the insertion of a computation of that expression at p does not introduce a new value on any path through p. Alternatively, a point p is safe if the expression is either available or anticipatable at that point. That is:

$$SAFEIN_i = AVIN_i + ANTIN_i$$

$$SAFEOUT_i = AVOUT_i + ANTOUT_i$$

3.4.1.2.4 *Safe Partial Availability.*

We say an expression is safe partially available at a point n, if at least one path exists from the entry node s to n that contains a computation of that expression, and if after the last such computation on this path prior to reaching n, say at node m, no modifications occur to its operands, with the path from the exit of node m to n safe. That is:

$$SPAVIN_i = \begin{cases} FALSE & \text{if } i = s \text{ or } \neg SAFEIN_i \\ \sum_{j \in pred(i)} SPAVOUT_j & \text{otherwise} \end{cases}$$

$$SPAVOUT_i = \begin{cases} FALSE & \text{if } \neg SAFEOUT_i \\ COMP_i + SPAVIN_i . TRANSP_i & \text{otherwise} \end{cases}$$

3.4.1.2.5 *Safe Partial Anticipability.*

We say an expression is safe partially anticipatable at a point m, if at least one path exists from m to the exit node e that contains a computation of that expression, and if after m prior to reaching the first such computation on this path, say at node n, no modifications of its operands occur, with the path from m to the entry of node n safe. That is:

$$SPANTOUT_i = \begin{cases} FALSE & \text{if } i = e \text{ or } \neg SAFEOUT_i \\ \sum_{j \in succ(i)} SPANTIN_j & \text{otherwise} \end{cases}$$

$$SPANTIN_i = \begin{cases} FALSE & \text{if } \neg SAFEIN_i \\ ANTLOC_i + SPANTOUT_i . TRANSP_i & \text{otherwise} \end{cases}$$

The iterative process for the resolution of the systems of Boolean equations can give several solutions depending on the initialization of the unknowns. For the preceding Boolean systems, the required solution is the largest one for the system involving the conjunction operator Π and the initialization value is *TRUE* for all the unknowns in this case. For the systems involving the disjunction operator Σ, the required solution is the smallest one and the initialization value is *FALSE* for all unknowns.

We next give the formal definitions of safe partial redundancy, total redundancy and isolatedness.

3.4.1.2.6 *Safe Partial Redundancy.*

A computation of an expression is safe partially redundant in a node i, denoted $SPREDUND_i$, if it is locally anticipatable and is safe partially available at the entry of node i. That is:

$$SPREDUND_i = ANTLOC_i.SPAVIN_i$$

3.4.1.2.7 *Total Redundancy.*

We say an expression is totally redundant, or simply redundant, in a node i, denoted by $REDUND_i$, if it is locally anticipatable and is available at the entry of node i. That is:

$$REDUND_i = ANTLOC_i.AVIN_i$$

3.4.1.2.8 *Isolatedness.*

An expression in node i is said to be isolated, denoted $ISOLATED_i$, if the expression is not safe partially available at the entry of node i and is either not safe partially anticipatable at the exit of node i or not locally available. That is:

$$ISOLATED_i = ANTLOC_i.\neg SPAVIN_i.(\neg COMP_i + \neg SPANTOUT_i)$$

3.4.2 Partial Redundancy Elimination Algorithm

The basic principle of the algorithm is to introduce new computations of the expression at points of the program chosen in such a way that the safe partially redundant computations become totally redundant. As in [28], our algorithm introduces a new auxiliary variable h for the expression concerned, inserts assignments of the form $h := expr$ at some program points and replaces some of the original computations of the expression by h, to achieve the transformation. The algorithm computes the points at which these insertions and replacements are to be done. We denote the insertion at the entry of node i by $INSERT_i$, insertion on edge (i, j) by $INSERT_{(i,j)}$ and replacement in node i by $REPLACE_i$.

The steps of the algorithm are as follows:

1. Compute *AVIN/AVOUT* and *ANTIN/ANTOUT* for all nodes.
2. Compute *SAFEIN/SAFEOUT* for all nodes.
3. Compute *SPAVIN/SPAVOUT* and *SPANTIN/SPANTOUT* for all nodes.
4. Compute points of insertions and replacements $INSERT_i$, $INSERT_{(i,j)}$ and $REPLACE_i$.

The points of insertions and replacements are computed using the following equations:

$$INSERT_i = COMP_i.\neg SPAVIN_i.SPANTOUT_i$$
$$INSERT_{(i,j)} = \neg SPAVOUT_i.SPAVIN_j.SPANTIN_j$$
$$REPLACE_i = ANTLOC_i.SPAVIN_i + COMP_i.SPANTOUT_i$$

Placement of a new computation on an edge means introducing a new node. Therefore, edge placements should be avoided as far as possible. We reduce the number of insertions on edges by the following strategy. For a node i:

If $(\prod_{j \in succ(i)}(INSERT_{(i,j)} + INSERT_j.(|pred(j)| = 1)))$ then do a single insertion at the exit of node i, instead of $INSERT_{(i,j)}$'s and $INSERT_j$'s in the preceding term.

The formal proofs of correctness and optimality of the algorithm are given in [12].

Example

We consider the same example as in [11]; refer to Figures 3.33 and 3.34.

Local boolean properties are:

$$ANTLOC = \{6, 7, 8, 9\}$$
$$COMP = \{6, 7, 8, 9\}$$

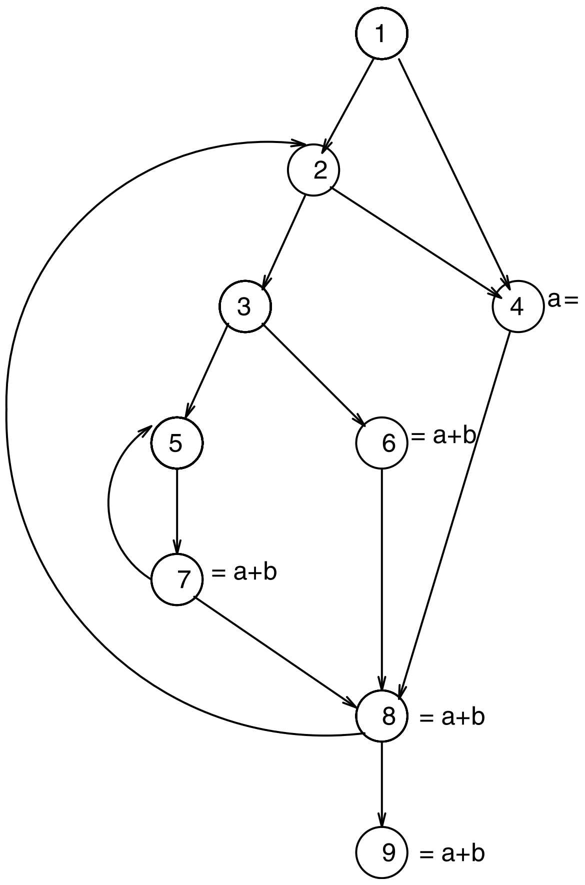


FIGURE 3.33 Initial program.

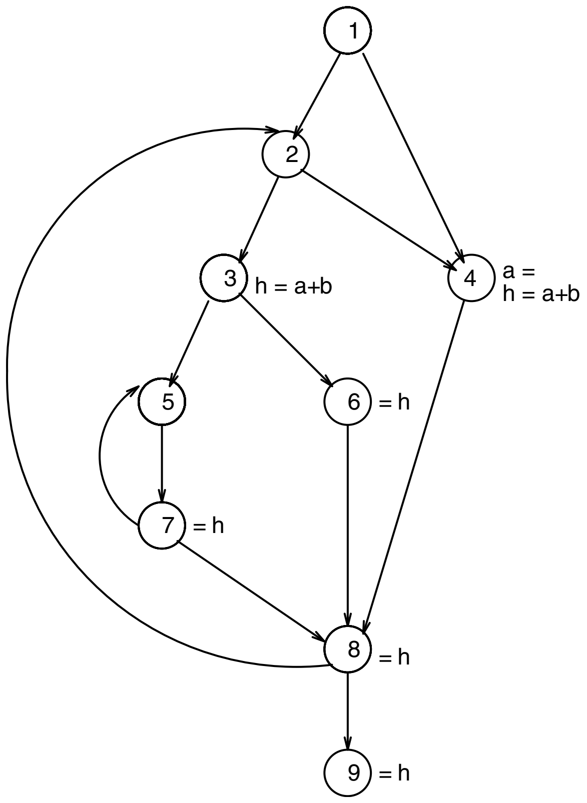


FIGURE 3.34 Transformed program.

Global Boolean properties are:

$$
\begin{aligned}
AVIN &= \{9\} \\
AVOUT &= \{6, 7, 8, 9\} \\
ANTIN &= \{3, 5, 6, 7, 8, 9\} \\
ANTOUT &= \{3, 4, 5, 6, 7\} \\
SAFEIN &= \{3, 5, 6, 7, 8, 9\} \\
SAFEOUT &= \{3, 4, 5, 6, 7, 8, 9\} \\
SPAVIN &= \{5, 7, 8, 9\} \\
SPAVOUT &= \{5, 6, 7, 8, 9\} \\
SPANTIN &= \{3, 5, 6, 7, 8, 9\} \\
SPANTOUT &= \{3, 4, 5, 6, 7, 8\}
\end{aligned}
$$

Insertions and replacements are:

$$\begin{aligned} INSERT_i &= \{6\} \\ INSERT_{(i,j)} &= \{(3,5), (4,8)\} \\ REPLACE_i &= \{6, 7, 8, 9\} \end{aligned}$$

Because $\prod_{j \in succ(i)} (INSERT_{(i,j)} + INSERT_j) = TRUE$ for $i = 3$, we insert a new computation at the exit of node 3, instead of $INSERT_6$ and $INSERT_{(3,5)}$. Similarly, we insert a new computation at the exit of node 4, instead of $INSERT_{(4,8)}$. Computations in nodes 6, 7, 8 and 9 are replaced. The final solution is insertions at the exit of nodes 3 and 4 and replacements in nodes 6, 7, 8 and 9. The transformed program is given in Figure 3.34.

3.5 Value Numbering

Value numbering is one of the oldest and still a very effective technique that is used for performing several optimizations in compilers [3, 8, 15, 16, 19, 30, 31]. It operates on flow graphs with basic blocks (having perhaps more than one instruction) and it has been extended to operate on SSA graphs. The central idea in this method is to assign numbers (called *value numbers*) to expressions in such a way that two expressions receive the same number if the compiler can prove that they are equal for all possible program inputs. This technique is useful in finding redundant computations and fold constants. Even though it was originally proposed as a local optimization technique applicable to basic blocks, it is now available in its global form. We assume that expressions in basic blocks have at most one operator (except for ϕ-functions in SSA forms). The two principal techniques to prove equivalence of expressions are hashing and partitioning.

The hashing scheme is simple and easy to understand. It uses a hashing function that combines the operator and the value numbers of operands of an expression (assume non-ϕ, for the present) contained in the instruction and produces a unique value number for the expression. If this number is already contained in the hash table, then the name corresponding to this existing value number refers to an earlier computation in the block. This name holds the same value as the expression for all inputs, and hence the expression can be replaced by the name. Any operator with known constant values is evaluated and the resulting constant value is used to replace any subsequent references. It is easy to adapt this algorithm to apply commutativity and simple algebraic identities without increasing its complexity.

The partitioning method operates on SSA forms and uses the technique of deterministic finite automation (DFA) minimization [4] to partition the values into congruence classes. Two expressions are congruent to each other, if their operators are identical and their operands are congruent to each other. Two constants are congruent to each other if their values are the same. Two variables are congruent to each other if the computations defining them are congruent to each other. The process starts by putting all expressions with the same operator into the same class and then refining the classes based on the equivalence of operands. Partitioning-based technique is described in [7, 15, 16, 20]. Partitioning-based methods are global techniques that operate on the whole SSA graph, unlike the hash-based methods. We provide a flavor of the partitioning method through an example. See Figures 3.35 and 3.36.

The initial and final value partitions are as shown in Figure 3.35. To begin with, variables that have been assigned constants with the same value are put in the same partition (P_1, and P_5), and so are variables assigned expressions with the same operator (P_3); ϕ-instructions of the same block are also bundled together (P_2, and P_4). Next, we start examining instructions in each partition pairwise, and split the partition if the operands of the instructions are not in the same partition. For example, we split P_3 into Q_3 and Q_4, because $x_2 + 2$ is not equivalent to $x_2 + 3$ (constants 2 and 3 are

Initial Partitions	Final Partitions
$P_1 = \{x_1, y_1\}$	$Q_1 = \{x_1, y_1\}$
$P_2 = \{x_2, y_2\}$	$Q_2 = \{x_2, y_2\}$
$P_3 = \{x_5, y_5, x_6, y_6\}$	$Q_3 = \{x_5, y_5\}$
$P_4 = \{x_4, y_4\}$	$Q_4 = \{x_6, y_6\}$
$P_5 = \{x_3, y_3\}$	$Q_5 = \{x_4, y_4\}$
	$Q_6 = \{x_3, y_3\}$

FIGURE 3.35 Initial and final value partitions for the SSA graph in Figure 3.36.

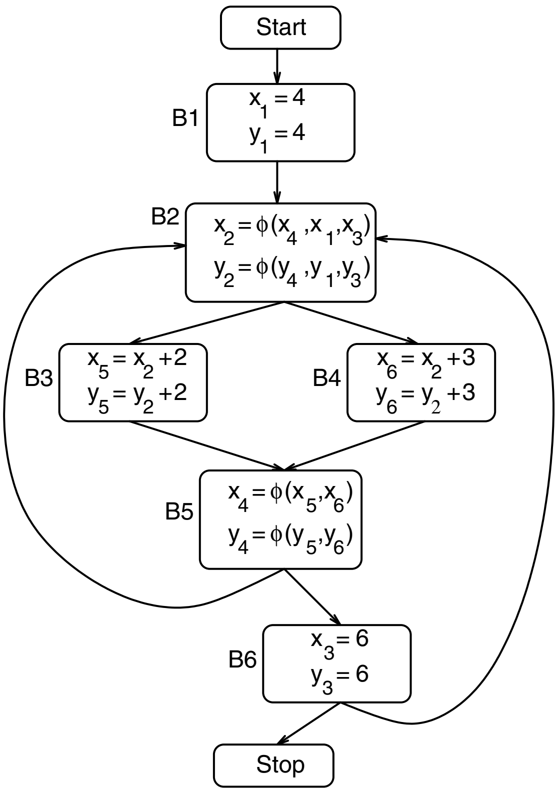


FIGURE 3.36 SSA graph for value-numbering with the partitioning technique.

not equivalent). Partitioning technique splits partitions only when necessary, whereas the hashing technique combines partitions whenever equivalence is found. Continuing the example, we cannot split x_2 and y_2 into different classes because their corresponding inputs, namely, (x_1, y_1), (x_3, y_3) and (x_4, y_4), belong to identical equivalent classes, P_1, P_5 and P_4, respectively. A similar argument holds for x_4 and y_4 as well. After partitioning, this information may be used for several optimizations such as common subexpression elimination and invariant code motion, as explained in [16].

Even though programs are available on which partitioning is more effective than hashing (and also vice versa), partitioning is not as effective as hashing on real-life programs [15]. Hashing is the

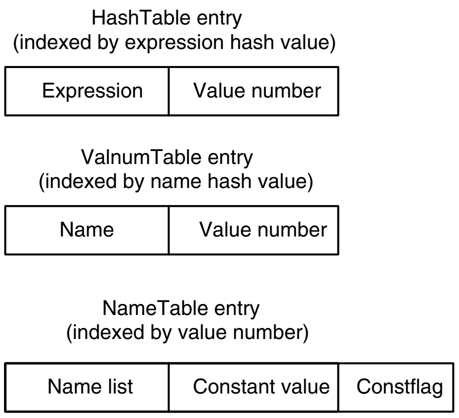


FIGURE 3.37 Data structures for value-numbering with basic blocks.

method of choice for most compilers and hence we describe hashing-based value numbering in detail in the following sections. Our presentation is based on the techniques described in [7, 8, 15, 31].

3.5.1 Hashing-Based Value Numbering: Basic Block Version

We first describe the basic block version of value numbering based on hashing, because this is the basis for the global versions. The algorithm uses three tables, namely, *HashTable, ValnumTable* and *NameTable*. *HashTable* is indexed using the output of a hashing function that takes an expression (with value numbers of its operands replacing the operands themselves) as a parameter. It stores expressions and their value numbers. *ValnumTable* is indexed using another hashing function that takes a variable name (a string) as a parameter. It stores variable names and their value numbers.

NameTable is indexed using a value number, and it stores the variable names (a list) corresponding to this value number, and also the constant value associated with this variable, if any, indicated by the field *Constflag*. The first name in the name list always corresponds to the first definition for the concerned value number. This first name is extracted using the function *Defining_var* and is used to replace other name occurrences in the basic block with the same value number.

The example given later makes this clear. A counter, *valcounter*, is used to generate new value numbers by simple incrementing. The array B of size n stores the n instructions of a basic block. Any good hashing scheme presented in books on data structures and algorithms can be used for the hashing functions [5, 6]. The value-numbering algorithm is presented in Figures 3.38 to 3.42. The data structures used in value numbering are shown in Figure 3.37.

Example of Value Numbering

The high-level language statements and the intermediate code in the form of quadruples are shown in Figure 3.43, along with the final transformed code. Let us run through the instructions in the basic block one at a time and observe how value numbering works on them. Processing the first instruction, $a = 10$, results in a entered into *ValnumTable* and *NameTable* with a value number, say 1, along with its constant value. When handling instruction 2, a is looked up in the *ValnumTable* and its constant value is fetched from *NameTable*. Constant folding results in b getting a constant value of 40; b is entered into *ValnumTable* and *NameTable* with a new value number, say 2, and also its constant value of 40.

Then i and j are entered into the two tables with new value numbers, when processing instruction 3. The expression $i * j$ is searched in *HashTable* (with value numbers of i and j used during hashing). Because it is not found there, it is entered into *HashTable* with a new value number. The name $t1$ is

```
function $CheckAndInsert$ ($ValnumTable, x, s$)
// $x$ is a variable name and $s$ is an assignment statement
begin
  if (not ($found\_VT$ ($ValnumTable, x$)) then // $x$ does not have a value number
    begin
      $vn\_new$ = $valcounter$++; // Generate a new value number
      $add\_VT$ ($ValnumTable, vn\_new, x$);
    end
  else begin
        $vn = Find\_valnum$ ($ValnumTable, x$);
        if ($NameTable[vn].Constflag$) then // $vn$ corresponds to a constant
          replace $x$ in $s$ by $NameTable$ [$vn$].$constval$
        else begin // Find the first variable with this value number
              $first\_var = Defining\_var$ ($NameTable, vn$);
              replace $x$ in $s$ by $first\_var$;
            end
      end
end
function $value\text{-}number$ ($B$) // $B$ is a basic block
begin
  for $i = 1$ to $n$ do // process all instructions in the basic block
    Let $s$ be the intruction at $B[i]$
    if $s$ is an assignment statement then
      begin
        let $s$ be of the form $a = b\ op\ c$;
        if $b$ is not a constant then
          $CheckAndInsert$ ($ValnumTable, b, s$);
        if $c$ is not a constant then
          $CheckAndInsert$ ($ValnumTable, c, s$);
        Simplify the new expression in $s$ by constant folding
        and replace it by the result
        // Note that constant folding may not be possible in the above step
        // if one or both operands in $s$ are not constants
        // Algebraic laws may be applied at this point in simplification
        let the new $s$ be $a = \langle expr \rangle$;
```

FIGURE 3.38 The value-numbering algorithm: Basic block version.

also entered into *ValnumTable* and *NameTable* with the same value number. The constant value of b, namely, 40, replaces b in instruction 4, gets into *HashTable* with a new value number and c enters the tables with this value number. Instructions 5 to 7 are processed in a similar way, with $t2$ becoming a constant of value 150 and this value of 150 replacing $t2$ in instruction 6. During the processing of instruction 8, e is replaced by i, and the expression $i * j$ is found in *HashTable*. *NameTable* gives the name of the first definition corresponding to it as $t1$. Therefore, $t1$ replaces $t3$ in instruction 10. Instructions 5 and 8 can be deleted, because the holding variables $t2$ and $t3$ have already been replaced by 150 and $t1$, respectively, in instructions referencing them.

3.5.2 Accommodating Extended Basic Blocks

An extended basic block is a sequence of basic blocks, $B_1, B_2, \ldots, B_k$, such that B_i is the unique predecessor of B_{i+1} ($1 \leq i < k$), and B_1 is either the start block or has no unique predecessor. Two or more extended basic blocks may share their first few blocks and this sharing is represented well using trees, as shown in Figure 3.44. Hashing-based value numbering applicable to basic blocks may

```
if ($\langle expr \rangle$ is a constant) then
  begin
    if ($found\_VT$ ($ValnumTable$, $a$)) then // $a$ is already present
      begin
        // remove existing $a$ and value number
        $vn = Find\_valnum$ ($ValnumTable$, $a$);
        $remove\_NT$ ($Nametable$, $vn$, $a$);
        $remove\_VT$ ($ValnumTable$, $vn$, $a$);
      end
    // Add the name $a$, its new value number and
    // its new constant value to the tables
    $vn\_new = valcounter + +$;
    $add\_NT$ ($NameTable$, $vn\_new$, $a$);
    $add\_NT\_Const$ ($NameTable$, $vn\_new$, $\langle expr \rangle$);
    $add\_VT$ ($ValnumTable$, $vn\_new$, $a$);
  end
else // $\langle expr \rangle$ is not a constant
  begin
    if (not $found\_HT$ ($HashTable$, $\langle expr \rangle$)) then
    // $\langle expr \rangle$ does not have a value number
     begin
        $vn\_new = valcounter + +$;
        $add\_HT$ ($HashTable$, $vn\_new$, $\langle expr \rangle$)
        if ($found\_VT$ ($ValnumTable$, $a$)) then // $a$ is already present
          begin
            // remove existing $a$ and value number
            $vn = Find\_valnum$ ($ValnumTable$, $a$);
            $remove\_NT$ ($NameTable$, $vn$, $a$);
            $remove\_VT$ ($ValnumTable$, $vn$, $a$);
          end
        // Add the name $a$ and the value number to the tables
        $add\_NT$ ($NameTable$, $vn\_new$, $a$);
        $add\_VT$ ($ValnumTable$, $vn\_new$, $a$);
     end
```

FIGURE 3.39 The value-numbering algorithm: Cont'd.

be used with extended basic blocks also with some changes. When the constituent basic blocks of an extended block are not shared by any other extended block, the matter is simple. We just process the instructions in the basic blocks of the extended block as if they belonged to a single basic block.

Sharing of blocks requires a scope structure in all the three tables, namely, *HashTable, ValnumTable* and *NameTable*. The modified algorithm is shown in Figure 3.45. We do a preorder traversal on the trees of extended basic blocks. Preorder traversal ensures that shared blocks are processed only once. Managing scoped tables involves removal of new table entries when the function *visit-ebb-tree* exits and also restoration of old entries that might have been invalidated by the new entries. For example, when a name a is redefined in a new block, the entry for name a is removed from the *NameTable* against its old value number (obtained from *ValnumTable*), before entering the name a against the new value number. The old entry must be restored when processing of the new block is completed. This indeed introduces considerable overheads. Thus, this version of the value-numbering algorithm may run slowly compared with the basic block version. As an example, let us consider the tree $T2$ in Figure 3.44. We start with B_2, process it completely and then call *visit-ebb-tree* on B_3, followed by a call on B_5. After processing B_5, the entries in tables made during the processing of B_5 are removed from the tables, and the changed entries are restored to their old values before a return to B_3. The

```
            else // $\langle expr \rangle$ is present in the hashtable and has a value number
              begin
                $expr\_vn = Find\_Expr\_valnum$ ($Hash\ Table$, $\langle expr \rangle$);
                if ($var\_found$ ($Name\ Table$, $expr\_vn$)) then
                  // $\langle expr \rangle$ is not hanging alone, it is attached to a variable
                  if ($is\_one\_of\_vars$ ($Name\ Table$, $expr\_vn$, $a$)) then
                    // $a$ is already attached to $\langle expr \rangle$, so $s$ is redundant
                    delete $s$
                  else
                    begin // Get the first defining variable for $\langle expr \rangle$
                      $old\_var = Defining\_var$ ($NameTable$, $expr\_vn$);
                      replace $\langle expr \rangle$ in $s$ by $old\_var$;
                      $add\_NT$ ($NameTable$, $expr\_vn$, $a$);
                    end
                else // $\langle expr \rangle$ is present in the hashtable but is not attached to
                    // any variable; reuse the value number, and the expression;
                    // this saves some time
                    $add\_NT$ ($NameTable$, $expr\_vn$, $a$);
              end
            end // end of $\langle expr \rangle$ not a constant
          end // end of assignment statement processing
        end for // end of for statement
    // Assignment statements of the form $a = op\, b$, and $a = b$ can be processed
    // in a similar way. A conditional statement of the form
    // 'if $\langle expr \rangle$ then goto $L$', can be replaced by two statements
    // S1: '$\langle newtemp \rangle = \langle expr \rangle$' and
    // S2: 'if $\langle newtemp \rangle$ then goto $L$'. $S_1$ can be processed as before
    // and $S_2$ needs no special processing, $\langle newtemp \rangle$ may need replacement
    // by another variable, if $\langle expr \rangle$ is already computed and available.
    end
```

FIGURE 3.40 The value-numbering algorithm: Cont'd.

function is now called on B_6; after processing the corresponding entries are deleted and the changes are undone. Similar processing ensues on blocks B_3,B_2 and B_4. Note that the shared blocks B_2 and B_3 are processed only once.

3.5.3 Value Numbering with Hashing and Static Single Assignment Forms

The principle of scoped tables is used here as well, but not with extended basic blocks. We use dominator trees and a reverse postorder over the SSA flow graph to guide traversals over the dominator trees of flow graphs. One of the reasons for choosing a reverse postorder over the SSA flow graph is to ensure that all the predecessors of a block are processed before the block is processed (this is obviously not possible in the presence of loops, with the effect of loops explained later). The dominator tree for a flow graph is shown in Figure 3.46, along with a depth-first tree for the same flow graph and traversal orders on these. The scope of the tables now extends along the paths in the dominator tree, instead of paths in the tree representation for extended basic blocks. The SSA form used here does not need any additional edges in the form of *u-d* or *d-u* information. The structures of the tables used in this algorithm are shown in Figure 3.47.

The SSA form introduces some specialties in instruction processing. This form has no "hanging expressions" (expressions not attached to any variable), because no variable is redefined and no definition is killed. This also means that there is no need to restore any old entries in tables when

```
// ValnumTable is always searched by hashing a name
// NameTable is always indexed using a value number
// HashTable is always searched by hashing an expression

function found_VT (ValnumTable, x)
    returns true if the name x is found in ValnumTable
    otherwise returns false
function add_VT (ValnumTable, vn_new, x)
    Insert name x with value number vn_new into ValnumTable
function remove_VT (ValnumTable, x)
    Removes the name x from ValnumTable
function Find_valnum (ValnumTable, x)
    returns value number of x in ValnumTable
function Defining_var (NameTable, vn)
    returns the first name in the namelist field of the entry at vn in the NameTable
function add_NT (NameTable, vn_new, x)
    Inserts name x into the name list at value number vn_new into NameTable
function remove_NT (NameTable, vn, a)
    Removes the name x from the namelist at vn in NameTable
```

FIGURE 3.41 The value-numbering algorithm: support functions.

```
function add_NT_Const (NameTable, vn_new, constval)
    Inserts the constant constval at vn_new in NameTable
function found_HT (HashTable, ⟨expr⟩))
    returns true if the expression ⟨expr⟩ is found in Hash Table
    otherwise returns false
function add_HT (HashTable, vn_new, ⟨expr⟩)
    Inserts the expression ⟨expr⟩ with value number vn_new into HashTable
function Find_Expr_valnum (HashTable, ⟨expr⟩)
    returns value number of the expression ⟨expr⟩
function var_found (NameTable, vn)
    returns true if the namelist at value number vn in NameTable is not empty
    otherwise returns false
function is_one_of_vars (NameTable, vn, x)
    returns true if the name x is found in the namelist at value number vn in NameTable
    otherwise returns false
```

FIGURE 3.42 The value-numbering algorithm: more support functions.

the processing of a block is completed. It is enough if the new entries are deleted. This reduces the overheads in managing scoped tables. We do not need the *NameTable* here.

A computation C (say of the form $a = b\ op\ c$) becomes redundant if $b\ op\ c$ has already been computed in one of the nodes dominating the node containing C. If the defining variable of the dominating computation is x, then C can be deleted and all occurrences of a can be replaced by x. This fact is recorded in the *ValnumTable* by entering a, and value number of x into it, and setting the field *replacing-variable* to x. From now on, whenever an expression involving a is to be processed, we search for a in the *ValnumTable* and get its *replacing-variable* field (which contains x). This replaces a in the expression processed. While processing an instruction of the form $p = q$, we take the *replacing-variable* of q (say r), and enter it along with p in the *ValnumTable*. This ensures that any future references of p are also replaced by r.

We maintain a global *ValnumTable* and a scoped *HashTable* as before, but over the dominator tree (*ValnumTable* is not scoped). For example in Figure 3.46, a computation in block B_5 can be

HLL Program	Quadruples before Value-Numbering	Quadruples after Value-Numbering
$a = 10$	1. $a = 10$	1. $a = 10$
$b = 4 * a$	2. $b = 4 * a$	2. $b = 40$
$\mathrm{c} = i * j + b$	3. $t1 = i * j$	3. $t1 = i * j$
$\mathrm{d} = 15 * a * c$	4. $\mathrm{c} = t1 + b$	4. $\mathrm{c} = t1 + 40$
$\mathrm{e} = i$	5. $t2 = 15 * a$	5. $t2 = 150$
$\mathrm{c} = e * j + i * a$	6. $\mathrm{d} = t2 * c$	6. $\mathrm{d} = 150 * c$
	7. $e = i$	7. $e = i$
	8. $t3 = e * j$	8. $t3 = i * j$
	9. $t4 = i * a$	9. $t4 = i * 10$
	10. $\mathrm{c} = t3 + t4$	10. $\mathrm{c} = t1 + t4$
		(Instructions 5 and 8 can be deleted)

FIGURE 3.43 Example of value-numbering.

replaced by a computation in block B_1 or B_4, because the tables for $B1$, $B4$ and B_5 are together while processing B_5. It is possible that no such previous computations are found in the *HashTable*, in which case, we generate a new value number and store the expression in the computation along with the new value number in the *HashTable*. The defining variable of the computation is also stored in the global *ValnumTable* along with the new value number. A global table is needed for *ValnumTable* while processing ϕ-instructions.

Processing ϕ-instructions is slightly more complicated. A ϕ-instruction receives inputs from several variables along different predecessors of a block. The inputs do not need to be defined in the immediate predecessors or dominators of the current block. They may in fact be defined in any block that has a control path to the current block. For example, in Figure 3.48, while processing block B_9, we need definitions of a_2, a_6, etc., which are not in the same scope as B_9 (over the dominator tree). However, each incoming arc corresponds to exactly one input parameter of the ϕ-instruction. This global nature of inputs requires a global *ValnumTable*, containing all the variable names in the SSA graph.

During the processing of a ϕ-instruction, it is possible that one or more of its inputs are not yet defined because the corresponding definitions reach the block through back arcs. Such entries are not found in the *ValnumTable*. In such a case, we simply assign a new value number to the ϕ-expression and record the defining variable of the ϕ-instruction along with this new value number in the global *ValnumTable*. The ϕ-expression is also stored with the new value number in the scoped *HashTable*. It may not be out of place to mention here that value numbering based on *partitioning* can handle some of the cases where definitions reach through back arcs. For details, refer to [7, 15] and the example discussed later in this section.

If all the input variables are found in the global *ValnumTable*, then we first replace the inputs of the ϕ-instruction by the entries found in the *ValnumTable*, and then go on to check whether the ϕ-expression is either meaningless or redundant. If these conditions are not true, then a new value number is generated and the simplified ϕ-expression and its defining variable are entered into the tables as explained before.

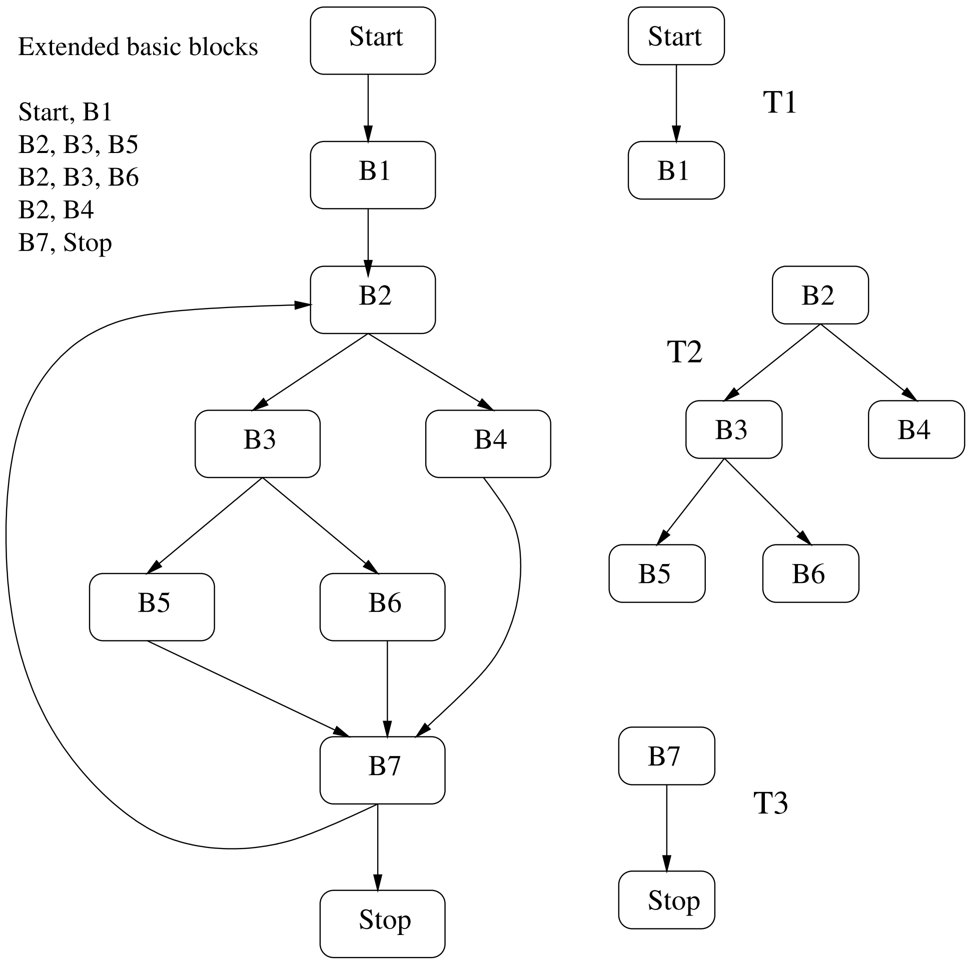


FIGURE 3.44 Extended basic blocks and their trees.

A ϕ-expression is meaningless, if all its inputs are identical. In such a case, the corresponding ϕ-instruction can be deleted and all occurrences of the defining variable of the ϕ-instruction can be replaced by the input parameter. This fact is recorded in the global *ValnumTable* along with the value number of the input parameter. For example, the instruction $u = \phi(a, b, c)$ may become $u = \phi(x, x, x)$, if a, b and c are all equivalent to x, as determined from the entries in *ValnumTable*. In such a case, we delete the instruction and record u in *ValnumTable* along with x and its value number, so that future occurrences of u can be replaced by x.

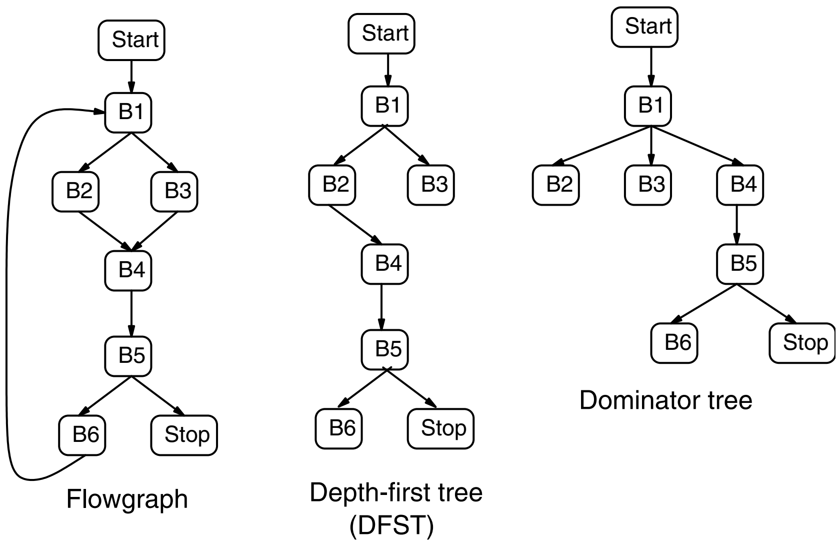
A ϕ-expression is redundant, if another ϕ-expression in the same basic block has exactly the same parameters. Note that we cannot use another ϕ-expression from a dominating block here because the control conditions for the blocks may be different. For example, the blocks B_1 and B_4 in Figure 3.46 may have the same ϕ-expression, but they may yield different values at runtime depending on the control flow. *HashTable* can be used to check redundancy of a ϕ-expression in the block. If a ϕ-expression is indeed redundant, then the corresponding ϕ-instruction can be deleted and all occurrences of the defining variable in the redundant ϕ-instruction can be replaced by the earlier nonredundant one. This information is recorded in the tables.

Figures 3.48 and 3.49 show an SSA graph before and after value numbering. Figure 3.50 shows the dominator tree and a reverse postorder for the same SSA graph. Block B_8 has a meaningless ϕ-instruction and block B_9 has a redundant ϕ-instruction. The instructions such as $a_2 = b_1$ in block

```
Function visit-ebb-tree(e) // e is node in the tree
begin
   // From now on, the new names will be entered with a new scope into the tables.
   // When searching the tables, we always search beginning with the current scope
   // and move to enclosing scopes. This is similar to the processing involved with
   // symbol tables for lexically scoped languages
   value-number (e.B);
   // Process the block e.B using the basic block version of the algorithm
   if (e.left $\neq$ null) then visit-ebb-tree(e.left);
   if (e.right $\neq$ null) then visit-ebb-tree(e.right);
   remove entries for new scope from all the tables
   and undo the changes in the tables of enclosing scopes;
end

begin // main calling loop
   for each tree t do visit-ebb-tree(t);
   // t is a tree representing an extended basic block
end
```

FIGURE 3.45 Value-numbering with extended basic blocks.



Postorder on DFST: B6, Stop, B5, B4, B2, B3, B1, Start
Reverse postorder on DFST: Start, B1, B3, B2, B4, B5, Stop, B6
Visit order on dominator tree is same as reverse postorder

FIGURE 3.46 Flowchart, DFST and dominator tree.

B_2 can perhaps be deleted, but are shown in Figure 3.49 for the sake of explaining the functioning of the algorithm. The SSA graph in Figure 3.48 has not been obtained by translation from a flow graph; it has been constructed to demonstrate the features of the algorithm.

As another example, consider the SSA graph shown in Figure 3.36. Hashing-based techniques discover fewer equivalences, shown as follows:

$$\{x_1, y_1\}, \{x_3, y_3\}, \{x_2\}, \{x_4\}, \{x_5\}, \{x_6\}, \{y_2\}, \{y_4\}, \{y_5\}, \{y_6\}$$

HashTable entry
(indexed by expression hash value)

Expression	Value number	Parameters for ϕ-function	Defining variable

ValnumTable
(indexed by name hash value)

Variable name	Value number	Constant value	Replacing variable

FIGURE 3.47 Data structures for value-numbering with SSA forms.

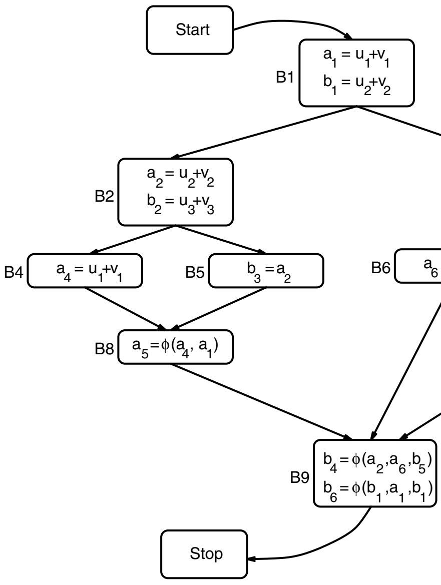


FIGURE 3.48 Example of value-numbering with SSA forms.

This is partly because of the back arcs. The values of x_2 and y_2 always are assigned different value numbers, because x_3, x_4, y_3 and y_4 reach the block B_2 through back edges and their corresponding instructions would not have been processed (present in blocks B_5 and B_6) while block B_2 is processed. Values x_5 and y_5 are not assigned the same value number because x_2 and y_2 do not have the same value number. The same is true of x_6 and y_6 also.

3.6 Conclusions and Future Directions

In this chapter, we discuss in detail several algorithms on both the traditional flow graph and also the more recently introduced SSA form. These algorithms demonstrate how the features of SSA graphs such as ϕ-functions, single assignment to variables and SSA edges (corresponding to *d-u* information) assist in making the algorithms global, faster and more versatile. For example, conditional constant

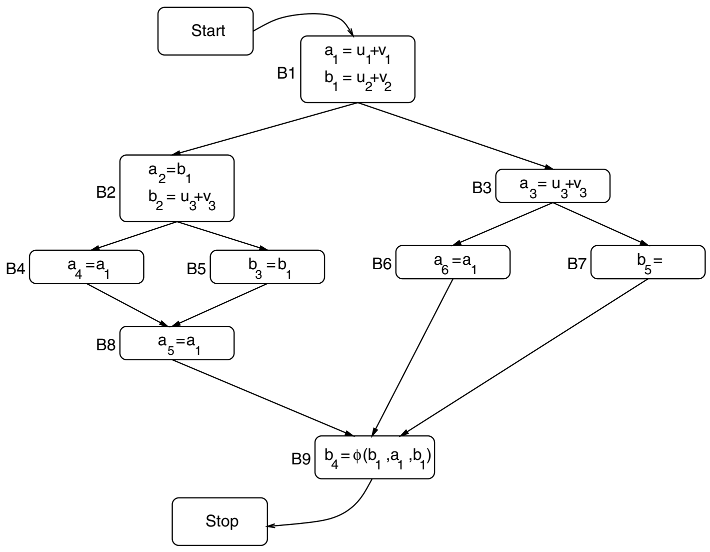
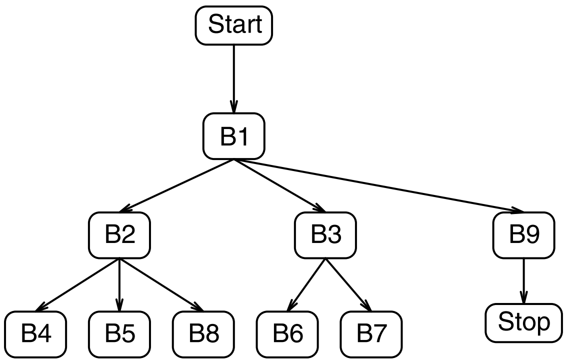


FIGURE 3.49 Example of value-numbering with SSA forms contd.



Reverse postorder on the SSA graph that is used with the dominator tree above:

Start,B1,B3,B7,B6,B2,B5,B4,B8,B9,Stop

FIGURE 3.50 Dominator tree and reverse postorder for Figure 3.48.

propagation becomes much faster on SSA graphs due to SSA edges and ϕ-functions. Further, it discovers at least the same set of constants as the algorithm on the flow graph. In the case of value numbering, the SSA version becomes not only faster and simpler but also global, due to single assignment property. The results available on other SSA-based algorithms are unfortunately not as striking in their demonstration of advantages over flow graph-based algorithms.

We do not make any recommendations to use any one of these forms exclusively in a compiler. Much more research on SSA-based optimization algorithms is essential before any such recommendations can be made. It is also necessary that more implementations of available SSA algorithms are carried out and comparisons are made with flow graph-based algorithms. Investigations on whether optimizations can be combined on the lines of [36], and on alternative SSA forms [42, 43] on which more optimization algorithms can operate, are required to be conducted. The issue of using dependence information in SSA forms also arises while tackling parallelization transformations. The array SSA form and its use in parallelization are discussed in [37]. Efficient storage allocation is an important question that needs to be addressed while dealing with code generation. It is not known whether SSA forms can be beneficially modified to include other data flow information such as liveness and aliasing. Efficient code generation techniques such as tree pattern matching cannot be applied directly to SSA forms (with SSA edges) and it is necessary to carry out more research on the needed modifications to these techniques (or new techniques) to support SSA forms.

References

[1] M.P. Gerlekm, E. Stolz and M. Wolfe, Beyond induction variables: detecting and classifying sequences using a demand-driven SSA form, *ACM TOPLAS*, 17(1), 85–122, 1995.

[2] M.N. Wegman and F.K. Zadeck, Constant propagation with conditional branches, *ACM TOPLAS*, 13(2), 181–210, 1991.

[3] A.V. Aho, R. Sethi and J.D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA, 1986.

[4] A.V. Aho, J.E. Hopcroft and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[5] A.V. Aho, J.E. Hopcroft and J.D. Ullman, *Data Structures and Algorithms*, Addison-Wesley, Reading, MA, 1983.

[6] T.H. Cormen, C.E. Leiserson and R.L. Rivest, *Introduction to Algorithms*, MIT Press and McGraw-Hill, New York, 1990.

[7] S.S. Muchnick, *Advanced Compiler Design Implementation*, Morgan Kaufmann, San Francisco, 1997.

[8] S.S. Muchnick and N.D. Jones, *Program Flow Analysis: Theory and Applications*, Prentice Hall, New York, 1981.

[9] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman and F.K. Zadeck, Efficiently computing the static single assignment form and the control dependence graph, *ACM TOPLAS*, 13(4), 451–490, 1991.

[10] J. Knoop, O. Rüthing and B. Steffen, Optimal code motion: theory and practice, *ACM TOPLAS*, 16(4), 1117–1155, 1994.

[11] E. Morel and C. Renvoise, Global optimization by suppression of partial redundancies, *Commun. ACM*, 22(2), 96–103, 1979.

[12] V.K. Paleri, Y.N. Srikant and P. Shankar, A simple algorithm for partial redundancy elimination, *ACM SIGPLAN Notices*, 33(12), 35–43, 1998.

[13] R. Kennedy, S. Chan, S.-M. Liu, R. Lo, P. Tu and F. Chow, Partial redundancy elimination in SSA form, *ACM TOPLAS*, 21(3), 627–676, 1999.

[14] V.C. Sreedhar and G.R. Gao, A Linear Time Algorithm for Placing ϕ-Nodes, ACM Symposium on Principles of Programming Languages, 1995, pp. 62–73.

[15] P. Briggs, K.D. Cooper and L.T. Simpson, Value numbering, *Software Pract. Exp.*, 27(6), 701–724, 1997.

[16] B. Alpern, M.N. Wegman and F.K. Zadeck, Detecting Equality of Variables in Programs, ACM Symposium on Principles of Programming Languages, 1988, pp. 1–11.

[17] K.D. Cooper, L.T. Simpson and C.A. Vick, Operator strength reduction, Technical report CRPC-TR95635-S, Rice University, Houston, TX, 1995.

[18] R. Gupta, Optimizing array bound checks using flow analysis, *ACM Lett. Programming Languages and Syst.*, 2(1–4), 135–150, 1993.

[19] K.D. Cooper and L.T. Simpson, SCC-based value-numbering, Technical report CRPC-TR95636-S, Rice University, Houston, TX, 1995.

[20] C. Click and K.D. Cooper, Combining analyses, combining optimizations, *ACM TOPLAS*, 17(2), 181–196, 1995.

[21] J.M. Asuru, Optimization of array subscript range checks, *ACM Lett. Programming Languages Syst.*, 1(2), 109–118, 1992.

[22] P. Kolte and M. Wolfe, Elimination of redundant array subscript range checks, *ACM SIGPLAN Notices*, 30(6), 270–278, 1995.

[23] F. Chow, A Portable Machine Independent Optimizer-Design and Implementation, Ph.D. dissertation, Department of Electrical Engineering, Stanford University, Stanford, CA; and Technical Report 83–254, Computer Systems Laboratories, Stanford University, Stanford, CA, 1983.

[24] D.M. Dhamdhere, A fast algorithm for code movement optimization, *SIGPLAN Notices*, 23(10), 172–180, 1988.

[25] D.M. Dhamdhere, Practical adaptation of the global optimization algorithm of Morel and Renvoise, *ACM TOPLAS*, 13(2), 291–294, 1991.

[26] V.M. Dhaneshwar and D.M. Dhamdhere, Strength reduction of large expressions, *J. Programming Languages*, 3, 95–120, 1995.

[27] K.H. Dreshler and M.P. Stadel, A variation of Knoop, Rüthing, and Steffen's lazy code motion, *SIGPLAN Notices*, 28(5), 29–38, 1993.

[28] J. Knoop, O. Rüthing and B. Steffen, Lazy code motion, *SIGPLAN Notices*, 27(7), 224–234, 1992.

[29] V.K. Paleri, An Environment for Automatic Generation of Code Optimizers, Ph.D. thesis, Department of Computer Science and Automation, Indian Institute of Science, Bangalore, India, 1999.

[30] B.K. Rosen, M.N. Wegman and F.K. Zadeck, Global Value Numbers and Redundant Computations, *ACM Symposium on Principles of Programming Languages*, 1988, pp. 12–27.

[31] J. Cocke and J.T. Schwartz, Programming Languages and Their Compilers: Preliminary notes, Technical report, Courant Institute of Mathematical Sciences, New York University, New York, 1970.

[32] M.P. Gerlek, M. Wolfe and E. Stolz, A reference chain approach for live variables, Technical report, Department of Computer Science and Engineering, Oregon Graduate Institute of Science and Technology, Klamath Falls, OR, 1994.

[33] R. Kennedy, F. Chow, P. Dahl, S.-M. Liu, R. Lo and M. Streich, Strength Reduction via SSAPRE, International Conference on Compiler Construction, 1998.

[34] J. Knoop, O. Rüthing and B. Steffen, Lazy strength reduction, *J. Programming Languages*, 1, 71–91, 1993.

[35] D. Dhamdhere, A new algorithm for composite hoisting and strength reduction optimization, *Int. J. Comput. Math*, 27, 1–14, 1989.

[36] C. Click and K.D. Cooper, Combining analyses, combining optimizations, *ACM TOPLAS*, 17(2), 181–196, 1995.

[37] K. Knobe and V. Sarkar, Array SSA Form and Its Use in Parallelization, ACM Symposium on Principles of Programming Languages, 1998, pp. 107–120.

[38] J.-D. Choi, R. Cytron and J. Ferrante, Automatic Construction of Sparse Data Flow Evaluation Graphs, ACM Symposium on Principles of Programming Languages, 1991, pp. 55–66.

[39] T. Lenganeur and R.E. Tarjan, A fast algorithm for finding dominators in a flow graph, *ACM TOPLAS*, 1(1), 121–141, 1979.

[40] D. Harel, A Linear Time Algorithm for Finding Dominators in Flow Graphs and Related Problems, ACM Symposium on Theory of Computing, 1985, pp. 185–194.

[41] A.W. Appel, *Modern Compiler Implementation in Java*, Cambridge University Press, London, 1998.

[42] P. Havlak, Construction of thinned gated single-assignment form, *Languages and Compilers for Parallel Computing, 6th International Workshop, Lecture Notes in Computer Sciences,* Vol. 768, Springer-Verlag, New York, 1994, pp. 477–499.

[43] D. Wiese, R.F. Crew, M. Ernst and B. Steensgaard, Value Dependence Graphs: Representation without Taxation, ACM Symposium on Principles of Programming Languages, 1994, pp. 197–310.

[44] R. Morgan, *Building an Optimizing Compiler*, Digital Press, Bethlehem, PA, 1998.

4

Profile-Guided Compiler Optimizations

Rajiv Gupta
University of Arizona

Eduard Mehofer
University of Vienna

Youtao Zhang
University of Arizona

4.1 Introduction

Over the past several decades numerous compile-time optimizations have been developed to speed up the execution of programs. Application of a typical optimization can be viewed as consisting of two primary tasks: uncovering optimization opportunities through static analysis of the program, and transforming the program to exploit the uncovered opportunities. Most of the early work on classical code optimizations is based on a very simple performance model. Optimizations are defined such that their application is always considered to have a positive impact on performance. Therefore, the focus of the research has been on developing aggressive analysis techniques for uncovering opportunities for optimization in greater numbers and designing powerful program transformations to exploit most if not all the uncovered opportunities.

Although the simplicity of the above approach is attractive, in recent years it has been recognized that the preceding approach for optimizing programs has serious drawbacks. On one hand, this approach fails to exploit many opportunities for optimization; on the other hand, it may apply optimizations that may have an adverse impact on performance. The first drawback can be remedied by employing a combination of static analysis and execution profiles to detect optimization opportunities. The second drawback can be addressed by using a sophisticated performance model during the application of optimizations. In particular, a cost-benefit analysis of a program transformation can be carried out before actually applying the transformation. Given adequate profile data, the following broad categories of optimization opportunities can now be exploited:

- *Some optimization opportunities are uncovered using static analysis whose exploitation involves performance trade-offs.* In some situations the analysis phase of an optimization algorithm may be successful in uncovering an optimization opportunity; however, the

0-8493-1240-7/03/$0.00+$1.50

transformation phase may require cost-benefit analysis to determine whether the optimization opportunity should be exploited. For example, the transformation may represent a trade-off between improved performance in one part of the program and degraded performance in another part of the program.

- *Other optimization opportunities cannot be uncovered by static analysis, but are frequently observed to exist during program execution.* Program optimizations can be designed to exploit characteristics of values, representing data or addresses, encountered by various instructions during program execution. Although static analysis can be used to relate values of variables, it is not amenable to identifying specific values involved. For example, by examining the statement $i \leftarrow i + 1$, we can statically determine that the statement increments the value of i by 1. However, in general, it is not likely that we can determine the exact value of i because the value of i may depend on a number of runtime factors such as the program input and the number of times the statement is executed.

To illustrate the preceding scenarios, consider the example shown in Figure 4.1. Let us consider the first scenario. The original code contains an optimization opportunity. If both conditionals evaluate to true, the expression $x \times y$ is computed twice. Moreover, we can make this determination by statically analyzing the program. To optimize the program we may wish to transform the code as shown in Figure 4.1(b). This transformation removes the redundant evaluation of $x \times y$ for true evaluations of the conditionals. However, if we consider the execution corresponding to false evaluations of the conditionals, we find that an evaluation of $x \times y$ is introduced in the transformed code where none was present prior to transformation. Therefore, we can conclude that the transformation is useful only if line 4 is executed less frequently than line 7. A simple cost-benefit analysis based on expected execution frequencies of various statements in the program can be used to decide whether the transformation should be applied.

Next let us consider an instance of the second scenario. Let us assume that static analysis cannot identify any specific values associated with variable y during program execution. However, by profiling the execution of the program we may determine that very frequently the value of y at line 4 is 1. Therefore, by using profiling we have identified an optimization opportunity that was not detected using static analysis, namely, that the multiply operation at line 4 can be frequently optimized away. We can transform the program as shown in Figure 4.1(c). The multiply operation at line 4 is executed conditionally in the transformed code. Because extra instructions are required to check whether the value of y is 1 and accordingly update t, it is important that we ensure the overall benefit of eliminating the multiply operation is greater than the overall cost of executing the extra instructions. A simple cost-benefit analysis based on the expected frequencies with which variable

```
1.  if () then
2.      z = x × y
3.  else
4.  ...
5.  endif
6.  if () then
7.      w = x × y
8.  else
9.      ...
10. endif
```

(a) Original Code.

```
1.  if () then
2.      t = z = x × y
3.  else
4.      t = x × y
5.  endif
6.  if () then
7.      w = t
8.  else
9.      ...
10. endif
```

(b) Rebundancy Removal.

```
1.  if () then
2.      t = z = x × y
3.  else
4.      t = (y = 1)?x : x × y
5.  endif
6.  if () then
7.      w = t
8.  else
9.      ...
10. endif
```

(c) Strength Reduction.

FIGURE 4.1 Examples of profile-guided transformations.

y has the value 1 or some other value at line 4 can be used to determine whether the transformation should be applied.

Although the example in Figure 4.1 illustrated that execution profiles can be used to both detect optimization opportunities and identify beneficial transformations in context of classical optimizations, similar situations also arise in the context of memory optimizations discussed later in this chapter.

Another reason for increased relevance of profile-guided optimizations is that many opportunities for optimizations that could not be exploited on processors of the past can now be exploited due to the advanced hardware features present in modern processors. Support for safe speculative execution and predicated execution can be helpful in carrying out profile-guided optimizations. For example, if the multiple operation in Figure 4.1 is replaced by a divide operation, then the placement of x/y at line 4 can result in a divide by 0 exception that may not have occurred during the original program execution. Therefore, the redundancy removal transformation is rendered illegal. However, the IA-64 processor provides hardware support for suppressing spurious exceptions and therefore allows the optimization of divide operation to be carried out [18]. Support for predicated execution, also provided by IA-64, can enable efficient implementation of the strength reduction transformation shown in the preceding example because it enables conditional execution of the multiply operation without introducing additional branch instructions.

From the previous discussion it is clear that the cost-benefit analysis should be an integral part of an optimization algorithm. Moreover, cost-benefit analysis for a given program execution can only be carried out if we are provided with estimates of execution frequencies of various runtime events that are both relevant to the program performance and are impacted by the program transformation. Although in the preceding example both the cost and the benefit was measured in terms of number of instructions (or cycles), this may not always be the case. In some situations although the benefit may be measured in terms of anticipated reduction in the number of instructions, the cost may be in measured in terms of code growth resulting from the transformation. This is because many of the optimizations essentially perform code specialization that results in code growth.

The profile-guided compilation process is summarized in Figure 4.2. Before profile-guided optimizations can be carried out, an instrumented version of the program must be run on one or more representative inputs to collect profile data. These profile data are then used by the optimizing compiler to recompile the program generating optimized code. The selection of representative inputs is important because the optimizations are expected to be beneficial only if the profile data that is collected is relatively stable across a wide range of inputs on which the program is expected to be executed. Although this chapter is devoted to static profile-guided optimization of programs, the concept of profile-guided optimization is also used by dynamic optimizers. However, in contrast to the techniques described in this chapter, the profiling and optimization techniques employed during dynamic optimization must be extremely lightweight.

The remainder of this chapter is organized as follows. In Section 4.2 we provide a brief overview of types of profile data that are used to guide a wide range of optimizations. In Section 4.3 we discuss the role of profiles in carrying out classical optimizations. We illustrate the principles of profile-guided optimization through an in-depth look at a partial redundancy elimination algorithm.

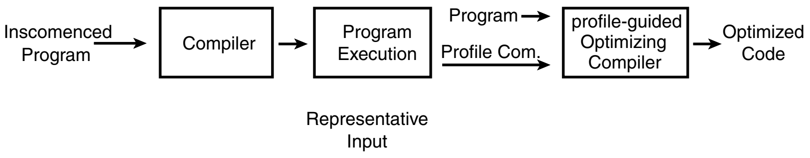


FIGURE 4.2 Profile-guided optimizing compiler.

In Section 4.4 we discuss the role of profiles in carrying out optimizations that improve the memory performance of a program. Concluding remarks are given in Section 4.5.

4.2 Types of Profile Information

Profiles provide summary information on past program executions that are used to guide program optimization. However, different types of optimizations require different types of profiles. In particular, three types of profile information are used in practice: control flow profiles, value profiles and address profiles. In this section we briefly discuss each of the profile types including the types of optimizations where they are used.

4.2.1 Control Flow Profiles

One form of profile captures a trace of the execution path taken by the program. This trace represents the order in which the nodes corresponding to basic blocks in the program control flow graph (CFG) are visited. We refer to such a profile as the program control flow trace (CFT). By examining a CFT we can compute the execution frequency of any given program subpath. As expected, CFTs can be extremely large in size and therefore representations that maintain CFTs in compressed form have been considered. Such compressed forms are referred to as *whole program paths* (WPPs). In [27] Larus used the Sequitur [31] algorithm for compression whereas in [43] Zhang and Gupta proposed alternative redundancy removal techniques for compression. However, even after compression, WPPs can be extremely large.

In practice, a number of approximations of CFT that directly measure the execution frequencies of selected program subpaths are used. These profiles differ in the degree of approximation involved and the cost for collecting them. The proposed approximations of control flow profiles include the following:

- *Node profiles* provide the execution frequencies of the basic blocks in the CFG. For some optimizations such profiles are adequate. For example, in making code placement decisions, as illustrated by the redundancy removal transformation of Figure 4.1(b), node profiles are sufficient.
- *Edge profiles* provide the execution frequencies of each edge in the CFG. The overhead of collecting edge profiles is comparable to the overhead of collecting node profiles. However, edge profiles are superior to node profiles because edge profiles cannot always be computed from node profiles whereas node profiles can always be computed from edge profiles. Edge profiles are widely used.
- *Two-edge profiles* [30] provide the execution frequencies of each pair of consecutive edges in the CFG. They are clearly superior to edge profiles. Edge profiles can always be computed from two-edge profiles. However, the reverse is not true. Two-edge profiles derive their increased power from their ability to capture the correlation between the executions of consecutive conditional branches.
- *Path profiles* [2] provide the execution frequencies of acyclic subpaths in the CFG such that they are acyclic and intraprocedural. Because a path is acyclic, it does not ever include a loop back edge and because it is intraprocedural, it terminates if an exit node of a procedure is reached. Path profiles are more precise than two-edge profiles for acyclic components of a CFG because they capture correlation across multiple conditional branches within an acyclic graph. However, two-edge profiles can capture correlation among a pair of conditional branches along a cyclic path whereas path profiles cannot do so.

	Node	Edge	Two-edge	Path
Length (nodes)	1	2	3	>1
Overlap (nodes)	0	0	2	0

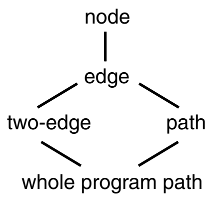


FIGURE 4.3 Relative precision of control flow profiles.

4.2.1.1 Relative Precision of Control Flow Profiles

Each of the preceding profiles can be used to derive an estimate of the execution frequency of an arbitrary subpath in the program. However, the precision of the estimate can vary from one type of profile data to another. If we examine the subpaths with execution frequencies that are directly measured by each of the preceding profiling algorithms, they can be distinguished based on two important characteristics. First, the lengths of the subpaths whose execution frequencies are collected vary. Second, the degree of overlap between two subpaths that may be executed one after another also varies. The table in Figure 4.3 gives the length and overlap, in terms of number of nodes, for each of the profile types. From this table we can conclude the relationships of the precisions for any pair of profile types. For example, the two-edge and path profiles are not comparable because the former has greater degree of overlap whereas the latter allows for longer paths. The precision relationships are summarized by the hierarchy shown in Figure 4.3.

Let us illustrate the estimation of the execution frequency of a given program path from various types of profiles. When such an estimate is not exact, we can obtain lower and upper bounds on the execution frequency. If the lower and upper bounds for a path are found to be l and h, then it means that the path was definitely executed l times and potentially executed as many as h times. Figure 4.4 shows a sample CFG and the complete CFT for a program execution. The node, edge, two-edge and path profiles corresponding to this program execution are also given. Let us estimate the frequency of path `abefgk` from various profiles.

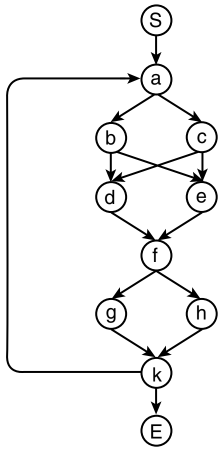
- *Estimate based on node profiles.* If we examine the execution frequencies of the nodes on path `abefgk`, the minimum frequency encountered is 70 for node `g`. Therefore the upper bound on the execution frequency of the path is 70. To find the lower bound on the execution frequency of path `abefgk`, first consider the lower bound on the execution frequency of subpath `efgk`. Because node `d` is executed 20 times, the lower bound on the execution frequency of `efgk` is 50. By further considering that the execution frequency of node `c` is 20, the lower bound on the execution frequency of path `abefgk` is 30. Therefore, we conclude that path `abefgk` is executed at least 30 times and potentially as many as 70 times.
- *Estimate based on edge profiles.* By examining the edge profiles, we can conclude that the subpaths `abef`, `fgk` and `fhk` are executed exactly 60, 70 and 30 times, respectively. Therefore, we can conclude that the path `abefgk` is executed at least 30 times and potentially 60 times. An algorithm for deriving these estimates can be found in [3].
- *Estimate based on two-edge profiles.* As in the case of edge profiles, we can deduce that the subpath `abef` has a frequency of 60. The frequency of subpaths `efg` and `efh` are measured directly to be 60 and 20, respectively. Therefore, we can conclude that the path `abefgk` is executed at least 40 times and possibly as many as 60 times.
- *Estimate based on path profiles.* During path profiling the execution frequency of path `abefgk` is directly measured because it is an acyclic path. Therefore, we know that its execution frequency is exactly 50.

4.2.1.2 Cost of Collecting Profiles

To collect the profiles we must execute instrumented versions of the program. The instrumentation code that is introduced depends on the type of profiles being collected. Although in general the overhead of instrumented code is linear in the length of the program execution, techniques can be employed to reduce the overhead.

During the collection of node profiles, instead of instrumenting each basic block to collect its execution frequency, we can introduce a counter for each control dependence region in the program. The execution frequency of a basic block is equal to the sum of the execution frequencies of the control dependence regions to which it belongs [32]. Because there are far fewer control dependence regions than there are basic blocks, this approach reduces the overhead of profiling. In Figure 4.4 the basic blocks `a`, `f` and `k` belong to the same control dependence region and they all have identical execution frequencies. Therefore, a single shared counter can be used to measure their execution frequencies.

Similarly, each edge in the CFG does not need to be instrumented because execution frequencies of some edges can be computed from execution frequencies of other edges. In Figure 4.4 the edges `fg` and `gk` always have the same frequency, so that at most one of them should be instrumented. Moreover, if the frequencies of edges `df`, `ef` and `fh` are known, in absence of exceptions, we can deduce the frequency of edge `fg`.



Control Flow Trace:
S(abdfgk)^{10}(acefhk)^{10}
(abdfhk)^{10}(acefgk)^{10}
(abefgk)^{10}(abefhk)^{10}E

Node	Frequency
S	1
a	100
b	80
c	20
d	20
e	80
f	100
g	70
h	30
k	100
E	1

Edge	Frequency
Sa	1
ab	80
ac,ce	20
bd,df	20
be	60
ef	80
fg,gk	70
fh,hk	30
ka	99
kE	1

Two-Edge	Frequency
Sab	1
abd,bdf	20
abe,bef	60
ace,cef	20
dfg	10
dfh	10
efg	60
efh	20
fgk	70
fhk	30
gka	80
hka	20
kac	20
kab	79
hkE	1

Path	Frequency
Sabdfgk	1
abdfgk	9
abdfhk	10
abefgk	50
acefhk	10
acefgk	10
abefhk	9
abefhkE	1

Profile →	Node	Edge	Two-edge	Path
Freq. Estimate for Path abcfgk	30-70	30-60	40-60	50

FIGURE 4.4 An example of control flow profiles.

The collection of two-edge and path profiles is more expensive. However, their computation can also be optimized by reducing instrumentation points. In [2] an algorithm is presented to reduce the overhead of instrumentation code during collection of path profiles. Each path in the acyclic graph is assigned a unique number. The form of the instrumentation is such that it computes the path number as the path is traversed. On reaching the end of the path, this path number is available and used to update the frequency counter associated with the path. By carefully placing the instrumentation code, the number of instructions needed to compute the path number can be minimized.

4.2.2 Value Profiles

Value profiles identify the specific values encountered as an operand of an instruction as well as the frequencies with which the values are encountered. The example in Figure 4.5 illustrates the form of these profiles. With this information the compiler can recognize operands that are almost always constant and utilize this information to carry out value specialization optimizations such as constant folding, strength reduction and motion of nearly invariant code out of loops.

Because the number of instructions in a program is large, and each operand of an instruction may potentially hold a very large number of values, collection of complete value profiles is not practical. Therefore, to reduce the size of the profile data and the execution time overhead of profiling the following two steps are taken.

First, only the most frequently appearing `N` values are collected for a given operand. Calder et al. [9] have proposed maintaining a top-n-value table (TNV) for a register written by an instruction. Each TNV table entry contains a pair of values: the value and the frequency with which that value is encountered. Least frequently used (LFU) replacement policy is used to choose an entry for replacement when the table is full. If we exclusively use the LFU policy for updating the TNV, the values that are encountered later in the program may not be able to reside in the table even if they are encountered frequently. This is because they may be repeatedly replaced. To avoid this situation, at regular intervals the bottom half of the table is cleared. By clearing part of the table, free entries are created that can be used by values encountered later in the program. Both the number of entries in the table and clearing interval are carefully tuned to get good results. Collecting only top `N` values not only reduces the profiling overhead but also makes the convergence to a steady state easier and faster to reach.

Second, the complimentary approach to reducing profiling overhead is to collect value profiles for only interesting instructions. Watterson and Debray [35] use a cost-benefit model where the cost is the testing cost of whether a register has a special value, and the benefit is the direct and indirect instruction savings that can be achieved by optimizing the program with this information. Control flow profiles are collected first to carry out cost-benefit analysis and identify candidates for value profiling.

Code:

```
I1: load R3, 0(R4)
I2: R2←R3 & 0xff
```

Value profile:

(instruction, register)	profiles (value,freq)
(I1,R3)	(0xb8d003400,10)...
...	...
(I1,R2)	(0,1000)
(I2,R3)	(0,100),(0x8900,200) ...,(0x2900,100)

FIGURE 4.5 Sample value profiles.

4.2.3 Address Profiles

Address profiles can be collected in form of a stream of memory addresses that are referenced by a program. These profiles are usually used to apply data layout and placement transformations for improving the performance of memory hierarchy. Depending on the optimization, the address traces can be collected at different levels of granularity. At the finest level of granularity, each memory address can be traced. Coarser level traces record references to individual objects instead of individual addresses.

A program address space can be divided into three parts: stack, heap and globals. Stack data typically exhibit good cache locality. Therefore, most of the research is focused on improving the data access behavior of globals and heap data. Programmers usually organize their data structures logically. They tend to put logically related data objects or data fields together. However, the logical relationships may be different from the order in which the data structures are actually accessed at runtime. By using the information provided by address profiles, compilers can reorganize the placement of data objects with respect to each other or the placement of fields within a data object such that closely accessed items are placed next to each other. In this way the cache behavior is improved.

A complete address trace of a program run can be extremely large. To compress the size of the address trace, Chilimbi [13] has proposed using the Sequitur algorithm to generate a compressed whole program stream (WPS) representation of the address trace in much the same way as Sequitur is used to compress a program control flow trace. To guide the application of data layout and placement transformations, the WPS representation is analyzed to identify hot address streams. These streams represent subsequences of addresses that are encountered very frequently during the program run.

Although the preceding approach first collects complete address profiles and then processes them to identify information useful in guiding data layout and placement transformations, another approach is to directly identify the useful information. Calder et al. [10] have proposed an algorithm based on such an approach. The information that they collect is represented by a graph named the *temporal relationship graph* (TRG). The nodes in this graph are data items of interest. Weighted links are established between pairs of nodes.

If references to a pair of data items are separated by fewer than a threshold number (say N) of other data references, then the weight associated with the link between the two items is incremented. To maintain the weights of all the links, an N-entry queue is maintained that records the latest N data items that are referenced by the program. The weights on the links at the end of the program run can be used by the compiler to identify data items that should be placed close to each other for achieving good cache behavior. Figure 4.6 shows an example of the information collected using this approach.

4.3 Profile-Guided Classical Optimizations

Simple optimization algorithms typically optimize statements that are determined to be optimizable under all conditions through static analysis of the program. On the other hand, more aggressive algorithms also optimize statements that are conditionally optimizable where the optimization opportunities are discovered either through static analysis or through profiling. As a consequence, often such algorithms involve replicating statements and creating unoptimized and optimized copies of them. Depending on the conditions that hold, appropriate copy of the statement is executed. The preceding process is also commonly referred to as *code specialization*. In some optimizations specialization leads to elimination of a copy (e.g., redundancy and dead code elimination) whereas during other optimizations specialization leads to simpler and more efficient code (e.g., strength reduction and constant folding).

Sample code:

```
for (1=0;i<2000;i++){
    switch (tag) {
    case 1:
        xa = *pa; ...; break;
    case 2:
        xb = *pb; ...; break;
    case 3:
        xc = *pc; ...; break;
    case 4:
        xd = *pd; ...; break;
    }
    ....
    pa = buf[i]
    ....
}
```

Declarations:

```
int tag:
int*pa,*pb,*pc,*pd;
int bu[2000]:
...
int xa,xb,xc,xd;
```

Address profile:

Link	Weight
(A(xa)A(pa))	500
(A(xb)A(pb))	20
(A(xc)A(pc))	2
(A(xd)A(pd))	10
....	....
(A(pa)A(bu1))	2000

FIGURE 4.6 Address profiles.

In this section we begin by providing a brief overview of various code specialization transformations followed by identification of specific optimizations that rely on them. We describe the critical role that profiling plays in carrying out these optimizations. Later we present a profile-guided partial redundancy algorithm to illustrate how profile data can be exploited in a systematic way in developing a profile-guided optimization algorithm.

4.3.1 Transformations

Different code specialization algorithms carry out code replication at different levels of granularity. For example, function inlining replicates entire functions, partial inlining replicates code along selected paths through a function and code motion based algorithms replicate individual statements. Primarily, the two classes of transformations that are used to carry out code replication and enable specialization of conditionally optimizable code are code motion of different types and control flow restructuring with varying scope.

4.3.1.1 Code Motion of Statements

The basic form of code motion, namely, safe code motion, in addition to honoring program data dependences, guarantees that for every execution of a statement during the execution of optimized code, a corresponding execution of the statement exists during the execution of unoptimized code. As a consequence, it must be the case that if an exception occurs during the execution of optimized code, it would have also occurred during the execution of unoptimized code. Hardware support present in modern processors such as IA-64 allows relaxation of the preceding constraint and yet preserves the program semantics [18]. In particular, speculative code motion allows the compiler to introduce executions of a statement in the optimized code that are not present in the unoptimized code. To be beneficial, the added cost of these extra executions must be offset by savings resulting from speculative code motion. Therefore, profile-based cost-benefit analysis is typically required to take advantage of speculative code motion. Predicated code motion further creates opportunities for performing code motion more freely [18]. Statements can be moved out of control structures and executed at a different program point under the original conditions by predicating its execution with an appropriately constructed predicate expression. Profiling can assist in estimating whether the benefit achieved by code motion outweighs the cost of evaluating the predicate expression.

4.3.1.2 Control Flow Restructuring

If the conditions under which a statement can be optimized hold along some execution paths but not others, then control flow restructuring can be employed to enable the optimization of the statement as follows. By using restructuring we can create a program such that when we reach the statement, based on the incoming edge along which we arrive at the statement, we can determine whether the statement can be optimized. We can create unoptimized and optimized copies of the statement and place them along the incoming edges. The scope of control flow restructuring determines the degree to which the code can be optimized. For example, the changes due to intraprocedural control flow restructuring are localized to within a procedure body whereas interprocedural restructuring changes the flow of control across procedure boundaries. The primary cost of using this transformation is the resulting growth in code size. Increasing the scope of restructuring also increases the amount of resulting growth in code size. Function inlining is one way to achieve interprocedural control flow restructuring. However, it is also accompanied with significant code growth. To limit code growth while performing interprocedural optimizations, a couple of alternative techniques have been proposed: partial inlining of frequently executed paths through a procedure [23] and creating procedures with multiple entries and multiple exits [4].

4.3.2 Optimizations

4.3.2.1 Partial Redundancy Elimination

The PRE of expressions is traditionally performed using safe code motion [15, 25, 36]. However, many opportunities for redundancy removal cannot be exploited using this restricted form of code motion. For example, in Figure 4.7(a) the evaluation of expression `x + y` in node 7 is partially redundant because if node 2 is visited prior to reaching node 7, then the expression has already been evaluated at node 2 and does not need to be evaluated again. However, the partial redundancy of `x + y` cannot be either reduced or eliminated using safe code motion. If speculative code motion is used to hoist `x + y` above node 6, as shown in Figure 4.7(b), a net reduction in partial redundancy of `x + y` results if the frequency of node 3 is less than the frequency of node 7. If control flow restructuring is employed in the manner shown in Figure 4.7(c), then the redundancy is entirely eliminated. However, some code growth also results. Cost-benefit functions based on profile data can be designed to selectively carry out the preceding transformations.

The use of speculation was first proposed in [20, 21] and that of control flow restructuring was first proposed in [40]. The use of the combination of all the preceding transformations to achieve greater benefits at lower costs is discussed in [6]. Although we have only considered redundancy elimination in the context of arithmetic expressions, partial redundancy elimination can also be applied in other contexts such as load removal [7] and array bound check elimination [8, 19].

4.3.2.2 Partial Dead Code Elimination

The PDE of an assignment is achieved by delaying the execution of the statement to a point where its execution is definitely required [26]. Using conservative code motion all opportunities for PDE cannot be exploited. For example, the assignment in node 2 of Figure 4.7(d) is partially dead because if control reaches nodes 6 or 7, the value computed by the assignment is never used. However, we cannot simply delay the execution of the assignment to node 8 because in some cases placement of the assignment at node 8 can block the correct value of `x` from reaching its use in node 10. Straightforward code motion can block the sinking of the assignment to node 4, and therefore any farther, thus preventing any optimization to occur. By predicating the statement we can enable its sinking past node 4 and down to 8 and thus achieve PDE — (see Figure 4.7(e)). This approach is clearly beneficial if the frequency of node 8 is less than that of node 2. Control flow restructuring can also be applied to carry out PDE as shown in Figure 4.7(f). As in the case of PRE, profile-based cost-benefit analysis is required to selectively apply the preceding forms of PDE. Predication-based

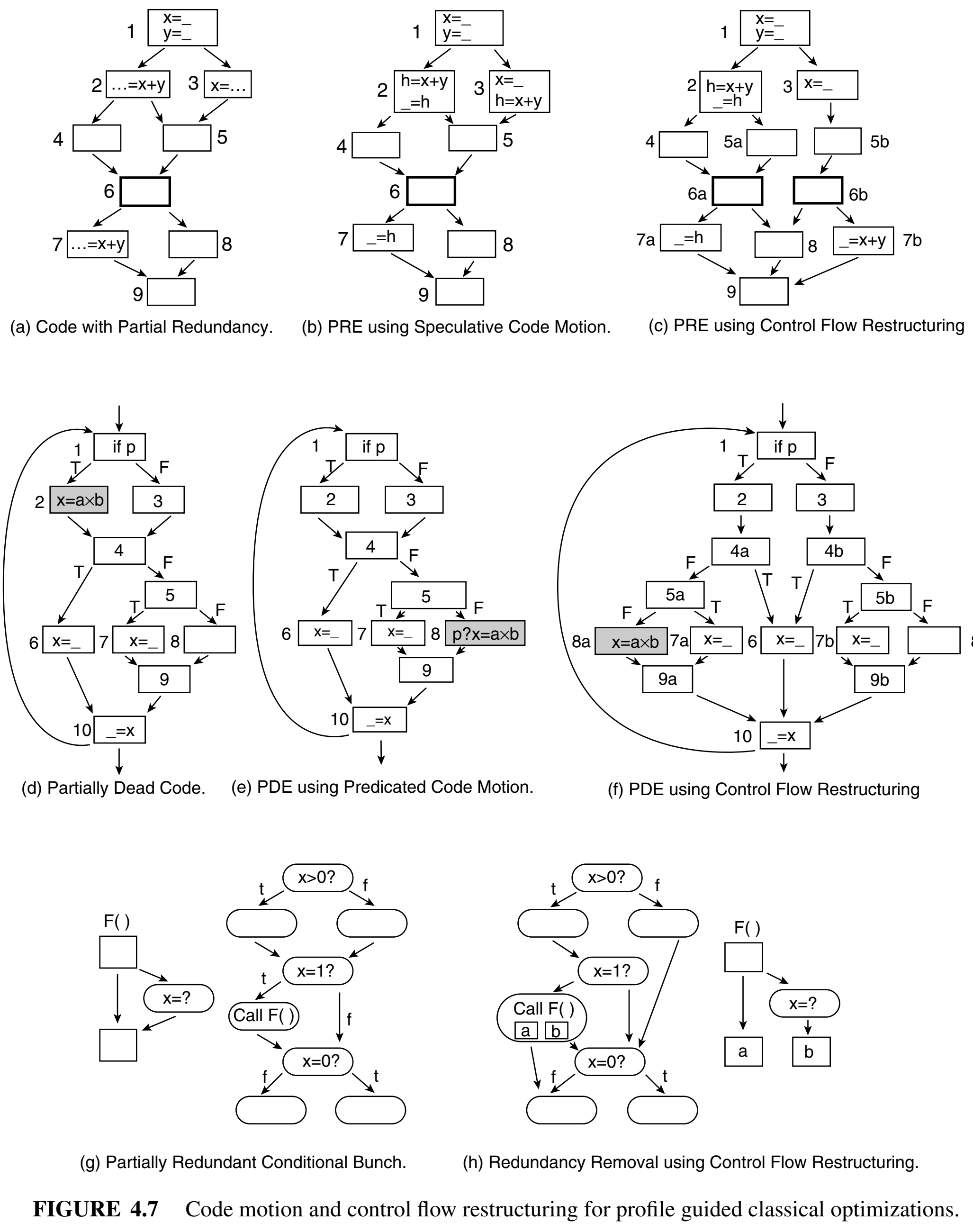

(a) Code with Partial Redundancy.
(b) PRE using Speculative Code Motion.
(c) PRE using Control Flow Restructuring
(d) Partially Dead Code.
(e) PDE using Predicated Code Motion.
(f) PDE using Control Flow Restructuring
(g) Partially Redundant Conditional Bunch.
(h) Redundancy Removal using Control Flow Restructuring.

FIGURE 4.7 Code motion and control flow restructuring for profile guided classical optimizations.

PDE was first proposed in [22]. A control flow restructuring algorithm that minimizes code growth associated with PDE can be found in [5]. Just as partially dead assignments can be eliminated using the motion and restructuring transformations, partially dead stores can also be eliminated.

4.3.2.3 Conditional Branch Elimination

A conditional branch is considered to be partially redundant if along some paths its outcome can be determined at compile time. Elimination of a partially redundant conditional branch requires flow graph restructuring to be performed. During flow graph restructuring, the paths along which the outcome of a conditional branch is known at compile time are separated from the paths along which its outcome is unknown through code duplication. The basic idea of intraprocedural restructuring was first proposed in [37] and then a more general algorithm based on interprocedural demand-driven analysis as well as profile-guided interprocedural control flow restructuring was given in [4].

The last example in Figure 4.7 illustrates this optimization. The flow graph in Figure 4.7(g) contains two paths, an intraprocedural one and an interprocedural one, along which conditional branch elimination is applicable. The first path is established when $x > 0$ is false and hence $x = 1$ is known to be false. The second path arises because if $x = 1$ is true, and value of x is preserved during the call to function $F()$, the condition $x = 0$ evaluates to false. The first opportunity is exploited through intraprocedural control flow restructuring and the second requires interprocedural restructuring that is achieved by splitting the exit of function $F()$ as shown in Figure 4.7(h).

4.3.3 Partial Redundancy Elimination via Speculative Code Motion

A computation is called partially redundant if at least one path exists on which it is computed twice. Such redundancies can be eliminated by executing the computation once at an appropriate program point, assigning the value to an auxiliary variable and replacing the original computations by the auxiliary variable. The problem of finding an appropriate program point is guided by the following transformational idea. Unnecessary recomputations can be avoided by moving computations in the opposite direction of the control flow and placing them in a more general context maximizing in this way the potential of redundant code that can be eliminated thereafter.

Under the restriction that no new computations are inserted along any path, computationally optimal results can be obtained statically [25]. However, it is assumed that all paths are equally important, which is not true in practice. In this section we present a PRE algorithm that takes profile information into account to reduce the number of recomputations further. By executing expressions speculatively, it enables the removal of redundancies along more frequently executed paths at the expense of introducing additional expression evaluations along less frequently executed paths. More specifically, speculation is the process of hoisting expressions such that an expression whose execution is controlled by a conditional prior to speculation is executed irrespective of the outcome of that conditional after speculation. In this way new computations may be introduced on some paths that can alter the semantics of a program in case these computations can cause exceptions. However, modern architectures like Intel IA-64 support speculative execution of instructions by suppressing exceptions making this kind of transformation safe.

The remainder of this section is organized as follows. We first present a cost model that can be used to identify profitable opportunities for speculation. Next we present a practical algorithm for carrying out cost-benefit analysis based on probabilistic data flow analysis. We show how under this analysis the application of the cost model is approximated. Finally, we present the details of the code motion framework used to carry out PRE using a combination of safe code motion and selectively enabled speculative code motion based on cost-benefit analysis.

4.3.3.1 Cost Model

For a given expression *exp* and a conditional node n we formulate a condition under which it is potentially profitable to enable speculative hoisting of *exp* above n. Essentially we identify the paths through n that can benefit from speculation of *exp* and paths that incur an additional execution time cost due to speculation of *exp*. If the probability of following paths that benefit is greater than the paths that incur a cost, then speculation of *exp* is enabled at n. The computation of these probabilities is based on an execution profile.

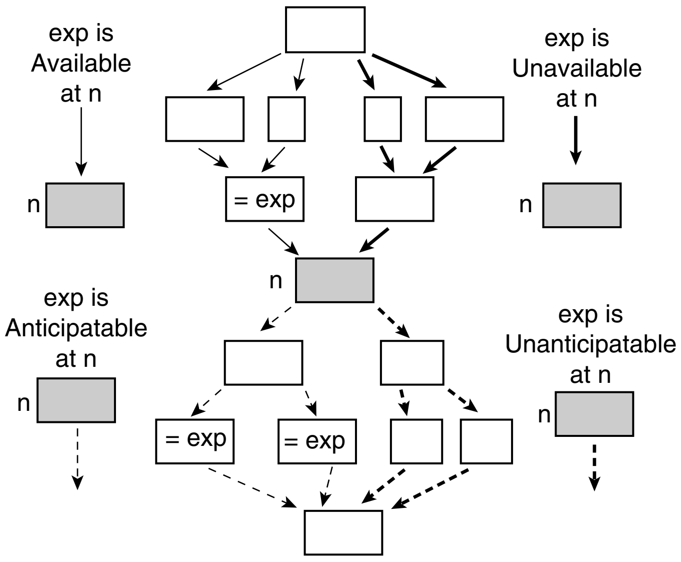


FIGURE 4.8 Path classification.

To develop the preceding cost model for hoisting an expression *exp* above a conditional node n, it is useful to categorize the program subpaths that either originate or terminate at n as follows (Figure 4.8):

1. *Available subpaths* are subpaths from the *start* node to n along which an evaluation of *exp* is encountered and is not killed by a redefinition of any of its operands prior to reaching n.
2. *Unavailable subpaths* are subpaths from the *start* node to n along which *exp* is not available at n. These subpaths include those along which an evaluation of *exp* is not encountered or if *exp* is encountered, it is killed by a redefinition of one of its operands prior to reaching n.
3. *Anticipatable subpaths* are subpaths from n to the *end* node along which *exp* is evaluated prior to any redefinition of the operands of *exp*.
4. *Unanticipatable subpaths* are subpaths from n to the *end* node along which *exp* is not anticipable at n. These subpaths include those along which *exp* is not evaluated or if *exp* is evaluated, it is done after a redefinition of one of its operands.

The paths that can potentially benefit from hoisting of *exp* above a conditional node n are the paths that are obtained by concatenating available subpaths with *anticipatable* subpaths at n. First this is because all along these paths there is redundancy. An evaluation of *exp* prior to reaching n causes an evaluation of *exp* performed after visiting n to become redundant. Second, the redundancy cannot be removed unless *exp* is hoisted above node n. Therefore, we can now state that, given a path p that passes through the conditional node n, the hoisting of an expression *exp* above n can *benefit* path p only if *exp* is available and anticipatable at n's entry along p. We denote the set of paths through n that benefit from hoisting of *exp* above n as $BenefitPaths_{exp}(n)$. The expected overall benefit of speculating *exp* can be computed by summing up the execution frequencies of paths in this set, shown as follows (Figure 4.9):

$$Benefit_{exp}(n) = \sum_{p \in BenefitPaths_{exp}(n)} Freq(p)$$

The paths that incur a cost due to hoisting of *exp* above a conditional node n are the paths that are obtained by concatenating unavailable subpaths with unanticipatable subpaths at n. This is because

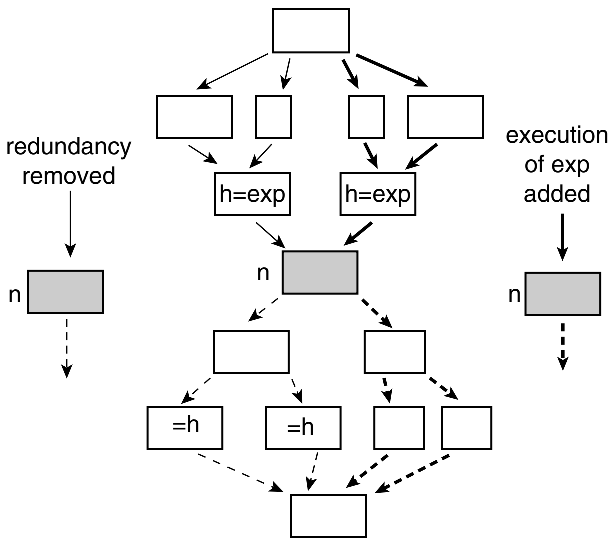


FIGURE 4.9 Paths that benefit vs. paths that incur a cost.

along these paths an additional evaluation of *exp* prior to reaching n is introduced. Therefore, given a path p that passes through a conditional node n, the hoisting of an expression *exp* above n *costs* path p only if *exp* is unavailable and unanticipatable at n's entry along p. We denote the set of paths through n that incur a cost due to hoisting of *exp* above n as $CostPaths_{exp}(n)$. The expected overall cost of speculating *exp* above n can be computed by summing up the execution frequencies of paths in this set shown as follows:

$$Cost_{exp}(n) = \sum_{p \in CostPaths_{exp}(n)} Freq(p)$$

Speculation should be enabled if based on the profile data we conclude that the expected benefit exceeds the expected cost. A Boolean variable $EnableSpec_{exp}(n)$ is associated with an expression and conditional node pair (exp, n), which is set to true if speculation for *exp* at n is enabled; otherwise it is set to false. For convenience we restate the speculation enabling condition in terms of probabilities as the detailed analysis algorithm that we present later is based on probabilistic data flow analysis. As shown later, speculation is enabled if the probability of following a path through n that benefits is greater than the probability of following a path that incurs an additional cost. $Freq(n)$ is the execution frequency of node n. Note that the sum of the following two probabilities does not need to be one because there may be other paths that are unaffected by speculation:

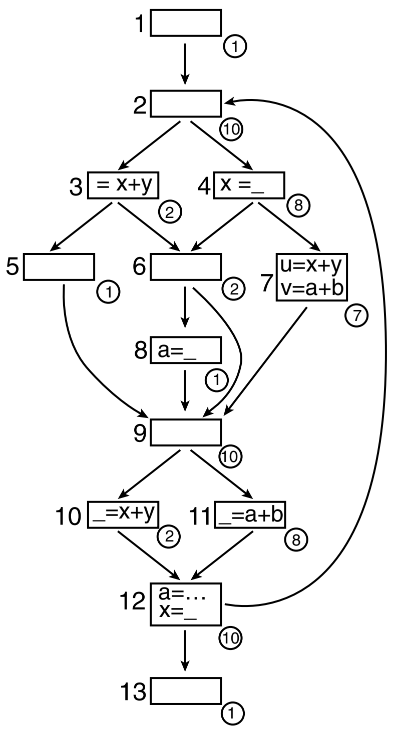
$$EnableSpec_{exp}(n) = ProbCost_{exp}(n) < ProbBenefit_{exp}(n)$$

where:

$$ProbCost_{exp}(n) = \frac{Cost_{exp}(n)}{Freq(n)}$$

and

$$ProbBenefit_{exp}(n) = \frac{Benefit_{exp}(n)}{Freq(n)}$$



<u>*Control Flow Trace :*</u>

1 – 2 – 3 – 5 – 9 – 10 – 12
–2 – 3 – 6 – 9 – 10 – 12
–(2– 4 – 7 – 9 – 11 – 12)^7
–2– 4 – 6 – 8 – 9 –11 –12 –13

Precise Probabilities	n			
	9	6	3	4
ProbCost$_{x+y}$(n)	$\frac{1}{10}$	$\frac{1}{2}$	$\frac{0}{2}$	$\frac{1}{8}$
ProbBenefit$_{x+y}$(n)	$\frac{2}{10}$	$\frac{1}{2}$	$\frac{0}{2}$	$\frac{0}{8}$
ProbCost$_{a+b}$(n)	$\frac{2}{10}$	$\frac{1}{2}$	$\frac{0}{2}$	$\frac{0}{8}$
ProbBenefit$_{a+b}$(n)	$\frac{7}{10}$	$\frac{0}{2}$	$\frac{0}{2}$	$\frac{0}{8}$

FIGURE 4.10 Enabling speculation using idealized cost model.

It should be noted that if an expression *exp* is never available or never anticipable at n, then $ProbBenefit_{exp}(n)$ is zero and therefore speculation is never enabled at n. On the other hand, if *exp* is always available or always anticipatable at n, then $ProbCost_{exp}(n)$ is zero and, if the $ProbBenefit_{exp}(n)$ is nonzero, speculation is enabled at n.

Let us apply the preceding cost model to the example in Figure 4.10. We assume that we are given the entire CFT shown in Figure 4.10 according to which the loop iterates 10 times. The encircled numbers at the right side of the nodes denote the number of times the node is visited during execution. Hence, $x + y$ is evaluated 11 times and $a + b$ is evaluated 15 times with 2 evaluations of $x + y$ and 7 evaluations of $a + b$ being redundant. Given the entire CFT, the cost model we have described is used to compute the precise probabilities given in the Figure 4.10. From these probabilities we conclude that we should enable speculative hoisting of both $x + y$ and $a + b$ above conditional node 9. However, the speculation for either of the expressions should not be enabled at conditional nodes 6, 3 and 4.

4.3.3.2 Cost-Benefit Analysis Based on Probabilistic Data Flow Analysis

The analysis we described in the preceding section is idealized in two respects. First, the availability and anticipability data flow information is required for each interesting path that passes through a conditional. Second, it is assumed that the entire CFT is available so that the execution frequencies of each of these interesting paths can be determined. Although it is possible to compute path-based data flow information and maintain complete control flow traces using the techniques described in [20] and [27, 43] respectively, for practical reasons it may be desirable to explore less expensive alternatives. In this section we present a detailed cost-benefit algorithm that uses probabilistic data flow analysis (PDFA) guided by two-edge profiles. Therefore this algorithm neither requires the entire control flow trace nor does is compute data flow information on a per path basis.

Whereas traditional data flow analysis calculates whether a data flow fact holds or does not hold at some program point, PDFAs calculate the probability with which a data flow fact can hold at some program point. Therefore, when applied to the data flow problems of availability and anticipability, PDFA provides us with the following probabilities: $AvailProb_{exp}(n)$ is the probability that expression *exp* is available at node *n* and $AntiProb_{exp}(n)$ is the probability that *exp* is anticipatable at node n. Therefore, we reformulate the speculation enabling condition for expression *exp* at a conditional node *n* in terms of these probabilities as follows:

$$EnableSpec_{exp}(n) = ProbCost_{exp}(n) < ProbBenefit_{exp}(n)$$

where

$$ProbBenefit_{exp}(n) \approx AvailProb_{exp}(n) \times AntiProb_{exp}(n)$$

and

$$ProbCost_{exp}(n) \approx (1 - AvailProb_{exp}(n)) \times (1 - AntiProb_{exp}(n))$$

It is important to note that the preceding formulation is an approximation of the prior cost model because it is based on the assumption that availability and anticipability at a conditional node are independent events. Therefore, we have computed the probability of following a path that benefits (incurs a cost) by taking the product of probabilities for availability (not availability) and anticipability (not anticipability). However, as we know, in practice the outcomes of conditionals in the program may be correlated. Therefore, the preceding model is approximate even though it uses precise probabilities for availability and anticipability.

Let us reconsider the example of Figure 4.10 in light of the preceding cost model. Before we can use this model, we should compute the availability and anticipability probabilities. We compute the precise probabilities using the CFT. As we can see from the results in Figure 4.11, even though the estimates for cost and benefit probabilities have changed, their relationship has not changed. Therefore, as before we again conclude that speculation should only be enabled at conditional node 9 for the two expressions.

Although in the preceding example we used an approximation of the cost model, we still used precise availability and anticipability probabilities derived from the CFT. Next we discuss how these probabilities can be approximated using PDFA based on two-edge profiles.

The first probabilistic data flow system that is based on edge probabilities was developed by Ramalingam [38]. A demand-driven frequency analyzer for which cost can be controlled by permitting a bounded degree of imprecision was proposed by Bodik et al. [6]. Because edge frequencies do not capture any branch correlation, the approximations introduced in computed probabilities can be considerable. Therefore, Mehofer and Scholz [30] proposed a data flow system based on two-edge probabilities. Instead of relating the data flow facts at a node only with the data flow facts at the immediate predecessor nodes in the flow graph using one-edge probabilities, the data flow facts at a node are related to two predecessor nodes using a flow graph with two-edge probabilities. In this way significantly better results can be achieved as presented in [30]. Of course, this approach can be made more precise by extending it beyond two edges; however, it can undermine our goal of computing the probabilities efficiently.

Availability and anticipability are calculated based on the traditional formulations of the data flow problems as shown in Figure 4.12(a). Availability analysis is guided by the local Boolean predicates *LocAvail* and *LocBlkAvail*. $LocAvail_{exp}(n)$ equals $true$, if an occurrence of *exp* in *n* is not blocked by a subsequent statement of *n*, that is, a statement that modifies variables used in *exp*. $LocBlkAvail_{exp}(n)$ equals $true$, if *exp* is blocked by some statement of *n*, that is, a statement modifying a variable used

Precise Probabilities	n			
	9	6	3	4
$AvailProb_{x+y}(n)$	$\frac{9}{10}$	$\frac{1}{2}$	$\frac{0}{2}$	$\frac{0}{8}$
$AntiProb_{x+y}(n)$	$\frac{2}{10}$	$\frac{1}{2}$	$\frac{2}{2}$	$\frac{7}{8}$
$AvailProb_{a+b}(n)$	$\frac{7}{10}$	$\frac{0}{2}$	$\frac{0}{2}$	$\frac{0}{8}$
$AntiProb_{a+b}(n)$	$\frac{8}{10}$	$\frac{0}{2}$	$\frac{0}{2}$	$\frac{7}{8}$

Approximate Probabilities	n			
	9	6	3	4
$ProbCost_{x+y}(n)$	0.08	0.25	0	0.125
$ProbBenefit_{x+y}(n)$	0.18	0.25	0	0
$ProbCost_{a+b}(n)$	0.06	1	1	0.125
$ProbBenefit_{a+b}(n)$	0.56	0	0	0

FIGURE 4.11 Enabling speculation using precise availability and anticipability probabilities.

in *exp*. Boolean conjunction is denoted by $\wedge$, Boolean disjunction is denoted by $\vee$ and boolean negation is denoted by a bar. Prefix *N* and *X* are used as abbreviations of entry and exit, respectively. The confluence operator $\bigwedge$ indicates that the availability problem is a forward all-path problem.

Similarly, anticipability analysis is guided by the local predicates *LocAnti* and *LocBlkAnti*. $LocAnti_{exp}(n)$ equals $true$, if an occurrence of *exp* in *n* is not blocked by a preceding statement of *n*, that is, a statement that modifies variables used in *exp*. $LocBlkAnti_{exp}(n)$ equals $true$, if *exp* is blocked by some statement of *n*, that is, a statement modifying a variable used in *exp*. The confluence operator $\wedge$ indicates that the anticipability problem is a backward all-path problem.

Because probabilistic data flow systems are based on any problems in which the meet operator is union, the analyses problems have to be transformed as described by Reps et al. [39]. If a "must-be-*X*" problem is an intersection problem, then the "may-not-be-*X*" problem is a union problem. The solution of the "must-be-*X*" problem is the complement of the solution of the "may-not-be-*X*" problem. Figure 4.12(b) presents the data flow equations after complementing the original ones. Note that the initializations of the start and end node have to be changed appropriately as well. Because now the confluence operator for both problems is union, the PDFA framework can be applied in a straightforward way. The solutions $N\text{-}NoAvail_*$, $X\text{-}NoAvail_*$, $N\text{-}NoAnti_*$ and $X\text{-}NoAnti_*$ of the equation systems denote the probabilities of the complementary problems. Thus, the probabilities for availability and anticipability are given by $N\text{-}Avail_* = 1 - N\text{-}NoAvail_*$, $X\text{-}Avail_* = 1 - X\text{-}NoAvail_*$, $N\text{-}Anti_* = 1 - N\text{-}NoAnti_*$ and $X\text{-}Anti_* = 1 - X\text{-}NoAnti_*$.

Next we discuss PDFAs in more detail and specify the equation system for the two-edge approach. PDFAs as presented in [30, 38] are applicable for a large class of data flow problems called *finite bidistributive subset problems*. This class of data flow problems requires (1) a finite set of data flow facts and (2) the data flow functions that distribute over both set union and set intersection. This class of data flow problems is more general than bit-vector problems but less general than the class of finite distributive subset problems introduced by Reps et al. [39].

Availability Analysis:

—*LocAvail*$_{exp}(n)$: There is an expression *exp* in n which is available at the exit of n (downwards exposed).

—*LocBlkAvail*$_{exp}(n)$: The expression *exp* is blocked by some statement of n.

$$N - Avail_{exp}(n) = \begin{cases} false & if\ n = start\ node \\ \bigwedge\limits_{m \in pred(n)} X - Avail_{exp}(M) & otherwise \end{cases}$$

$$X - Avail_{exp}(n) = LocAvail_{exp}(n) \vee (N - Avail_{exp}(n) \vee \overline{LocBlkAvail_{exp}(n)})$$

Anticipability Analysis:

—*LocAnti*$_{exp}(n)$: There is a hoisting candidate *exp* in n (upwards exposed).

—*LocBlkAnti*$_{exp}(n)$: The hoisting of *exp* is blocked by some satement of n.

$$N - Anti_{exp}(n) = LocAnti_{exp}(n) \vee (X - Anti_{exp}(n) \wedge \overline{LocBlkAnti_{exp}(n)})$$

$$X - Anti_{exp}(n) = \begin{cases} false & if\ n = end\ node \\ \bigwedge\limits_{m \in succ(n)} X - Anti_{exp}(m) & otherwise \end{cases}$$

(a) Original Data Flow Equations.

Dual Availability Analysis:

$$N - NoAvail_{exp}(n) = \begin{cases} true & if\ n = start\ node \\ \bigvee\limits_{m \in pred(n)} X - NoAvail_{exp}(m) & otherwise \end{cases}$$

$$X - NoAvail_{exp}(n) = (LocBlkAvail_{exp}(n) \wedge \overline{LocAvail_{exp}(n)}) \vee \\ (N - NoAvail_{exp}(n) \wedge \overline{LocAvail_{exp}(n)})$$

Dual Anticipability Analysis:

$$N - NoAnti_{exp}(n) = (LocBlkAnti_{exp}(n) \wedge \overline{LocAnti_{exp}(n)}) \vee \\ (X - NoAnti_{exp}(n) \wedge \overline{LocAnti_{exp}(n)})$$

$$X - NoAnti_{exp}(n) = \begin{cases} true & if\ n = end\ node \\ \bigvee\limits_{m \in succ(n)} N - NoAnti_{exp}(m) & otherwise \end{cases}$$

(b) Data Flow Equations After Transformation.

FIGURE 4.12 Availability and anticipability analysis.

PDFA equations are built by utilizing a so-called *exploded control flow graph* (ECFG) [39], which is created from the original CFG and a data flow problem. Figure 4.13 depicts the ECFG for the availability problem for a subgraph of our example consisting of the nodes 1, 2, 3, 4 and 5.

The ECFG has $N \times (D \cup \{\Lambda\})$ nodes with N denoting the node set of the corresponding CFG and D denoting the data flow information set extended by the special symbol Λ used to calculate node frequencies. Data flow facts d_1 and d_2 designate expressions $x + y$ and $a + b$, respectively. The edges of the ECFG are derived from the representation relation [39] of a data flow function f, $R_f \subseteq (D \cup \{\Lambda\}) \times (D \cup \{\Lambda\})$, as follows:

$$\begin{aligned} R_f = &\{(\Lambda, \Lambda)\} \\ &\cup \{(\Lambda, y) | y \in f(\phi)\} \\ &\cup \{(x, y) | y \in f(\{x\}) \wedge y \notin f(\phi)\} \end{aligned}$$

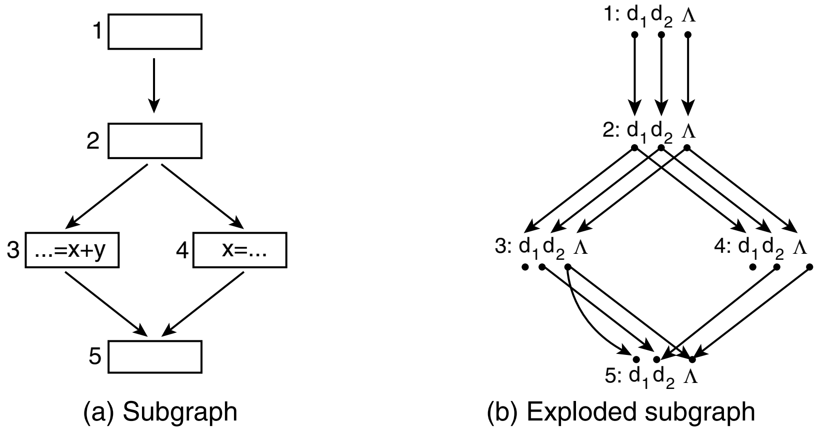


FIGURE 4.13 CFG and ECFG of subgraph of our example.

Note that nodes (n, Λ) are always connected with the successor nodes. Nodes 1 and 2 do not affect availability that is described by the identity function where all nodes are connected with the corresponding "exploded" successor nodes. However, because data flow fact d_1 is generated at node 3, we have an edge from $(3, \Lambda)$ to $(6, d_1)$. On the other hand, because data flow fact d_1 is killed at node 4, no outgoing edge exists from node $(4, d_1)$. In this way an ECFG represents DFA functions explicitly.

For the two-edge approach we require two-edge probabilities $p(u, v, w)$ that specify the probability that execution can follow edge vw once edge uv has been reached:

$$p(u, v, w) = \begin{cases} \frac{occurs(\mathrm{uvw}, \pi)}{occurs(\mathrm{uv}, \pi)} & \text{if } occurs(\mathrm{uv}, \pi) \neq 0, \\ 0 & \text{otherwise} \end{cases}$$

Function occurs denotes the number of occurrences of a path in the CFT π. Further, we need the notion of predecessor edge similar to predecessor node:

$$In(\mathrm{vw}, \delta) = \{(u, \delta'') | (u, \delta'') \to (v, \delta') \to (w, \delta) \in ECFG\}$$

$In(\mathrm{vw}, \delta)$ denotes the set of predecessor edges of edge vw with data fact δ in the ECFG. Finally, Figure 4.14 gives the equation system of the two-edge approach.

The unknowns of the equation system are related to the ECFG edges. Unknown $y(\mathrm{vw}, d)$ denotes the expected frequency of data flow fact d to hold true at node w under the condition that edge vw has been taken, and $y(\mathrm{vw}, \Lambda)$ denotes the expected number of times that edge vw is executed. In Figure 4.14 the core of the equation system is Equation (3). The unknown related to edge vw and fact δ is linked to all unknowns related to precessor edges weighting it with a two-edge probability.

For initialization reasons we introduce an artificial "root" edge $\epsilon\mathrm{s}$ from the pseudo node ϵ to the start node s. In Figure 4.14 Equation (1) and Equation (2) show the initialization of the equation system with $c(d)$ denoting the initial values of the corresponding data flow problem, Equation (4) describes the relation between nodes and edges in the ECFG. Once the equation system with the frequency unknowns has been solved, $Prob(v, d) = y(v, d)/y(v, \Lambda)$ denotes the probability of data flow fact d to hold true in node v. The two-edge equation system of Figure 4.14 has been presented in terms of forward data flow problems. For backward problems the two-edge probability and the equation system must be adapted accordingly.

The probabilities for availability of the two-edge approach coincide with the precise results for our example. Note that for solving the availability of $x + y$ precisely at conditional node 9, it is important to figure out whether path $3 - 6 - 9$ or $4 - 6 - 9$ has been taken. Fortunately, the two-edge approach

$$(1) \qquad y(\in s, \Lambda) = 1$$

for all *d* in *D*:

$$(2) \qquad y(\in s, d) = c(d)$$

for all vw in E: for all δ in D:

$$(3) \qquad y(vw, \delta) = \sum_{(u,\, \delta') \in In(vw, \delta)} p(u,v,w) * y(uv, \delta')$$

for all w in *N*: for all δ in *D*:

$$(4) \qquad y(w, \delta) = \sum_{u \in pred(v)} y(vw, \delta)$$

FIGURE 4.14 Two-edge equation system.

succeeds in enabling hoisting of both expressions at node 9. However, in general, the probabilities may deviate from the precise results such that beneficial opportunities for improvements are missed. Thus, a trade-off exists between preciseness of the results having possibly direct influence on the optimizations performed and the effort required to obtain those results.

4.3.3.3 Speculative Code Motion Framework

We next present the speculative code motion framework that is used to carry out the optimization. This framework hoists expressions to appropriate points in the program using safe code motion across conditionals at which speculation is disabled and speculative code motion across conditionals where speculation is enabled. A number of PRE algorithms can be found in the literature. We adapt the safe code motion based PRE algorithm proposed by Steffen [41].

The original analysis in [41] consists of a backward data flow analysis phase followed by a forward data flow analysis phase. The backward data flow is used to identify all down-safe points, that is, points to which expression evaluations can be safely hoisted. Forward data flow analysis identifies the earliest points at which expression evaluations can be placed. Finally, the expression evaluations are placed at points that are both earliest and down-safe. We modify the down-safety analysis to take advantage of speculation past the conditional nodes where speculation has been enabled. The earliestness analysis remains unchanged.

The placement points identified by the preceding algorithm are entry or exit points of nodes in the CFG. In some situations the flow graph may not contain a node at the appropriate placement point. To ensure this never happens, certain so-called *critical edges* are split and a synthetic node is introduced. These are edges from nodes with more than one successor to nodes with more than one predecessor. Therefore, in the flow graph of Figure 4.10, edge $6 \rightarrow 9$ is a critical edge. Thus, a node is introduced along this edge before performing the analysis for carrying out PRE.

The data flow equations for computing down-safety of an expression *exp* are as follows. An expression *exp* is down-safe at entry of node n if either it is computed in n (i.e., $Used_{exp}(n)$ is true) or it is down-safe at n's exit and preserved by n (i.e., $Pres_{exp}(n)$ is true). The expression *exp* is down-safe at the exit of a conditional node n if one of the following conditions is true: *exp* is down-safe at entry points of all of n's successor nodes; or speculation of *exp* has been enabled at n, that is, $AppEnabSpec_{exp}(n)$ is true, and *exp* is down-safe at least one of n's successor nodes. The first of the preceding two conditions was used in Steffen's original code motion framework as it carries out safe code motion. The second condition has been added to allow useful speculative

n	exp			
	x+y		a+b	
	X-DSAFE	N-DSAFE	X-DSAFE	N-DSAFE
1	F	F	F	F
2	F	F	F	F
3	T	T	F	F
4	T	F	F	F
5	T	T	T	T
6	T	T	F	F
7	T	T	T	T
8	T	T	T	F
8a	T	T	T	T
9	T	T	T	T
10	F	T	F	F
11	F	F	F	T
12	F	F	F	F
13	F	F	F	F

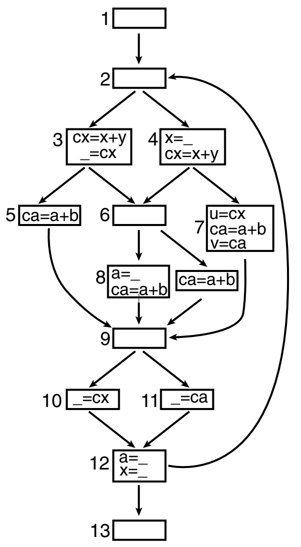


FIGURE 4.15 Speculative PRE based on modified down-safety analysis.

motion of *exp* to occur. If the node under examination is not a conditional node, then only the first of the two conditions is checked:

$$N - DSafe_{exp}(n) = Used_{exp}(n) \vee (Pres_{exp}(n) \wedge X - DSafe_{exp}(n))$$

$$X - DSafe_{exp}(n) = \begin{cases} \text{False} & n \text{ is the exit node} \\ \bigwedge_{m \in Succ(n)} DSafe_{exp}(m) & n \text{ is a conditional} \\ \vee(AppEnabSpec_{exp}(n) \wedge \bigvee_{m \in Succ(n)} Dsafe_{exp}(m)) & \\ \bigwedge_{m \in Succ(n)} DSafe_{exp}(m) & \text{otherwise} \end{cases}$$

The outcome of applying the preceding algorithm to the example of Figure 4.10 is shown in Figure 4.15. From the results of down-safety analysis we observe the following. The expressions $x + y$ and $a + b$ are both down-safe at the entry of node 9 because speculation is enabled at 9 for both expressions. Although we did not formally describe the earliestness analysis, it is easy to informally observe that the earliest points at which $x + y$ is down-safe are the entry point of node 3 (because $x + y$ is not down-safe at the exit of node 2) and the exit point of node 4 (because $x + y$ is not down-safe at the entry of node 4). Therefore, evaluations of $x + y$ are placed at these points and assigned to the temporary cx that replaces all the original occurrences of $x + y$ in the program. Similarly, we also observe that the earliest points at which $a + b$ is down-safe are the entry point of node 5, entry point of node 7, exit point of node 8 and entry point of node 8a. Evaluations of $a + b$ are placed at these four points and assigned to the temporary ca that replaces all original occurrences of $a + b$.

The optimized placement results in 10 evaluations of $x + y$ and $a + b$ reducing the evaluations of $x + y$ by 1 and $a + b$ by 5, respectively. Hence, the enabling of speculation for the two expressions at node 9 was profitable and resulted in more efficient code. This example also illustrates the need for breaking critical edges because an evaluation of $a + b$ has been placed in the newly created node 8a.

4.3.4 Cost of Analysis

When applied to the entire program, the compile-time cost of a profile-guided optimization algorithm can be expected to be higher than its non-profile-guided counterpart. For example, the speculative PRE algorithm of the preceding section is more expensive than the traditional safe code motion-based algorithm. Additional cost results from the cost-benefit analysis associated with enabling speculation. However, the following approaches can be applied to limit the cost of profile-guided algorithms:

- The application of optimization algorithms can be limited so that only the code belonging to frequently executed (hot) program regions is aggressively optimized.
- Instead of carrying out exhaustive data flow analysis, we can employ demand driven analysis techniques [16, 17, 24]. These techniques limit the cost of data flow analysis by only computing data flow facts that are relevant to a code optimization. This approach is particularly beneficial for expensive analyses (e.g., interprocedural analysis).
- Conservatively, imprecise frequency analysis techniques can be used to limit the cost of frequency analysis. A frequency analyzer for which cost can be controlled by permitting a bounded degree of imprecision was proposed by Bodik et al. [6]. In addition, this analyzer is demand-driven.

4.4 Profile-Guided Memory Optimizations

Over the past decade, whereas the processor speeds have risen by 55% per year, the memory speeds have only improved by 7% per year. As a result, the cache miss penalties have increased from several cycles to over 100 cycles. Because the memory accesses, especially those due to loads, are on the critical path, cache misses greatly reduce the ability of a modern processor to effectively exploit instruction level parallelism.

Broadly speaking, two classes of optimization techniques are aimed at improving cache performance: those that reduce the number of cache misses to minimize memory stalls and others that better tolerate cache miss penalty. The focus of this section is on optimizations of the first type.

A program's address space is divided into three parts: heap, stack and global space. Stack data typically exhibit good cache behavior. Therefore, most of the research has been aimed at improving the cache locality for statically allocated global data and dynamically allocated heap data. Although the optimizations for statically allocated data are typically carried out at compile time, optimizations applicable to dynamically allocated data must use a combination of compile time decisions and runtime actions. The techniques used to improve data locality can be divided into the following categories:

- *Object placement* techniques determine a placement of data objects in relation to each other that provides improved cache behavior.
- *Object layout* techniques determine a layout of fields within a large data object to improve cache locality.
- *Object layout and placement* techniques alter both the layout of fields within an object as well as the placement of data objects in relation to each other.
- *Object compression* techniques improve cache behavior by reducing the data memory footprint of a program

4.4.1 Object Placement

The address profiles provide the compiler with the information useful in organizing the placement of objects in memory in relation to one another. For example, we can categorize objects as frequently referenced hot objects and infrequently accessed cold objects. The hot objects can be placed together separate from the cold objects. By further seeing which hot objects are frequently used together, the placement of hot objects in relation to each other can be determined.

For statically allocated global data, object placement can be carried out at one time during compilation. However, the placement of heap objects can only be carried incrementally and at runtime as they are dynamically allocated. Standard system provided memory allocators (e.g., malloc) do not provide any control over data placement. The two proposed approaches for placement of heap objects are memory clustering and object migration.

4.4.1.1 Object Migration

In this approach standard memory allocators are used; and therefore when the heap objects are initially allocated, their placement is not optimized. At runtime the objects are migrated from their original locations to other ones to improve cache locality. Two main challenges of carrying out effective object migration exist.

First, a substantial overhead of object migration exists. The two sources of this overhead are object migration, which requires extra execution time, and access to migrated objects, which may require additional operations in some situations. If the gain of improved cache locality outweighs the migration overhead, only then is the object migration optimization profitable. The role of address profiles is to identify objects that should be colocated through migration.

Second, the challenge is that of maintaining program correctness in the presence of object migration. If the programmer makes use of location of data in developing the code, clearly object migration can lead to correctness problems. For example, consider the objects corresponding to variable a and structure pointed to by p shown as follows. The user may assume that the objects are colocated and use address arithmetic in accessing the fields. In particular, the user may access $p \rightarrow c$ data field using $\&a + 2 * sizeof(int)$. Clearly, object migration can cause such a code to fail.

```
int    a;
struct {
   int    b;
   int    c;
} *p;
```

One approach to addressing the correctness issue is to provide programming guidelines that restrict the manner in which data can be accessed. Address arithmetic can be allowed but only within an object; that is, we can prohibit the user from deriving the address of one object from the address of another object. This restriction would eliminate the problem illustrated in the preceding example. Another approach is to provide hardware support that may enable safe application of object migration in some situations.

Luk and Mowry [28] have proposed hardware support for memory forwarding that uses limited hardware support to enable aggressive object migration. The problem addressed by this support is as follows. When an object is migrated, pointers to that object may exist elsewhere in the program. Because these pointers are out of date, any accesses through them are illegal. To correctly handle accesses to migrated objects through such pointers the following support is provided. An extra bit is attached to each machine word that has the following semantics. If the bit attached is set, it indicates that the object that resided at that location has been migrated and the location now contains a forwarding address for the migrated object. The load and store instructions are implemented to carry out the extra level of dereferencing based on the contents of the bit attached to the memory location addressed. Thus, this approach enables migration of objects if forwarding addresses are left behind at locations where the objects originally resided. It should be noted that extra overhead is

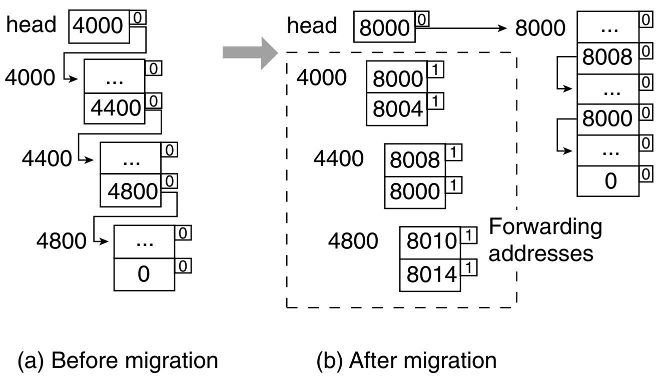


FIGURE 4.16 Memory forwarding.

associated with memory forwarding both in terms of extra storage that is needed to hold forwarding addresses and extra execution time required to carry out extra level of dereferencing.

The example in Figure 4.16 illustrates the preceding approach. It shows a link list where initially all the items in the link list reside at their originally assigned locations. The link list after some of the elements have been migrated to adjacent memory locations is shown next. The list elements are migrated so that fewer cache misses occur when the list is traversed. Note that the original locations of the migrated list elements now contain forwarding addresses. If any dangling pointers still point to these old addresses, they are forwarded to the new addresses by the hardware.

4.4.1.2 Memory Clustering

This approach supports memory management routines that provide limited user level control over object placement. Therefore, unlike object migration that dynamically changes object placement, this approach places objects in more desirable locations to begin with. As a result the migration overhead is avoided. However, this approach burdens the programmer with the task of providing hints to the memory allocator for making good object placement decisions.

An example of a user level memory allocator is `ccmalloc` allocator proposed by Chilimbi et al. [11]. In addition to providing the size of the object during memory allocation, the user provides a pointer to an existing object that is likely to be accessed contemporaneously with the newly allocated object. Whenever possible, `ccmalloc` allocates the new object in the same cache block as the existing object. The address profiles can be used by the user to identify objects that are accessed contemporaneously.

In Figure 4.17 we show the grouping of nodes of a binary tree with and without clustering. Group of nodes belonging to the same cache block are shown by the shading in Figure 4.17. The grouping prior to clustering reflects the order in which the nodes are created. The grouping following the use of `ccmalloc` allocates a group of neighboring nodes to the same cache block. By assuming that this decision was based on address profiles of perhaps a breadth first traversal of the tree, the optimized placement of tree nodes can lead to better cache performance.

Although the preceding approach requires a programmer's involvement, it is also possible to automate this process using address profiles. We can identify references that appear close to each other in the address profiles and generate a signature for these group of references. When creating nodes at runtime, an attempt can be made to colocate the nodes with the same signature.

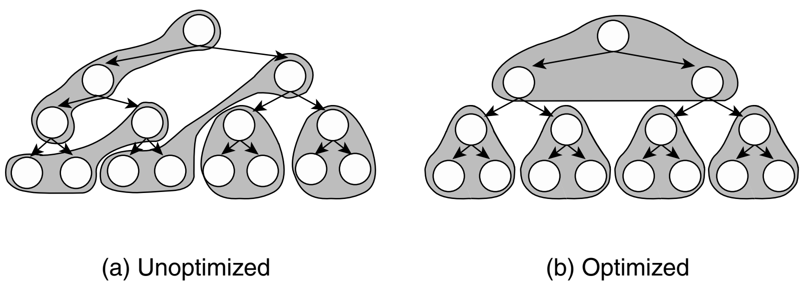


FIGURE 4.17 A ccmalloc example.

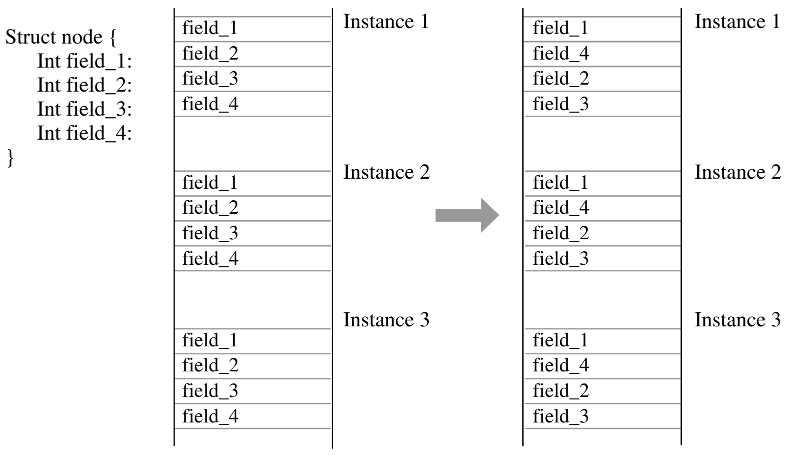


FIGURE 4.18 Field reorganization.

4.4.2 Object Layout

A data structure is often defined by the programmer to support code readability. Therefore, logically related data fields are often put together. The compiler simply uses a memory layout for the fields that mirrors the order in which the fields are declared. However, this order may not be consistent with the ordering that incurs fewer cache misses. To improve cache performance, the layout of fields within an object can be reordered so that the fields that are accessed frequently and contemporaneously are placed close to each other. Of course, this transformation is only useful if the object is large enough that it extends across multiple cache blocks. Truong et al. [34] evaluated this approach and showed that when a node spans several cache blocks, object layout optimization implicitly takes advantage of cache line prefetching and reduces cache pollution, thus improving cache performance.

Figure 4.18 shows the default memory layout used by the compiler on the left. On the right the layout generated after profile-guided field reorganization is shown under the assumption that the cache block is large enough to hold two fields and fields 1 and 4 are used together. A pair of accesses of fields 1 and 4 would generate two cache misses before reorganization and only one cache miss after reorganization.

4.4.3 Object Layout and Placement

Some highly aggressive locality improving transformations result in simultaneous changing of object layouts and their placement with respect to each other. Consider a data structure that contains multiple objects (or nodes) of the same type. Moreover, each object contains several fields; some are hot

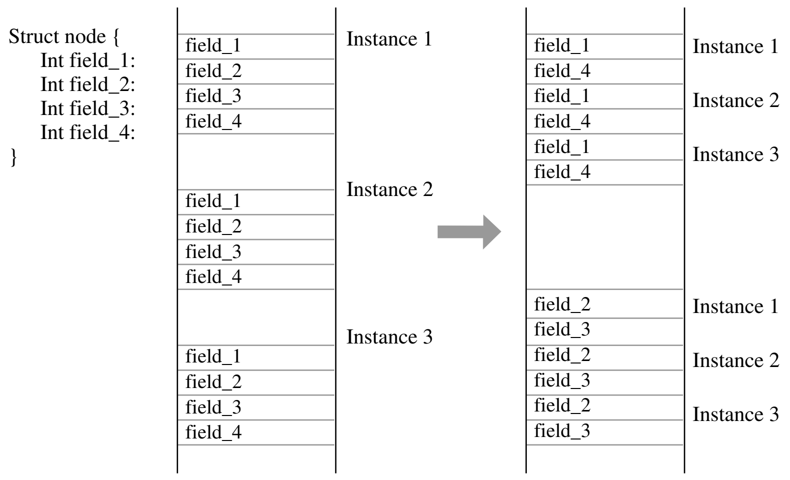


FIGURE 4.19 Interleaving instances of the same type.

(frequently used) whereas others are cold (rarely used). Two transformations, instance interleaving and object splitting, have been proposed to improve the cache behavior of such data structures.

4.4.3.1 Instance Interleaving

Instead of allocating an object in a contiguous set of memory locations, this approach proposed by Truong et al. [34] interleaves the storing of multiple instances of object instances of the same type. Hot fields from different object instances are stored together and so are the cold instances. The application of this transformation to the example of Figure 4.18 is illustrated in Figure 4.19. The hot fields (1 and 4) from different object instances are stored together in one place and the cold fields (2 and 3) from different object instances are also stored together. Clearly, from this example we can observe that both the internal layout and the relative placement of objects are changed by this transformation.

4.4.3.2 Object Splitting

Chilimbi et al. [12] also proposed a transformation that separates hot fields within an object from the cold fields. In fact, they split an object into two parts, the hot primary part and the cold secondary part. Hot fields are accessed directly, whereas a pointer to the cold part is stored within the hot part and thus an extra level of indirection is involved in accessing cold fields. This transformation does not explicitly interleave the hot (cold) parts from different object instances. The locality across objects can be improved by using memory clustering (e.g., `ccmalloc`) to group together hot (or cold) fields from different object instances. The example in Figure 4.20 illustrates this transformation. Compared with instance interleaving, this approach is much cleaner and easy to manage because the modification to the original source code is easier to perform.

It is important to note that changing an object layout alone is not useful if the object is small and fits within a cache block. However, when layout and placement are simultaneously changed, it is beneficial to even split small objects because the splitting can allow clustering of a greater number of hot parts of objects into a single cache block.

4.4.4 Object Compression

A complimentary technique for improving data cache behavior is that of object compression. This technique improves cache performance by reducing the memory requirement by packing greater

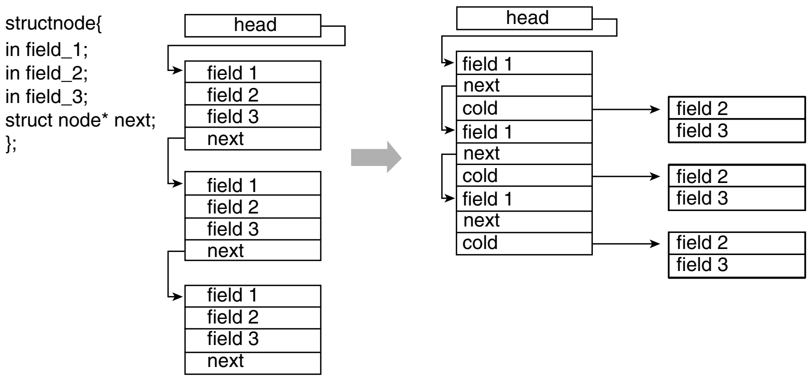


FIGURE 4.20 Object splitting.

amounts of data per cache block. Higher data density per cache block reduces the number of cache misses that take place when accessing a given amount of data from memory.

Compile-time techniques have been developed to exploit the presence of narrow width data to achieve compression. Stephenson et al. [33] have proposed bit width analysis that is used to discover narrow width data by performing value range analysis. For example, a flag declared as an integer may take only 0 or 1 values. Once the compiler has proved that certain data items do not require a complete word of memory, they are compressed to a smaller size. This approach is particularly useful for programs with narrow width multimedia data that are not packed by the user. The work by Davidson and Jinturkar [14] also carries out memory; coalescing using compile-time analysis. The preceding techniques share one common characteristic, they are applicable only when the compiler can determine that the data compressed are fully compressible and they only apply to narrow width nonpointer data.

Zhang and Gupta [45] have proposed profile-guided compression transformations that apply to partially compressible data, and in addition to handling narrow width nonpointer data they also apply to pointer data. Therefore, these transformations are quite effective in compressing heap objects. The generality of this approach is quite important because experience has shown that although heap allocated data structures are highly compressible, they are almost never fully compressible. This approach is also simpler in one respect. It does not require complex compile-time analysis to prove that the data are always compressible. Instead simple profile-guided heuristics can be used by the compiler to determine that the data structure compressed contains mostly compressible data.

Compression of partially compressible data structures is based on the observation by Zhang et al. [44] that majority of the data values encountered in data memory allocated to a program can be divided into two categories: very small constant values and very large address values. In a node belonging to a heap allocated data structure we are likely to have some data fields that may contain small constants and pointer fields that contain large address values. If the 18 higher order bits of small constants are the same, then we can truncate the value to 15 lower order bits because the remaining bits can be simply obtained by replicating the highest order bit of this truncated entity. If the pointer field stored in a node shares a common 17 bit prefix with the address at which the node itself is stored, then we can truncate the pointer field also to 15 bits. This is because the full contents of the pointer can always be reconstructed from the address at which the node resides. Two truncated 15 bit entities, each representing either integer data or a pointer, can be compressed into a single word. Of the two remaining bits of a 32 bit word, one is unused and the other is used to indicate whether the word contains two compressed values. The word cannot contain two compressed values in case the values are not found to be compressible at runtime. In this case it contains a pointer to a location where the two fields are stored in uncompressed form.

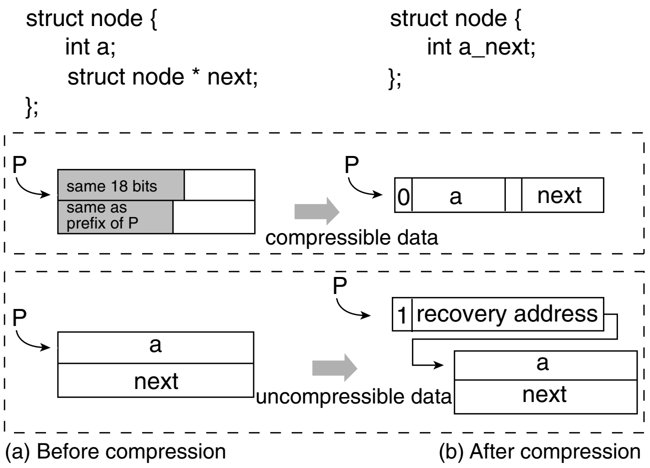


FIGURE 4.21 Compression of partially compressible data structure.

The benefits of the preceding optimization increase with the extent to which large and critical data structures in the program can be compressed. The cost of the optimization is the additional instructions for carrying out data compression, expanding compressed data prior to their use in computations, checking the compressibility of data and accessing uncompressible data through an extra level of indirection. If profiling indicates that the values in a data structure are mostly compressible, then the benefits are found to outweigh the costs.

In Figure 4.21 the node structure contains an integer field `a` and a pointer field `next`. Let us assume that profiling indicates that these fields are mostly compressible. In this situation the compiler replaces them by a single combined field. At runtime when a node is created, a reduced amount of storage is allocated to accommodate field `a_next`. If the data stored are found to be compressible, then they are stored in this single word as shown in Figure 4.21. The most significant bit is set to 0 if the data are held by this compressed field. On the other hand, if the data stored are not compressible, additional storage is allocated to store the data and a pointer to this location is placed in `a_next`. The most significant bit of `a_next` is set to 1.

4.4.5 Profile-Guided Code Layout

The techniques discussed so far are applicable to program data. Techniques have also been proposed to improve instruction cache behavior [29, 42]. The goal of such techniques is to layout the code in a manner that increases the fraction of instructions per fetched instruction cache block that is actually executed. Note that all instructions belonging to a cache block may not be executed due to the presence of branches in the code.

Consider the example in Figure 4.22: the sizes of cache line and basic blocks (a, b, c and d) are shown. For illustration purposes, let us assume that we have a two-entry fully associative cache. If no layout optimizations are performed, it is likely that the compiler can generate the layout marked as the original layout in the figure. This layout introduces a large number of instruction cache misses primarily because the blocks a, b and d that are on the frequently executed path cannot be all kept in the cache. Meanwhile, block c is fetched repeatedly but is rarely executed. By using path profile data we can easily determine that it is more beneficial to use the optimized layout shown in Figure 4.22 in which blocks b and d are packed into the same cache line. The number of cache misses is greatly reduced in this case. We have illustrated the application of code layout optimization at the basic block level. Techniques for layout optimization at the procedural level have also been developed [29].

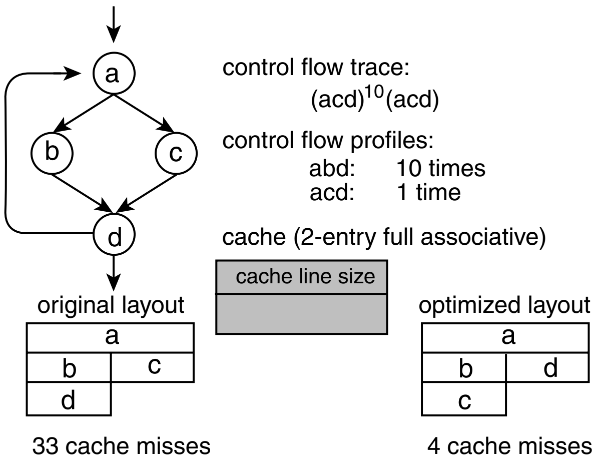


FIGURE 4.22 Code placement.

4.5 Concluding Remarks

In this chapter we identify optimization opportunities that may exist during program execution but cannot be exploited without the availability of profile data. Different types of profile data that are useful for code optimization are identified. The use of this profile data in carrying out profile-guided classical and memory optimizations is discussed. We identify the key issues involved in developing profile-guided optimization algorithms and demonstrate these issues by developing a speculative PRE algorithm.

References

[1] A.-H.A. Badawy, A. Aggarwal, D. Yeung and C.-W. Tseng, Evaluating the Impact of Memory System Performance on Software Prefetching and Locality Optimizations, in 15th Annual ACM International Conference on Supercomputing (ICS), Sorrento, Italy, June 2001, pp. 486–500.

[2] T. Ball and J.R. Larus, Efficient Path Profiling, in 29th IEEE/ACM International Symposium on Microarchitecture (MICRO), December 1996, pp. 46–57.

[3] T. Ball, P. Mataga and M. Sagiv, Edge Profiling versus Path Profiling: The Showdown, in 25th ACM SIGPLAN-SIGACT Symposium on Principle of Programming Languages (POPL), San Diego, CA, January 1998, pp. 134–148.

[4] R. Bodik, R. Gupta and M.L. Soffa, Interprocedural Conditional Branch Elimination, in ACM SIGPLAN Conference on Programming Language Design and Implementation, Las Vegas, NV, June 1997, pp. 146–158.

[5] R. Bodik and R. Gupta, Partial Dead Code Elimination Using Slicing Transformations, in ACM SIGPLAN Conference on Programming Language Design and Implementation, Las Vegas, NV, June 1997, pp. 159–170.

[6] R. Bodik, R. Gupta and M.L. Soffa, Complete Removal of Redundant Expressions, in ACM SIGPLAN Conference on Programming Language Design and Implementation, Montreal, Canada, June 1998, pp. 1–14.

[7] R. Bodik, R. Gupta and M.L. Soffa, Load-Reuse Analysis: Design and Evaluation, in ACM SIGPLAN Conference on Programming Language Design and Implementation, Atlanta, GA, May 1999, pp. 64–76.

[8] R. Bodik, R. Gupta and V. Sarkar, ABCD: Eliminating Array Bounds Checks on Demand, in ACM SIGPLAN Conference on Programming Language Design and Implementation, Vancouver, Canada, June 2000, pp. 321–333.

[9] B. Calder, P. Feller and A. Eustace, Value Profiling, in 30th IEEE/ACM International Symposium on Microarchitecture (MICRO), December 1997, pp. 259–269.

[10] B. Calder, C. Krintz, S. John and T. Austin, Cache-Conscious Data Placement, in 8th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), San Jose, CA, October 1998, pp. 139–149.

[11] T.M. Chilimbi, M.D. Hill and J.R. Larus, Cache-Conscious Structure Layout, in ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Atlanta, GA, May 1999, pp. 1–12.

[12] T.M. Chilimbi, B. Davidson and J.R. Larus, Cache-Conscious Structure Definition, in ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Atlanta, GA, May 1999, pp. 13–24.

[13] T.M. Chilimbi, Efficient Representations and Abstractions for Quantifying and Exploiting Data Reference Locality, in ACM SIGPLAN Conference on Programming Language Design and Implementation, Snowbird, UT, June 2001, pp. 191–202.

[14] J. Davidson and S. Jinturkar, Memory Access Coalescing: A Technique for Eliminating Redundant Memory Accesses, in ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Orlando, FL, June 1994, pp. 186–195.

[15] D.M. Dhamdhere, Practical adaptation of global optimization algorithm of Morel and Renvoise, *ACM Trans. Programming Languages*, 13(2), 291–294, 1991.

[16] E. Duesterwald, R. Gupta and M.L. Soffa, A practical framework for demand-driven interprocedural data flow analysis, *ACM Trans. Programming Languages Syst.*, 19(6), 992–1030, November 1997.

[17] E. Duesterwald, R. Gupta and M.L. Soffa, Demand-Driven Computation of Interprocedural Data Flow, in ACM SIGPLAN-SIGACT 22nd Symposium on Principles of Programming Languages (POPL), San Francisco, CA, January 1995, pp. 37–48.

[18] C. Dulong, The IA-64 Architecture at Work, in IEEE Comput., 31(7), 24–32, July 1998.

[19] R. Gupta, A Fresh Look at Optimizing Array Bound Checks, in ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), White Plains, NY, June 1990, pp. 272–282.

[20] R. Gupta, D. Berson and J.Z. Fang, Path Profile Guided Partial Redundancy Elimination Using Speculation, in IEEE International Conference on Computer Languages (ICCL), Chicago, IL, May 1998, pp. 230–239.

[21] R. Gupta, D. Berson and J.Z. Fang, Resource-Sensitive Profile-Directed Data Flow Analysis for Code Optimization, in IEEE/ACM 30th International Symposium on Microarchitecture (MICRO), Research Triangle Park, NC, December 1997, pp. 358–368.

[22] R. Gupta, D. Berson and J.Z. Fang, Path Profile Guided Partial Dead Code Elimination Using Predication, in International Conference on Parallel Architectures and Compilation Techniques (PACT), San Francisco, CA, November 1997, pp. 102–115.

[23] R.E. Hank, W.-M. Hwu and B.R. Rau, Region-based Compilation: An Introduction and Motivation, in 28th IEEE/ACM International Symposium on Microarchitecture (MICRO), Ann Arbor, MI, 1995, pp. 158–168.

[24] N. Heintze and O. Tardieu, Demand-Driven Pointer Analysis, in ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, UT, June 2001, pp. 24–34.

[25] J. Knoop, O. Ruthing and B. Steffen, Lazy Code Motion, in ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), San Francisco, CA, June 1992, pp. 224–234.

[26] J. Knoop, O. Ruthing and B. Steffen, Partial Dead Code Elimination, in ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Orlando, FL, June 1994, pp. 147–158.

[27] J. Larus, Whole Program Paths, ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Atlanta, GA, May 1999, pp. 259–269.

[28] C.-K. Luk and T.C. Mowry, Memory Forwarding: Enabling Aggressive Layout Optimizations by Guaranteeing the Safety of Data Relocation, in 26th IEEE/ACM International Symposium on Computer Architecture (ISCA), May 1999, pp. 88–99.

[29] S. McFarling, Program Optimization for Instruction Caches, in 3rd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), April 1989, pp. 183–191.

[30] E. Mehofer and B. Scholz, A Novel Probabilistic Data Flow Framework, in 10th International Conference on Compiler Construction (CC), Genova, Italy, *Lecture Notes in Computer Science*, Vol. 2027, Springer-Verlag, April 2001, pp. 37–51.

[31] C.G. Nevil-Manning and I.H. Witten, Linear-time, Incremental Hierarchy Inference for Compression, in Data Compression Conference, Snowbird, UT, IEEE Computer Society, 1997, pp. 3–11.

[32] V. Sarkar, Determining Average Program Execution Times and Their Variance, in ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Portland, OR, June 1989, pp. 298–312.

[33] M. Stephenson, J. Babb and S. Amarasinghe, Bitwidth Analysis with Application to Silicon Compilation, in ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, Canada, June 2000, pp. 108–120.

[34] D.N. Truong, F. Bodin and A. Seznec, Improving Cache Behavior of Dynamically Allocated Data Structures, in International Conference on Parallel Architectures and Compilation Techniques (PACT), Paris, France, 1998, pp. 322–329.

[35] S. Watterson and S. Debray, Goal-Directed Value Profiling, in *10th International Conference on Compiler Construction (CC)*, LNCS *Lecture Notes in Computer Sciences*, Vol. 2027, Springer-Verlag, New York, 2001.

[36] E. Morel and C. Renvoise, Global optimization by suppression of partial redundancies, *Commun. ACM*, 22(2), 96–103, 1979.

[37] F. Mueller and D. Whalley, Avoiding Conditional Branches by Code Replication, in ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), La Jolla, CA, June 1995, pp. 56–66.

[38] G. Ramalingam, Data Flow Frequency Analysis, in ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Philadelphia, PA, 1996, pp. 267–277.

[39] T. Reps, S. Horwitz and M. Sagiv, Precise Interprocedural Dataflow Analysis Via Graph Reachability, in ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL), San Francisco, CA, January 1995, pp. 49–61.

[40] B. Steffen, Property Oriented Expansion, in *International Static Analysis Symposium* (SAS), Germany, September 1996, *Lecture Notes in Computer Science*, Vol. 1145, Springer-Verlag, New York, 1996, pp. 22–41.

[41] B. Steffen, Data Flow Analysis as Model Checking, *TACS*, Sendai, Japan, *Lecture Notes in Computer Science*, Vol. 526, Springer-Verlag, New York, 1991, pp. 346–364.

[42] C. Young, D.S. Johnson, D.R. Karger and M.D. Smith, Near-optimal Intraprocedural Branch Alignment, in ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Las Vegas, NV, June 1997, pp. 183–193.

[43] Y. Zhang and R. Gupta, Timestamped Whole Program Path Representation, in ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, UT, June 2001, pp. 180–190.

[44] Y. Zhang, J. Yang and R. Gupta, Frequent Value Locality and Value-Centric Data Cache Design, in 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Cambridge, MA, November 2000, pp. 150–159.

[45] Y. Zhang and R. Gupta, Data Compression Transformations for Dynamically Allocated Data Structures, in International Conference on Compiler Construction, Grenoble, France, April 2002.

6 Optimizations for Object-Oriented Languages

Andreas Krall
Technische Universität Wien

Nigel Horspool
University of Victoria

6.1 Introduction

This chapter gives an introduction to optimization techniques appropriate for object-oriented languages. The topics covered include object and class layout, method invocation, efficient runtime-type checks, devirtualization with type analysis techniques and escape analyses. Object allocation and garbage collection techniques are also very important for the efficiency of object-oriented programming languages. However, because of their complexity and limited space, this important topic must unfortunately be omitted. A good reference is the book by Jones and Lins [16].

Optimization issues relevant to a variety of programming languages, including C++, Java, Eiffel, Smalltalk and Theta, are discussed. However, to ensure consistent treatment, all examples have been converted to Java syntax. When necessary, some liberties with Java syntax, such as true multiple inheritance, have been made.

0-8493-1240-7/03/$0.00+$1.50

6.2 Object Layout and Method Invocation

The memory layout of an object and how the layout supports dynamic dispatch are crucial factors in the performance of object-oriented programming languages. For single inheritance there are only few efficient techniques: dispatch using a virtual dispatch table and direct calling guarded by a type test. For multiple inheritance many different techniques with different compromises are available: embedding superclasses, trampolines and table compression.

6.2.1 Single Inheritance

In the case of single inheritance, the layout of a superclass is a prefix of the layout of the subclass. Figure 6.1 shows the layouts of an example class and subclass. Access to instance variables requires just one load or store instruction. Adding new instance variables in subclasses is simple.

Invocation of virtual methods can be implemented by a method pointer table (virtual function table, *vtbl*). Each object contains a pointer to the virtual method table. The *vtbl* of a subclass is an extension of the superclass. If the implementation of a method of the superclass is used by the subclass, the pointer in the *vtbl* of the subclass is the same as the pointer in the superclass.

Figure 6.2 shows the virtual tables for the classes `Point` and `ColorPnt` defined as follows:

```
class Point {
    int x, y;
    void move(int x, int y) {...}
    void draw() {...}
    }

class ColorPnt extends Point {
    int color;
    void draw() {...}
    void setcolor(int c) {...}
    }
```

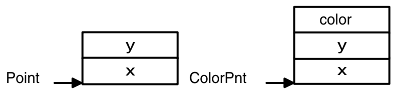


FIGURE 6.1 Single inheritance layout.

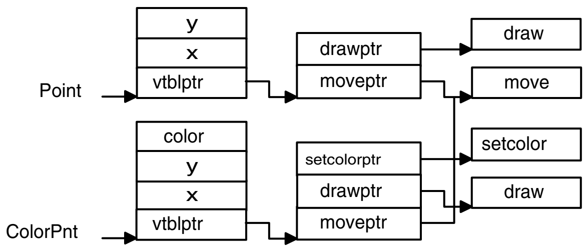


FIGURE 6.2 Single inheritance layout with virtual method table.

Each method is assigned a fixed offset in the virtual method table. Method invocation is just three machine code instructions (two load instructions and one indirect jump):

```
LD  vtblptr,(obj)          ; load vtbl pointer
LD  mptr,method(vtblptr)   ; load method pointer
JSR (mptr)                 ; call method
```

Dynamic dispatching using a virtual method table has the advantage that it is fast and executes in constant time. It is both possible to add new methods and to override old ones. One extra word of memory is needed for every object. On modern architectures, load instructions and indirect jumps are expensive. Therefore, Rose [22] suggested fat dispatch tables, where the method code is directly placed in the virtual method table eliminating one indirection. The problem with fat dispatch tables is that the offsets for different method implementations must be the same. Either memory is wasted or large methods must branch to overflow code.

6.2.2 Multiple Inheritance

While designing multiple inheritance for C++, Stroustrup also proposed different implementation strategies [26]. Extending the superclasses as in single inheritance does not work anymore. The fields of the superclass are embedded as a contiguous block. Figure 6.3 demonstrates embedding for class `ColorPnt`, which is defined as follows:

```
class Point {
    int x, y;
    }

class Colored {
    int color;
    }

class ColorPnt extends Point, Colored {}
```

Embedding allows fast access to instance variables exactly as in single inheritance. The object pointer is adjusted to the embedded object whenever explicit or implicit pointer casting occurs (assignments, type casts, parameter and result passing). Pointer adjustment has to be suppressed for casts of `null` pointers:

```
Colored    col;
Colorpoint cp;
col = cp;  // col=cp; if (cp!=null)col=(Colored)((int*)cp+2)
```

In C++ the pointer adjustments break if type casts outside the class hierarchy are used (e.g., casting to `void*`). Garbage collection becomes more complex because pointers also point into the middle of objects.

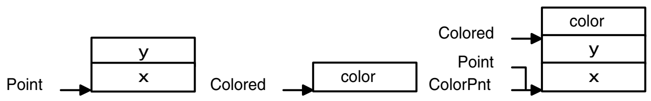


FIGURE 6.3 Multiple inheritance layout.

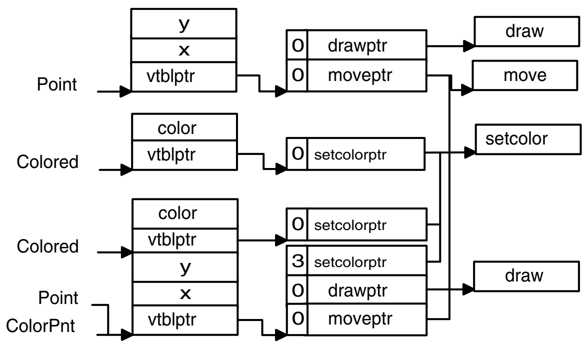


FIGURE 6.4 Multiple inheritance layout with virtual method table.

Dynamic dispatching also can be solved for embedding. For every superclass, virtual method tables have to be created and multiple *vtbl* pointers are included in the object. A problem occurs with implicit casts from the actual receiver to the formal receiver. The caller does not know the type of the formal receiver in the callee, and the callee does not know the type of the actual receiver of the caller. Therefore, this type information has to be stored as an adjustment offset in the virtual method table. Given the following definition for class `ColorPnt`, the virtual tables are organized as shown in Figure 6.4.

```
class Point {
    int x, y;
    void move(int x, int y) {...}
    void draw() {...}
    }

class Colored {
    int color;
    void setcolor(int c) {...}
    }

class ColorPnt extends Point, Colored {
    void draw() {...}
    }
```

Method invocation now requires four to five machine instructions, depending on the computer architecture:

```
LD  vtblptr,(obj)           ; load vtbl pointer
LD  mptr,method_ptr(vtblptr) ; load method pointer
LD  off,method_off(vtblptr)  ; load adjustment offset
ADD obj,off,obj             ; adjust receiver
JSR (mptr)                  ; call method
```

This overhead in table space and program code is even necessary when multiple inheritance is not used. Furthermore, adjustments to the remaining parameters and the result are not possible. A solution that eliminates much of the overhead is to insert a small piece of code called a *trampoline* that performs the pointer adjustments and then jumps to the original method. The advantages are a smaller table (no storing of an offset) and fast method invocation when multiple inheritance is not used (the same dispatch code as in single inheritance). In the example of Figure 6.4, the `setcolorptr` method pointer in the virtual method table of `Colorpoint` would point to code which adds three to the receiver before jumping to the code of method `setcolor`:

```
ADD obj,3,obj                 ; adjust receiver
BR  setcolor                  ; call method
```

When instance variables of common superclasses need to be shared, the offset of each such variable has to be stored in the virtual method table. Each access to a shared variable then incurs an additional penalty of loading and adding the appropriate offset.

6.2.3 Bidirectional Object Layout

The programming language Theta [20], like Java, uses single inheritance with multiple subtyping. For this language, Myers proposed a bidirectional object layout and showed how the bidirectional layout rules can be extended to support true multiple implementation inheritance [21]. The idea behind bidirectional layout is that data structures can be extended in two directions and information can be shared in a way that leads to less indirection and smaller memory usage. Both the object and virtual method table extend in both directions. The object contains the instance variable fields, the pointer to the object's virtual method table, with negative offsets the pointer to the interface method tables. The method dispatch tables also extend in both directions. The superclass information fields are in the middle of the tables, and subclass fields are at both ends of the tables. Figure 6.5 shows the object and method table layout scheme.

The key to the efficiency of the bidirectional layout is the merging of interface and class headers. Determining an optimal layout is not feasible; therefore a heuristic is used. The details of the algorithm can be found in the original article by Myers.

Given classes `C1` and `C2` defined as follows, the bidirectional layout scheme would be as shown in Figure 6.6:

```
interface A {                      interface B extends A{
    int a1() {...}                     int b1() {...}
    int a2() {...}                     }
    }

class C1 implements A {            class C2 implements B {
    int v1;                            int v2;
    int a1() {...}                     int a2() {...}
    int a2() {...}                     int b1() {...}
    int c1() {...}                     int c2() {...}
    }                                  }
```

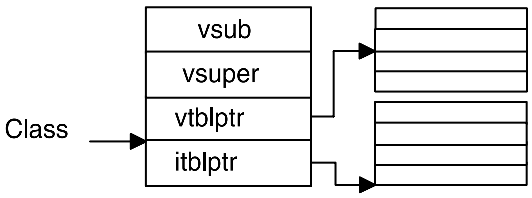


FIGURE 6.5 Bidirectional object layout with dispatch tables.

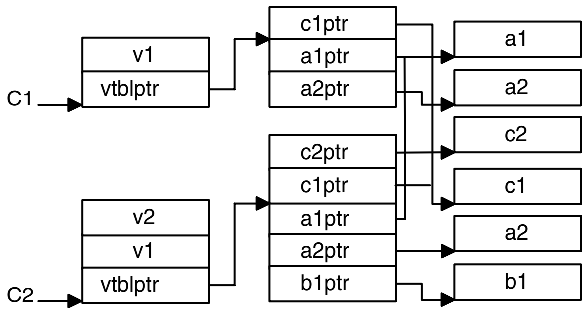


FIGURE 6.6 Bidirectional object layout with virtual method table.

As Figure 6.6 shows, the bidirectional layout reduces object and *vtbl* sizes. No additional dispatch headers and method tables are needed.

In a large C++ library with 426 classes, bidirectional layout only needs additional dispatch headers in 27 classes with a maximum of 4 dispatch headers compared with 54 classes with more than 7 dispatch headers.

6.2.4 Dispatch Table Compression

Invoking a method in an object-oriented language requires looking up the address of the block of code that implements that method and passing control to it. In some cases, the lookup may be performed at compile time. Perhaps there is only one implementation of the method in the class and its subclasses; perhaps the language has provided a declaration that forces the call to be nonvirtual; perhaps the compiler has performed a static analysis that can determine a unique implementation is always called at a particular call site. In other cases, a runtime lookup is required.

In principle, the lookup can be implemented as indexing a two-dimensional table. Each method name in the program can be given a number, and each class in the program can be given a number. Then, the method call:

```
result = obj.m(a1,a2);
```

can be implemented by these three actions:

1. Fetch a pointer to the appropriate row of the dispatch table from the `obj` object.
2. Index the dispatch table row with the method number.
3. Transfer control to the address obtained.

Note that if two classes `C1` and `C2` are not related by an ancestor relationship or do not have a subclass in common (due to multiple inheritance), and if the language is strongly typed (as with C++ and Java), then the numbers assigned to the methods of `C1` do not have to be disjoint from the numbers used for the methods of `C2`.

The standard implementation of method dispatch in a strongly typed language such as C++ using a virtual table (*vtbl*) may be seen to be equivalent. Each virtual table implements one row of the two-dimensional dispatch table.

The size of the dispatch table (or of all the virtual tables summed together) can be appreciable. As reported in [28], the dispatch table for the ObjectWorks Smalltalk class library would require approximately 16 MB if no compression were to be performed. In this library, there are 776 classes and 5325 selectors (i.e., method names).

For a statically typed language, virtual tables provide an effective way to compress the dispatch table because all entries in each virtual table are used. For a dynamically typed language, such as Smalltalk, any object can in principle be invoked with any method name. Most methods are not implemented and therefore most entries in the dispatch table can be filled in with the address of the "not implemented" error reporting routine. The table for dynamically typed languages therefore tends to be quite sparse, and that is a property that can be exploited to achieve table compression.

A second property that is possessed by the dispatch tables for both statically and dynamically typed languages is that many rows tend to be similar. When a subclass is defined, only some of the methods in the parent class are normally redefined. Therefore, the rows for these two classes would have identical entries for all except the few redefined methods.

Both properties are also possessed by LR parser tables and there has been considerable research into compressing such tables. A comprehensive survey of parse table compression is provided by Dencker et al. [11]. It should not be surprising that two of the most effective techniques for compressing static dispatch tables have also been used for parse table compression.

6.2.4.1 Virtual Tables

As noted earlier, virtual tables provide an effective implementation of the dispatch table for statically typed languages. Because methods can be numbered compactly for each class hierarchy to leave no unused entries in each virtual table, a good degree of compression is automatically obtained. For the ObjectWorks example used in [28], and if virtual tables could be used (they cannot), the total size of the dispatch tables would be reduced from 16 MB to 868 KB.

6.2.4.2 Selector Coloring Compression

This is a compression method based on graph coloring. Two rows of the dispatch table can be merged if no column contains different method address for the two classes. (An unimplemented method corresponds to an empty entry in the table.) The graph is constructed with one node per class; and an edge connects two nodes if the corresponding classes provide different implementations for the same method name. A heuristic algorithm may then be used to assign colors to the nodes so that no two adjacent nodes have the same color, and the minimal number of distinct colors is used. (Heuristics must be used in practice because graph coloring is a NP-complete problem.) Each color corresponds to the index for a row in the compressed table.

Note: A second graph coloring pass may be used to combine columns of the table.

Implementation of the method invocation code does not need to change from that given earlier. Each object contains a reference to a possibly shared row of the dispatch table. However, if two classes `C1` and `C2` share the same row and `C1` implements method `m` whereas `C2` does not, then the code for `m` should begin with a check that the control was reached via dispatching on an object of type `C1`. This extra check is the performance penalty for using table compression.

For the ObjectWorks example, the size of the dispatch table would be reduced from 16 to 1.15-MB. Of course, an increase occurs in code size to implement the checks that verify the correct class of the object. That increase is estimated as close to 400 KB for the ObjectWorks library.

6.2.4.3 Row Displacement Compression

The idea is to combine all the rows of the dispatch table into a single very large vector. If the rows are simply placed one after the other, this is exactly equivalent to using the two-dimensional table. However, it is possible to have two or more rows overlapping in memory as long as an entry in one row corresponds to empty entries in the other rows.

A simply greedy algorithm works well when constructing the vector. The first row is placed at the beginning of the vector; then the second row is aligned on top of the first row and tested to see if a conflicting entry exists. If there is a conflict, the row is shifted one position to the right and the test is repeated, and so on until a nonconflicting alignment is found.

Implementation of the method invocation code is again unchanged. As before, a test to verify the class of the current object must be placed at the beginning of any method that can be accessed via more than one row of the dispatch table. For the ObjectWorks example, the size of the dispatch table would be reduced from 16 MB to 819 KB, with the same 400 KB penalty for implementing checks in methods to verify the class.

6.2.4.4 Partitioning

It is pointed out in [28] that good compression combined with fast access code can be achieved by breaking dispatch tables into pieces. If two classes have the same implementations for 50 methods, say, and different implementations for just 5 methods, then we could have a single shared dispatch table for the common methods plus two separate but small tables for the 5 methods where implementations differ. The partitioning approach generalizes this insight by allowing any number of partitions to be created.

For each method, the compiler must predetermine its partition number within the owning class and its offset within a partition table. The method lookup code requires indexing the class top-level table with the method partition number to obtain a pointer to the partition table, and then indexing that partition table with the method offset.

To keep the access code as efficient as possible, the partitioning should be regular. That is, each class must have the same number of partitions, and all partitions accessed via the same offset in each class table must have the same size.

The partitioning approach advocated by Vitek and Horspool proceeds in three steps [28]. First, the compiler divides the method selectors into two sets: one set contains the conflict selectors that are selectors that are implemented by two classes unrelated by inheritance, and the other set contains all other method selectors. Two separate dispatch tables are created for the two sets of methods.

Second, two columns in a table may be combined if no two classes provide different implementations for the two methods. Merging columns may be performed using graph coloring heuristics to achieve the best results.

Third, and finally, the two tables are divided into equal sized partitions and any two partitions that are discovered to have identical contents are shared. It is possible to use two different partition sizes for splitting the two tables. Although a clever reordering of the columns might increase the opportunities for partition table sharing, good results are achieved without that extra work.

Vitek and Horspool report that a partition size of 14 entries gave good results with the ObjectWorks library [28]. The total size of all the tables came to 221 KB for the library, with a penalty for increased code size of less than 300 KB.

6.2.5 Java Class Layout and Method Invocation

Java and the programming language Theta do not implement multiple inheritance, but single inheritance with multiple subtyping. This important difference makes object layout and method dispatch more efficient. Although the bidirectional layout was designed for a language with multiple subtyping, it has the problem that more than one *vtbl* pointer has to be included in objects. The CACAO JIT compiler [18] moves the dispatch header of the bidirectional layout into the class information with negative offsets from the *vtbl*. For each implemented interface a distinct interface *vtbl* exists. Unimplemented interfaces are represented by null pointers. An example of the layout used by CACAO is shown in Figure 6.7.

To call a virtual method, two memory access instructions are necessary (load the class pointer, load the method pointer) followed by the call instruction. Calling an interface method needs an additional indirection.

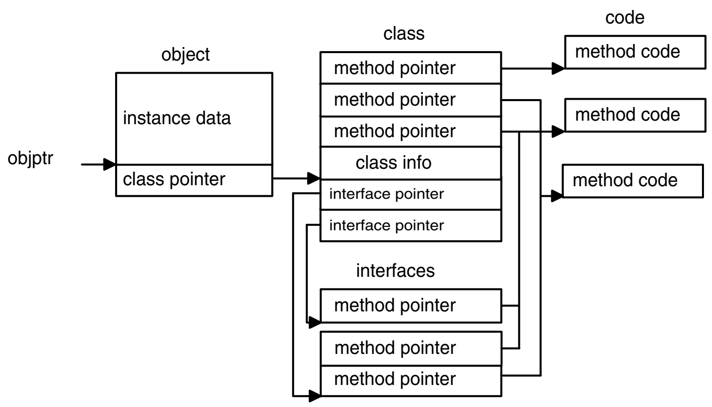


FIGURE 6.7 CACAO object and compact class descriptor layout.

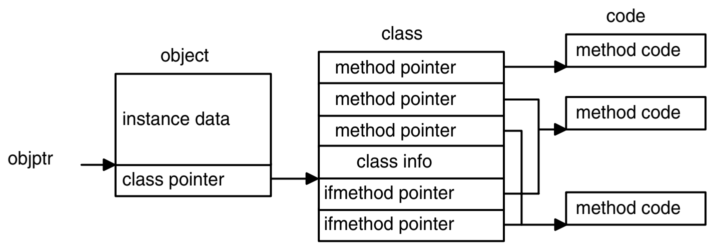


FIGURE 6.8 CACAO object and fast class descriptor layout.

In the faster scheme, we store interface methods in an additional table at negative offsets from the *vtbl*, as shown in Figure 6.8. Segregating the interface virtual function table keeps the standard virtual function table small and allows interface methods to be called with just two memory accesses. The memory consumption of virtual function tables containing interface and class methods would be *number of (classes + interfaces)* × *number of distinct methods*. The memory consumption of the interface tables is only *number of classes that implement interfaces* × *number of interface methods*. Coloring can be used to reduce the number of distinct offsets for interface methods further, but complicates dynamic class loading, leading to renumbering and code patching.

The Jalapeno virtual machine (VM) implements an interface method invocation similar to the fast class descriptor layout of CACAO; however, instead of coloring, hashing of the method indices is used [1]. The table for the interface method pointers is allocated with a fixed size much smaller than the number of interface methods. When two method indices hash to the same offset, a conflict resolving stub is called instead of the interface methods directly. For conflict resolution the stub is passed to the method index in a scratch register as additional argument. An interface method invocation can be executed with the following four machine instructions:

```
LD  vtblptr,(obj)                ; load vtbl pointer
LD  mptr,hash(method_ptr)(vtblptr); load method pointer
MV  mindex,idreg                 ; load method index
JSR (mptr)                       ; call method (or conflict stub)
```

The number of machine instructions is the same as in the compact class descriptor layout of CACAO, but the indirection is eliminated, which reduces the number of cycles needed for the execution of this instruction sequence on a pipelined architecture. Compared with the old interface method invocation in the Jalapeno VM, which searched the superclass hierarchy for a matching method signature, the new method yields runtimes that range from a 1% reduction in speed to a 51% speedup.

6.2.6 Dispatch without Virtual Method Tables

Virtual method tables are the most common way to implement dynamic dispatch. Despite the use of branch target caches and similar techniques, indirect branches are expensive on modern architectures. The SmallEiffel compiler [30] uses a technique similar to polymorphic in-line caches [15]. The indirect branch is replaced by a binary search for the type of the receiver and a direct branch to the method or the inlined code of the method.

Usually an object contains a pointer to the class information and the virtual method table. SmallEiffel replaces the pointer by an integer representing the type of the object. This type identifier is used in a dispatch function that searches for the type of the receiver. SmallEiffel uses a binary search, but a linear search weighted by the frequency of the receiver type would be possible also. The dispatch functions are shared between calls with the same statically determined set of concrete types. Assuming that the type identifiers T_A, T_B, T_C and T_D are sorted by increasing number, the dispatch code for calling `x.f` is:

```
if $id_x \leq T_B$ then
    if $id_x \leq T_A$ then $f_A(x)$
    else $f_B(x)$
else if $id_x \leq T_C$ then $f_C(x)$
    else $f_D(x)$
```

Obviously the method calls are inlined when the code is reasonably small. An empirical evaluation showed that for a method invocation with three concrete types, dispatching with binary search is between 10 and 48% faster than dispatching with a virtual method table. For a megamorphic call with 50 concrete types, the performance of the two dispatch techniques is about the same. Dispatch without virtual method calls cannot be used easily in a language with dynamic class loading (e.g., Java). Either the code has to be patched at runtime or some escape mechanism is necessary.

6.3 Fast-Type Inclusion Tests

In a statically typed language, an assignment may require a runtime test to verify correctness. For example, if class `B` is a subclass of `A`, then the assignment to `b` in the following Java code:

```
A a = new B();
...    // intervening code omitted
B b = a;
```

needs validation to ensure that `a` holds a value of type `B` (or some subclass of `B`) instead of type `A`. Usually that validation can be a runtime test.

Java also has an explicit `instanceof` test to check whether an object has the same type or is a subtype of a given type. Other object-oriented languages have similar tests. Static analysis is not very effective in eliminating these tests [14]. Therefore, efficient runtime type checking is very important.

The obvious implementation of a type inclusion test is for a representation of the class hierarchy graph to be held in memory and that graph to be traversed, searching to see whether one node is an ancestor of the other node. The traversal is straightforward to implement for a language with single inheritance, and less so for multiple inheritance. However, the defect with this approach is that the execution time of the test increases with the depth of the class hierarchy. Small improvements are possible if one or two supertypes are cached [25].

6.3.1 Binary Matrix Implementation

Given two types, c_1 and c_2, it is straightforward to precompute a binary matrix that is indexed by numbers associated with the two types `BM[`c_1`,`c_2`]` to determine whether one type is a subtype of the other.

Accessing an entry in the binary matrix requires only a few machine instructions, but the matrix can be inconveniently large, perhaps hundreds of kilobytes in size. Compaction of the binary matrix is possible, but that makes the access code more complicated.

6.3.2 Cohen's Algorithm

Cohen [10] adapted the notion of a display table (used for finding frames in block structured programming languages). Cohen's idea applies to languages with single inheritance, so that the class hierarchy graph is a tree.

Each type has a unique type identifier, *tid*, which is simply a number. A runtime data structure records the complete path of each type to the root of the class as a vector of type identifiers. The *tid* in say position three of that vector would identify the ancestor at level three in the class hierarchy.

If the compiler has to implement the test:

```
if (obj instanceof C) ...
```

then the level and type identifier for class `C` are both constants, C_level and C_tid, determined by the compiler. The steps needed to implement the test are simply:

```
level := obj.level;
if level < C_level then
    result := false
else
    result := (obj.display[C_level] = C_tid)
```

Cohen's algorithm is easy to implement, requires constant time for lookups and uses little storage. However, it works only for single inheritance hierarchies. Extending the algorithm to work with multiple inheritance hierarchies is not trivial.

6.3.3 Relative Numbering

There is a well-known algorithm for testing whether one node is an ancestor of another in a tree that works by associating a pair of integers with each node. An example tree and the numbering is shown in Figure 6.9.

To test whether node $n1$ is a descendant of $n2$ (or equal), the test is implemented as:

```
n1.left ≥ n2.left and n1.right ≤ n2.right
```

where the two numbers associated with a node are named *left* and *right*.

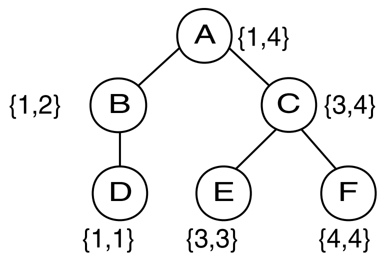


FIGURE 6.9 Relative Numbering for a Hierarchy.

Although the scheme is simple and efficient, no obvious way is available to extend it to multiple inheritance hierarchies.

6.3.4 Hierarchical Encoding

With hierarchical encoding, a bit vector is associated with each type. Each bit vector implements a compact set of integers. The test of whether type `t1` is a subclass of `t2` is implemented as the subset test `t2.vector` $\subseteq$ `t1.vector`. The approach works with multiple inheritance hierarchies.

A simple way to implement the bit vectors is to number all the nodes in the class hierarchy graph. The bit vector for node n represents the set of all the integers associated with n and the ancestors of n. That yields correct but unnecessarily large vectors.

Caseau [8], Aït-Kaci [4] and Krall et al. [19] all provide algorithms for constructing much smaller bit vectors. The algorithms are, however, computationally expensive and would need to be reexecuted after even a small change to the class hierarchy.

6.3.5 Further Algorithms

Vitek et al. [29] describe three more type test algorithms that they call *packed encoding*, *bit-packed encoding* and *compact encoding*. All three perform worse than hierarchical encoding if the total size of the data tables is used as the only criterion. However, these algorithms are much faster and are therefore more suitable for an environment where the class hierarchy may be dynamically updated, as with Java or Smalltalk, for example.

6.3.6 Partitioning the Class Hierarchy

Because type tests for trees are more (space) efficient than type tests for direct acyclic graphs (DAGs), a possible solution is to split a DAG into a tree part and the remaining graph. For languages with single inheritance and multiple subtyping, this partitioning of the class hierarchy is already done in the language itself.

CACAO uses a subtype test based on relative numbering for classes and a kind of matrix implementation for interfaces. Two numbers *low* and *high* are stored for each class in the class hierarchy. A depth-first traversal of the hierarchy increments a counter for each class and assigns the counter to the low field when the class is first encountered and assigns the counter to the high field when the traversal leaves the class. In languages with dynamic class loading a renumbering of the hierarchy is needed whenever a class is added. A class *sub* is a subtype of another class *super*, if *super.low* $\leq$ *sub.low* $\leq$ *super.high*. Because a range check is implemented more efficiently by an unsigned comparison, CACAO stores the difference between the low and high values and

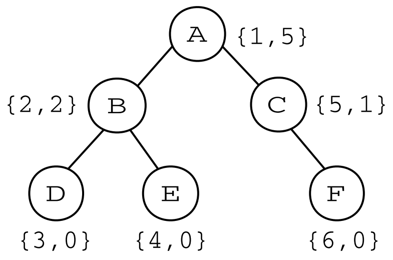


FIGURE 6.10 Elative numbering with baseval and diffval pairs.

compares it against the difference of the low values of both classes. The code for `instanceof` looks similar to:

```
return (unsigned) (sub->vftbl->baseval - super->vftbl->baseval)
   <= (unsigned) (super->vftbl->diffval);
```

Figure 6.10 shows an example hierarchy using baseval and diffval pairs. For leaf nodes in the class hierarchy the `diffval` is 0, which results in a faster test (a simple comparison of the `baseval` fields of the sub- and superclass). In general, a just-in-time (JIT) compiler can generate the faster test only for final classes. An AOT compiler or a JIT compiler that does patching of the already generated machine code may additionally replace both the `baseval` and the `diffval` of the superclass by a constant. Currently, CACAO uses constants only when dynamic class loading is not used.

CACAO stores an interface table at negative offsets from the virtual method table (as seen in Figure 6.7). This table is needed for the invocation of interface methods. The same table is additionally used by the subtype test for interfaces. If the table is empty for the index of the superclass, the subtype test fails. The code for `instanceof` looks similar to:

```
return (sub->vftbl->interfacetable[-super->index] != NULL);
```

Both subtype tests can be implemented by just a few machine code instructions without using branches that are expensive on modern processors.

6.4 Devirtualization

Devirtualization is a technique to reduce the overhead of virtual method invocation in object-oriented languages. The aim of this technique is to statically determine which methods can be invoked by virtual method calls. If exactly one method is resolved for a method call, the method can be inlined or the virtual method call can be replaced by a static method call. The analyses necessary for devirtualization also improve the accuracy of the call graph and the accuracy of subsequent inter-procedural analyses. We first discuss different type analysis algorithms, comparing their precision and complexity. Then, different solutions for devirtualization of extensible class hierarchies and similar problems are presented.

6.4.1 Class Hierarchy Analysis

The simplest devirtualization technique is class hierarchy analysis (CHA), which determines the class hierarchy used in a program. A Java class file contains information about all referenced classes. This information can be used to create a conservative approximation of the class hierarchy. The

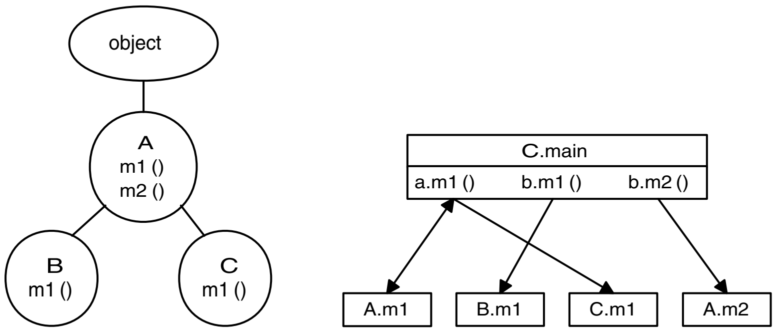


FIGURE 6.11 Class hierarchy and call graph.

approximation is formed by computing the transitive closure of all classes referenced by the class containing the main method. A more accurate hierarchy can be constructed by computing the call graph [12]. CHA uses the declared types for the receiver of a virtual method call for determining all possible receivers.

As an example, Figure 6.11 shows the class hierarchy and call graph that corresponds to the following fragment of Java code:

```
class A extends Object {
    void m1() {...}
    void m2() {...}
    }

class B extends A {
    void m1() {...}
    }

class C extends A {
    void m1() {...}
    public static void main(...) {
        A a = new A();
        B b = new B();

        a.m1(); b.m1(); b.m2();
        }
    }
```

Informally, CHA collects all methods in the call graph in a work list of methods. This work list is initialized with the *main* method. Every method is added to this work list that is inherited by a subtype of the declared type of a virtual method call in the body of each method in the work list. The algorithm given in Figure 6.12 gives the computation of the class hierarchy and call graph in more detail.

6.4.2 Rapid Type Analysis

A more accurate class hierarchy and call graph can be computed if the type of the receiver can be determined more precisely than the declared type specifies. Rapid type analysis (RTA) uses the fact that a method *m* of a class *c* can be invoked only if an object of type *c* is created during the execution

```
$main$            // the main method in a program
$x()$             // call of static method $x$
$type(x)$         // the declared type of the expression $x$
$x.y()$           // call of virtual method $y$ in expression $x$
$subtype(x)$      // $x$ and all classes which are a subtype of class $x$
$method(x, y)$    // the method $y$ which is defined for class $x$

$callgraph := main$
$hierarchy := \{\}$
for each $m \in callgraph$ do
    for each $m_{stat}()$ occurring in $m$ do
        if $m_{stat} \notin callgraph$ then
            add $m_{stat}$ to $callgraph$
    for each $e.m_{vir}()$ occurring in $m$ do
        for each $c \in subtype(type(e))$ do
        $m_{def} := method(c; m_{vir})$
        if $m_{def} \notin callgraph$ then
            add $m_{def}$ to $callgraph$
            add $c$ to $hierarchy$
```

FIGURE 6.12 Class hierarchy analysis.

of a program [5, 7]. RTA refines the class hierarchy by only including classes for which objects can be created at runtime. The pessimistic algorithm includes all classes in the class hierarchy for which instantiations occur in methods of the call graph from CHA.

The optimistic algorithm initially assumes that no methods besides `main` are called and that no objects are instantiated. It traverses the call graph initially ignoring virtual calls (only marking them in a class mapping as a potential call) following static calls only. When the instantiation of an object is found during the analysis, all virtual methods of the corresponding class that were left out previously are then traversed as well. The live part of the call graph and the set of instantiated classes grow interleaved as the algorithm proceeds.

Figure 6.13 shows the rapid type analysis algorithm.

6.4.3 Other Fast Precise-Type Analysis Algorithms

Tip and Palsberg [27] evaluated different algorithms that are more precise but are on average only a factor of five slower than RTA. The different type analysis algorithms differ primarily in the number of sets of types used. RTA uses one set for the whole program. 0-Control flow analysis (CFA) [13, 23] uses one set per expression. *k*-*l*-CFA makes separate analyses for *k* levels of method invocations and uses more than one set per expression. These algorithms have high computational complexity and only work for small programs. Therefore, Tip and Palsberg evaluated the design space between RTA and 0-CFA [27].

XTA uses a distinct set for each field and method of a class, fast type analysis (FTA) uses one set for all fields and a distinct set for every method of a class and MTA uses one set for all methods and a distinct set for every field of a class. Arrays are modeled as classes with one instance field. All algorithms use iterative propagation-based flow analysis to compute the results. Three work lists are associated with each method or field that keeps track of processed types. *New* types are propagated to the method or field in the current iteration and can be propagated onward in the next iteration. Current types can be propagated onward in the current iteration. Processed types have been propagated onward in previous iterations.

Tip and Palsberg efficiently implemented the type sets using a combination of array-based and hash-based data structures to allow efficient membership tests, element additions and iterations

main	// the main method in a program
new x	// instantiation of an object of class x
marked(x)	// marked methods of class x
$x()$	// call of static method x
$type(x)$	// the declared type of the expression x
$x.y()$	// call of virtual method y in expression x
subtype(x)	// x and all classes which are a subtype of class x
method(x, y)	// the method y which is defined for class x
mark(m, x)	// mark method m in class x

```
callgraph := main
hierarchy := {}
for each $m \in callgraph$ do
    for each new $c$ occurring in $m$ do
        if $c \notin hierarchy$ then
            add $c$ to hierarchy
            for each $m_{mark} \in$ marked($c$) do
                add $m_{mark}$ to callgraph
    for each $m_{stat}()$ occurring in $m$ do
        if $m_{stat} \notin callgraph$ then
            add $m_{stat}$ to callgraph
    for each $e.m_{vir}()$ occurring in $body(m)$ do
        for each $c \in subtype(type(e))$ do
        $m_{def} := method(c, m_{vir})$
        if $m_{def} \notin callgraph$ then
            if $c \notin hierarchy$ then
                $mark(m_{def,c})$
            else
                add $m_{def}$ to callgraph
```

FIGURE 6.13 Rapid type analysis.

over all elements [27]. Type inclusion is implemented by relative numbering. These techniques are necessary because the type sets are filtered by the types of fields, method parameters and method return types. Additionally, type casts restricting return types are used for filtering.

All these algorithms are more precise than RTA. On the range of Java benchmark programs (benchmark code only), MTA computes call graphs with 0.6% fewer methods and 1.6% fewer edges than RTA. FTA computes call graphs with 1.4% fewer methods and 6.6% fewer edges than RTA. XTA computes call graphs with 1.6% fewer methods and 7.2% fewer edges than RTA. All algorithms are about five times as slow as RTA. Therefore, XTA has the best precision and performance trade-off of the three algorithms compared.

6.4.4 Variable Type Analysis

RTA is imprecise because every type that is instantiated somewhere in the program and is a subtype of the declared type can potentially be the receiver of a method invocation. Variable type analysis (VTA) is more precise because it computes reaching type information, taking into consideration chains of assignments between instantiations and method invocations [24], but it does ignore type casts. It is a flow-insensitive algorithm that avoids iterations over the program. The analysis works by constructing a directed type propagation graph where nodes represent variables and edges represent assignments. Reaching type information is initialized by object instantiations and propagated along the edges of the type propagation graph.

The type propagation graph is constructed from the classes and methods contained in the conservative call graph. For every class *c* and for every field *f* of *c* that has a reference type a node *c.f* is created. Additionally for every method *c.m* and:

- For every formal parameter *p* (including *this*) of *c.m* that has a reference type, create a node *c.m.p*.
- For every local variable *l* of *c.m* that has a reference type, create a node *c.m.l*.
- If *c.m* returns a reference type, create a node *c.m.ret*.

After the nodes are created, then for every assignment of reference types an edge is added to the graph. Assignments are either explicit assignments of local variables or object fields or assignments resulting from passing of parameters or returning a reference value. Native methods are handled by summarizing their effects on the analysis.

To avoid alias analysis, all variable references and all their aliases are represented by exactly one node. In Java, locals and parameters cannot be aliased. All instances of a field of a class are represented by one node. Arrays could introduce aliasing. Different variables could point to the same array. Therefore, if both sides of an assignment are of type `Object` or if at least one side is an array type, edges in both directions are added.

Type propagation is accomplished in two phases. The first phase detects strongly connected components in the type propagation graph. All nodes of a strongly connected component are collapsed into one supernode. The type of this supernode is the union of the types of all its subnodes. The remaining graph is a DAG. Types are propagated in a topological order where a node is processed after all its predecessors have been processed. The complexity of both strongly connected component detection and type propagation are linear in the maximum of the number of edges and nodes. The most expensive operation is the union of type sets.

The algorithm does no killing of types on casts or declared types. An algorithm using declared type information would be more precise, but collapsing of strongly connected components would not be possible anymore. Impossible types are filtered after type propagation has been finished.

Over the set of Java benchmarks (the benchmark code only), VTA computes call graphs with 0.1 to 6.6% fewer methods and 1.1 to 18% fewer edges than RTA. For the set of Java applications (including the libraries), VTA computes call graphs with 2.1 to 20% fewer methods and 7.7 to 27% fewer edges than RTA. The implementation is untuned and written in Java. The performance numbers indicate that the algorithm scales linearly with the size of the program (54 sec for 27,570 instructions, 102 sec for 55,468 instructions).

6.4.5 Cartesian Product Algorithm

Agesen [2] developed a type analysis algorithm called the Cartesian product algorithm (CPA) where a method is analyzed separately for every combination of argument types (Cartesian product of the argument types). For example, a method with two arguments, where the first argument can be of type `Int` and `Long` and the second argument can be of type `Float` and `Double`, can result in four different analyses with argument types (`Int`, `Float`), (`Long`, `Float`), (`Int`, `Double`) and (`Long`, `Double`). For better precision, CPA computes the needed analyses lazily. Only argument type combinations are analyzed that occur during program analysis. The return type of a method is the union of the return types of the different analyses for a specific method invocation.

CPA is precise in the sense that it can analyze arbitrary deep call chains without loss of precision. CPA is efficient, because redundant analysis is avoided. However, megamorphic call sites, where the method has many arguments and an argument has a high number of different types, can lead to long analysis times. Therefore, Agesen restricted the number of different types for an argument and combined the analyses if the number exceeded a small constant. The bounded CPA scales well for bigger programs too.

6.4.6 Comparisons and Related Work

Grove et al. investigated the precision and complexity of a set of type analysis algorithms [13]. They implemented a framework where different algorithms can be evaluated. They evaluated RTA, the bounded CPA and different levels of *k*-*l*-CFA. CPA gives more accuracy than 0-CFA with reasonable computation times. The higher levels of *k*-*l*-CFA cannot be used for bigger applications.

CHA and RTA have the same complexity, but RTA always produces more accurate results. The results for XTA and VTA cannot be compared directly, because different programs are used for benchmarking and implementation of VTA was not done for performance. It can be estimated that the runtime for the algorithms can be similar, but VTA can produce more accurate results. 0-CFA is more accurate than the other algorithms at slightly higher analysis costs.

6.4.7 Inlining and Devirtualization Techniques

Ishizaki et al. [16] point out that straightforward devirtualization may have little effect on the performance of Java programs. Because Java is strongly typed, a *vtbl* can be used for dispatching. Devirtualizing simply removes the lookup in the *vtbl*, and that is not significant compared with the other costs of calling a method. Significant performance gains only arise if the devirtualized method is inlined at the call site; many opportunities for devirtualization are lost in any case because Java has dynamic class loading.

Ishizaki et al. [16] propose a technique based on code patching that allows methods to be inlined and inlining to be removed if dynamic class loading subsequently requires the method call to be implemented by the normal *vtbl* dispatching again. Their code patching technique avoids any need to recompile the code of the caller.

An example can make the idea clear. Suppose that the program to be compiled contains the following Java statements:

```
i = i + 1;
obj.meth(i,j);
j = j - 1;
```

The compiler would normally generate the following pattern of code for those statements:

```
// code for i = i + 1
// code to load arguments i and j
// dispatch code to lookup address of method meth
// ... and pass obj and arguments to that method
// code for j = j - 1
```

Now suppose that analysis of the program shows that only one possible implementation of method *meth* can be invoked at this call site. If the body of that method is reasonably small, it can be inlined. The generated code corresponds to the following pattern:

```
        // code for i = i + 1
        // inlined code for method meth parameterized
        // ... by the arguments obj, i, and j
L2:     // code for j = j - 1
```

```
...   // much omitted code

L1:   // code to load arguments i and j
      // dispatch code to lookup address of method meth
      // ... and pass arguments and obj to that method
      goto L2;
```

Label L1 is not reached with this version of the code.

Now suppose that dynamic class loading causes an alternative implementation of method *meth* to be loaded. The runtime environment now uninlines the method call by patching the code. It overwrites the first word of the inlined method with a branch instruction, so that the patched code corresponds to the following pattern:

```
      // code for i = i + 1
      goto L1
      // remainder of code of inlined method, which
      // is now unreachable.
L2:   // code for j = j - 1

...   // much omitted code

L1:   // code to load arguments i and j
      // dispatch code to lookup address of method meth
      // ... and pass arguments and obj to that method
      goto L2;
```

This patched code contains two more branches than the original unoptimized program and would therefore run more slowly; it also contains unreachable code that incurs a modest space penalty. The assumption is that dynamic class loading is rare and that methods rarely need to be uninlined.

Experiments with a JIT compiler showed that the number of dynamic method calls is reduced by 9 to 97% on their test suite. The effect on execution speed ranged from a small 1% worsening of performance to an impressive 133% improvement in performance, with a geometric mean speedup of 16%.

Ishizaki et al. [16] also point out that a similar technique is applicable to a method invoked via an interface. If only one class implements an interface class, we can generate code that assumes this class is actually used and we can inline methods of that class that are not overridden by any of its subclasses. If the assumption is later broken by dynamically loading a new class that also implements the interface or that overrides the method, we can patch the code to revert to the original full scheme of looking up the class and looking up the method.

6.5 Escape Analysis

In general, instances of classes are dynamically allocated. Storage for these instances is normally allocated on the heap. In a language such as C++ where the programmer is responsible for allocating and deallocating memory for objects on the heap, the program should free the memory for a class instance when it is no longer needed. Other languages, such as Java, provide automatic garbage collection. At periodic intervals, the garbage collector is invoked to perform the computationally intensive task of tracing through the references between objects and determining which objects can no longer be referenced. The storage for these objects is reclaimed.

The goal of escape analysis is to determine which objects have lifetimes that do not stretch outside the lifetime of their immediately enclosing scopes. The storage for such objects can be safely allocated as part of the current stack frame; that is, their storage can be allocated on the runtime stack. (For C programmers who use the `gcc` C compiler, the transformation is equivalent to replacing a use of the `malloc` function with the `alloca` function.) This optimization is valuable for Java programs. The transformation also improves the data locality of the program and, depending on the computer cache, can significantly reduce execution time.

Another benefit of escape analysis is that objects with lifetimes that are confined to within a single scope cannot be shared between two threads. Therefore, any synchronization actions for these objects can be eliminated. Escape analysis does not capture all possibilities for synchronization removal, however. If this is deemed to be an important optimization, then a separate analysis to uncover unnecessary synchronization operations should be performed.

Algorithms for escape analysis are based on abstract interpretation techniques [3]. There are different algorithms that make different trade-offs between the precision of the analysis and the length of time the analysis takes. The better the precision, the more opportunities for optimization that should be found.

The reported speedup of Java programs can range up to 44% [6], but that last figure includes savings due to synchronization removal and due to inlining of small methods. (Blanchet [6] reports an average speedup of 21% in his experiments.) Inlining significantly increases the number of opportunities for finding objects that do not escape from their enclosing scope, especially because many methods allocate a new object that is returned as the result of the method call.

6.5.1 Escape Analysis by Abstract Interpretation

A prototype implementation of escape analysis was included in the IBM High Performance Compiler for Java. This implementation is based on straightforward abstract interpretation techniques and has been selected for presentation in this text because it is relatively easy to understand. Further details of the algorithm may be found in the paper published by Choi et al. [9].

The approach of Choi et al. [9] attempts to determine two properties for each allocated object — whether the object escapes from a method (i.e., from the scope where it is allocated), and whether the object escapes from the thread that created it. It is possible that an object escapes the method but does not escape from the thread, and thus synchronization code may be removed. The converse is not possible; if an object does not escape a method, then it cannot escape its thread. The analysis therefore uses the very simple lattice of three values shown in Figure 6.14. If analysis determines that an object status is *NoEscape*, then the object definitely does not escape from its method or from its thread; if the status is *ArgScape*, the object may escape from a method via its arguments but definitely does not escape the thread; finally, *GlobalEscape* means that the object may escape from both the method and the thread.

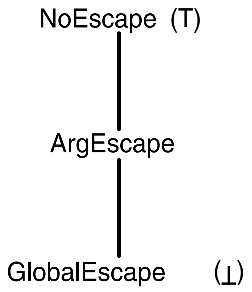


FIGURE 6.14 Lattice elements for escape analysis.

The two versions of the analysis are a fast flow-insensitive version that yields imprecise results, and a slower flow-sensitive version that gives better results. *Imprecise* means that the analysis can be overly conservative, reporting many objects as having *GlobalEscape* status when a more accurate analysis might have shown the status as one of the other two possibilities, or reporting *ArgEscape* instead of *NoEscape*. Imprecision in this manner does not cause incorrect optimizations to be made; some opportunities for optimization can simply be missed. We give only the more precise flow-sensitive version of the analysis in this chapter.

6.5.1.1 Connection Graphs

As its name suggests, abstract interpretation involves interpretive execution of the program. With this form of execution, the contents of variables (or fields of classes when analyzing a Java program) are tracked. However, we do not attempt to determine the contents of the variables for normal execution of the program — we would of course simply execute the program to do that. To perform escape analysis, we are interested only in following an object `O` from its point of allocation, knowing which variables reference `O` and which other objects are referenced by `O` fields. The abstraction implied in the name *abstract interpretation* is to abstract out just the referencing information, using a graph structure where nodes represent variables and objects, and directed edges represent object references and containment of fields inside objects. Choi et al. [9] call this graph a *connection graph*.

A sample connection graph is shown in Figure 6.15; it shows the program state after executing the code:

```
A a = new A();    // line L1
a.b1 = new B();   // line L2
a.b2 = a.b1;
```

where we assume that the only fields of `AC` are `b1` and `b2`, and `BC` has only fields with intrinsic types (i.e., the types `int` and `char`).

The notational conventions used in the connection graph are as follows. A circle node represents a variable (i.e., a field of a class or a formal parameter of a method); a square node represents an object instance. An edge from a circle to a square represents a reference; an edge from a square to a circle represents ownership of fields.

The graph shown in Figure 6.15 on the left is a simplification of that used by Choi et al. [9]. Their more complicated graph structure has two kinds of edges. An edge drawn as a dotted arrow is called a *deferred edge*. When there is an assignment from copies one object reference to another, such as:

```
p = q;  // p and q have class types
```

then the effect of the assignment is shown as a deferred edge from the node for `p` to the node for `q`. In Figure 6.15, the graph on the right uses a deferred edge to show the effect of an assignment from one variable to another.

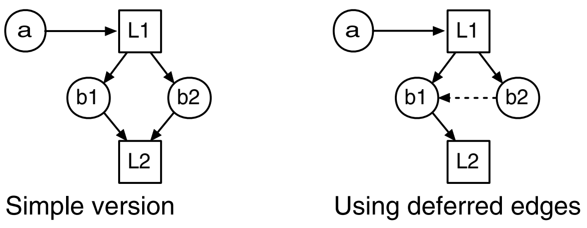


FIGURE 6.15 A sample connection graph.

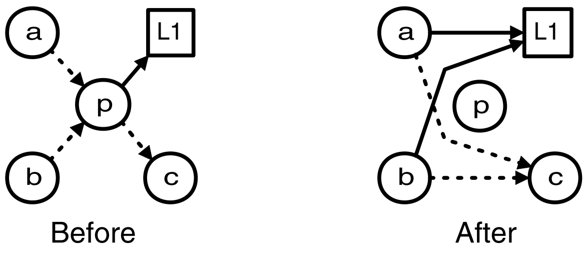


FIGURE 6.16 Effect of bypass operation.

Each node in a connection graph has an associated escape state, chosen from the three possibilities given in Figure 6.14. If a connection graph has been constructed for a method *M* and if *O* is an object node in *M*, then if *O* can be reached from any node in the graph whose escape state is other than *NoEscape* then *O* may escape from *M*. A similar property holds for objects escaping from a thread.

6.5.1.2 Intraprocedural Abstract Interpretation

The abstract interpretation is performed on a low-level version of the code where only one action at a time is performed. The Java bytecode is adequate for this purpose, as are other intermediate representation formats used in a typical compiler. Assuming that the code has been suitably simplified, abstract interpretation of the code within a method steps through the code and performs an action appropriate for each low-level operation. This interpretive execution involves following control flow edges, as explained in more detail later.

The actions for assignment statements and instantiations of new object instances are shown next. Each action involves an update to the connection graph. An assignment to a variable `p` *kills* any previous value that the variable previously had. The kill part of an assignment to `p` is implemented by an update to the connection graph that is called *ByPass(p)*. The *ByPass(p)* operation redirects or removes deferred edges as illustrated in Figure 6.16. Note also that compound operations, such as `p.a.b.c`, are assumed to be decomposed into simpler steps that dereference only one level at a time — the bytecode form of the program automatically possesses this property:

```
p = new C(); // line L
```

If the connection graph does not already contain an object node labeled `L` then one is created and added to the graph. If the node needs to be created, then nodes for the fields of `C` that have nonintrinsic types are also created and are connected by edges pointing from the object node. Any outgoing edges from the node for `p` are deleted by applying the *ByPass(p)* operation, then a new edge from `p` to the object node for `L` is added.

```
p = q;
```

The *ByPass(p)* operation is applied to the graph. Then a new deferred edge from `p` to `q` is created.

```
p.f = q;
```

If `p` does not point to any object nodes in the connection graph, then a new object node (with the appropriate fields for the datatype of `p`) is created and an edge from `p` to the new object node is added to the graph. (Choi et al. [9] call this new object node a *phantom node*. Two reasons why phantom nodes may arise are (1) the original program may contain an error and `p` would actually be null, referencing no object, when this statement is reached; and (2) `p` may reference an object outside the current method — and that situation can be covered by the interprocedural analysis.) Then, for each object node that is connected to `p` by an edge, an assignment to the f field of that object is performed. That assignment is implemented by adding a deferred edge from the f node to the q node. Note that no *ByPass* operation is performed (to kill the previous value of f) because there is not necessarily a unique object that p references, and we cannot therefore be sure that the assignment kills all previous values for f.

p = q.f;

As before, if q does not point at any object nodes, then a phantom node is created and an edge from q to the new node is added to the graph. Then, *ByPass(p)* is applied, and deferred edges are added from p to all the f nodes that q is connected to by field edges.

In principle, a different connection graph represents the state of the program at each statement in the method. Thus, when the abstract interpretation action modifies the graph, it is modifying a copy of the graph. When analyzing a sequence of statements in a basic block, the analysis proceeds sequentially through the statements in order. At a point where control flow diverges, such as at a conditional statement, each successor statement of the conditional is analyzed using a separate copy of the connection graph. At a point where two or more control paths converge, the connections graphs from each predecessor statements are merged.

A small example is given to make the process clearer. Suppose that the code inside some method is as follows, with the declarations of classes A, B1 and B2 omitted:

```
A a = new A();       // line L1
if (i > 0)
   a.f1 = new B1(); // line L3
else
   a.f1 = new B2(); // Line L5
a.f2 = a.f1;        // Line L6
```

The connection graphs that are constructed by the abstract interpretation actions are shown in Figure 6.17. Diagram 1 in the figure shows the state after executing line L1; diagrams 2 and 3 show the states after lines L3 and L5, respectively. Note that diagrams 2 and 3 are obtained by applying the effects of lines L3 and L5 to the state in diagram 1. After the if statement, the two control flow paths merge; the graph in diagram 4 is the result of merging diagrams 2 and 3. Finally, diagram 5 shows the effect of applying line L6.

If the program contains a loop, abstract interpretation is performed repeatedly until the connection graphs converge. Convergence is guaranteed because the maximum number of nodes in the graph is a finite number that is proportional to the number of occurrences of new in the source code, and the number of edges that can be added between the nodes is also finite. If, for example, the source code to be analyzed is:

```
Node head = null;
for( int cnt = 0;  cnt < 1000;  cnt++ ) {
   Node n = new Node();  // Line L3
   n.next = head;
   n.data = cnt;
   head = n;
}
```

then analysis gives the connection graphs shown in Figure 6.18. After analyzing the loop body once, the graph has the structure shown on the left; after analyzing a second time, the graph has converged to the diagram on the right. Even though the actual program allocates 1000 instances of the Node

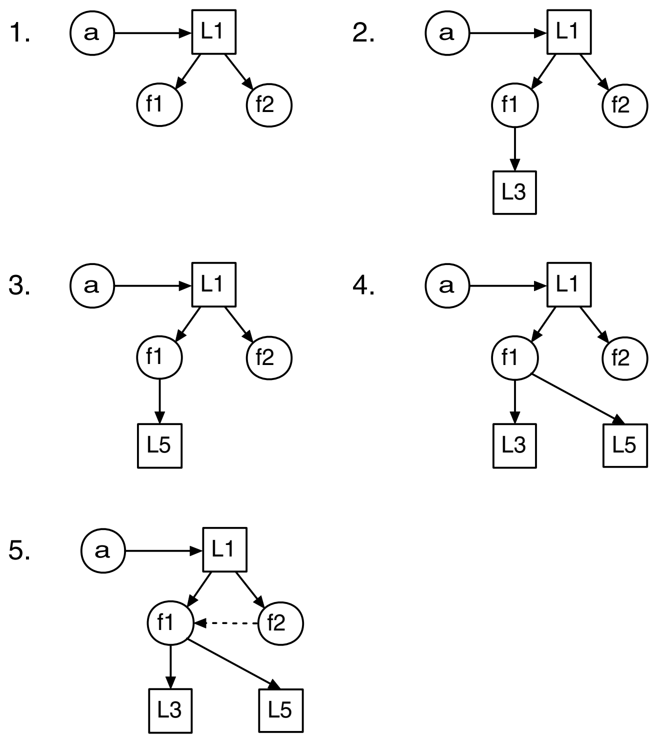


FIGURE 6.17 Sequence of connection graphs.

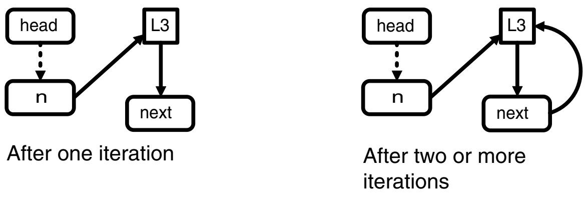


FIGURE 6.18 Connection graph for a loop.

class, only one `new` operation is in the code and therefore only one object node is in the graph. The fact that one graph node represents 1000 objects in the program is one of the approximations inherent in the graph structure invented by Choi et al. [9].

6.5.1.3 Interprocedural Abstract Interpretation

A call to a method *M* is equivalent to copying the actual parameters (i.e., the arguments are passed in the method call) to the formal parameters, then executing the body of *M*, and finally copying any value returned by *M* as its result back to the caller. If *M* has already been analyzed intraprocedurally following the approach described earlier, the effect of *M* can be summarized with a connection graph. That summary information eliminates the need to reanalyze *M* for each call site in the program.

It is necessary to analyze each method in the reverse of the order implied by the *call graph*. If method A may call methods B and C, then B and C should be analyzed before A. Recursive edges in the call graph are ignored when determining the order. Java has virtual method calls — at a method call site where it is not known which method implementation is invoked, the analysis must assume that all the possible implementations are called, combining the effects from all the possibilities. The interprocedural analysis iterates over all the methods in the call graph until the results converge.

The extra actions needed to create the summary information for a method *M* follow:

Entry to a method M

If M has $n-1$ formal parameters, $\mathtt{f_i}, \ldots \mathtt{f_n}$, then n object nodes $\mathtt{a_1} \ldots \mathtt{a_n}$ are created. These correspond to the actual parameters (or arguments). The extra parameter corresponds to the implicit `this` parameter. Because Java has call-by-value semantics, an implicit assignment exists from each actual parameter to the corresponding formal parameter at method invocation. These assignments are modeled by creating a deferred edge from $\mathtt{f_i}$ to $\mathtt{a_i}$ for each parameter. The escape state initially associated with a $\mathtt{f_i}$ node is *NoEscape* and the state initially associated with a $\mathtt{a_i}$ node is *ArgEscape*. By having created the first object nodes for the parameters, the body of the method is analyzed using the approach described for intraprocedural analysis.

Exit from method M

A `return` statement is processed by creating a dummy variable named *return* to which the returned result is assigned. If there are multiple `return` statements in the method, the different result values are merged into the connection graph by adding deferred edges from the *return* node to the different results.

When the whole method body has been processed, the connection graph that was created after the `return` statement represents the summary information for the method. In particular, after the *ByPass* function has been used to eliminate all deferred edges, the connection graph can be partitioned into three subgraphs:

Global escape nodes. All nodes that are reachable from a node whose associated state is *GlobalEscape* are themselves nodes that are considered to be global escape nodes and form the first subgraph. The nodes initially marked as *GlobalEscape* are the static fields of any classes and instances of any class that implements the `Runnable` interface.

Argument escape nodes. All nodes reachable from a mode whose associated state is *ArgEscape*, but are not reachable from a *Global Escape* node, are in the second subgraph. The nodes initially marked as *ArgEscape* are the argument nodes $\mathtt{a_1} \ldots \mathtt{a_n}$.

No escape nodes. All other nodes in the connection graph form the third subgraph and have *NoEscape* status.

All objects created within a method *M* and that have the *NoEscape* status after the three subgraphs are determined can be safely allocated on the stack. The third subgraph represents the summary information for the method because it shows which objects can be reached via the arguments passed to the method. The remaining part that remains to be covered is how to process a method call when it is encountered in the body of the method analyzed.

Suppose that while analyzing some method `m1`, we reach a method call:

```
result = obj.m2(p1,p2);
```

and that we have previously analyzed method `m2` in the class to which `obj` belongs. The analysis algorithm creates three formal parameter nodes $\hat{\mathtt{a}}_1$, $\hat{\mathtt{a}}_2$, $\hat{\mathtt{a}}_3$ and processes three assignments:

$$\hat{a}_1 = \mathtt{obj}; \quad \hat{a}_2 = \mathtt{p1}; \quad \hat{a}_3 = \mathtt{p2};$$

The nodes $\hat{\mathtt{a}}_1 \ldots$ correspond to the argument nodes $\mathtt{a_1} \ldots$ in the connection graph that summarizes method `m2`, whereas the values `obj`, `p1` and `p2` are nodes within the connection graph constructed for method `m1`.

The summary connection graph for `m2` is used like a macro. Connections from `a1`... are copied, becoming edge connections from `obj`. Field nodes that are children of object values that `ai` nodes reference are matched against field nodes that are children of the `obj`, `p1` and `p2` nodes, and so on recursively. During this process, many of the phantom nodes that were introduced into the connection graph of `m2` can be matched against nodes of `m1`. The algorithm for matching and duplicating the nodes and edges is omitted for space reasons.

One more issue needs to be covered. If the method call graph contains cycles, as occurs in the presence of direct or indirect recursion, then it appears to be necessary to use a method summary connection graph before it has been created. For this situation, a special bottom graph is used in place of the connection graph for the unanalyzed method. The bottom graph represents a worst-case (or conservative) scenario for escape of objects from any method in the program. The bottom graph has one node for every class. A points-to edge is from the node for class C1 to the node for class C2 if C1 contains a field of type C2; a deferred edge is from the node for C1 to the node for C2 if C2 is a subtype of C1. This graph can be used to make a conservative estimate for the escape of objects passed to a method by matching the argument types against the nodes in the bottom graph.

6.5.2 Other Approaches

A different analysis technique was developed by Blanchet [6]. His approach also uses abstract interpretation, but three significant differences exist.

First, the Java bytecode is directly interpreted so that an abstract representation of the values on the Java runtime stack is managed. During abstract interpretation, each bytecode operation has an effect on that representation of the stack.

Second, information is propagated both forward and backward. Forward propagation occurs when instructions are analyzed following the normal flow of control — as in the approach of Choi et al. [9]. Backward propagation is performed by interpretively executing the bytecode instructions along the reverse of control flow paths. The reverse execution mode has, of course, different semantics for the abstract meaning of each instruction (e.g., an instruction that pushes a value when interpreted forward becomes one that pops a value when analyzed backward). The combination of forward and backward analysis passes enables much more precise results to be obtained, especially when used in conjunction with the program analyzed containing no errors. (This is a common assumption for code optimization.)

Third, Blanchet [6] uses a quite different domain of values to represent escape of objects. He represents each class type by an integer, which is the context for what may escape from an instance of that class. His abstract values are equations (or context transformers) that map from the contexts of the arguments and result of a method to the escape contexts of concrete values. Instead of manipulating a collection of graphs, as Choi et al. [9], Blanchet [6] manipulates sets of equations.

No comparison has been made between the two approaches of Choi et al. and Blanchet. Blanchet states that the Choi et al. analysis is more time consuming, and that is almost certainly true. Blanchet also claims bigger speedups for his set of sample programs, but Blanchet also performs extensive inlining of small methods.

6.6 Conclusion

We presented the most important optimizations for object-oriented programming languages that should give language implementors good choices for their work. Method invocation can be efficiently solved by different kind of dispatch tables and inlining. Inlining and specialization greatly improve the performance but need precise and efficient type analysis algorithms. Escape analysis computes the informations necessary to allocate objects on the runtime stack.

References

[1] B. Alpern, A. Cocchi, D. Grove and D. Lieber, Efficient dispatch of Java interface methods, in *HPCN '01, Java in High Performance Computing*, V. Getov and G.K. Thiruvathukal, Eds., *Lecture Notes in Computer Science*, Vol. 2110, Springer-Verlag, New York, 2001, pp. 621–628.

[2] O. Agesen, The Cartesian product algorithm: simple and precise type inference of parametric polymorphism, in *ECOOP '95 — Object-Oriented Programming, 9th European Conference*, W.G. Olthoff, Ed., *Lecture Notes in Computer Science*, Vol. 952, August 7–11, 1995, Springer-Verlag, New York, 1995, pp. 2–26.

[3] S. Abramsky and C. Hankin, *Abstract Interpretation of Declarative Languages*, Ellis Horwood, New York, 1987.

[4] H. Aït-Kaci, R. Boyer, P. Lincoln and R. Nasr, Efficient implementation of lattice operations, *Trans. Programming Languages Syst.*, 11(1), 115–146, November 1989.

[5] D.F. Bacon, Fast and Effective Optimization of Statically Typed Object-Oriented Languages, Ph.D. thesis, University of California, Berkeley, 1997.

[6] B. Blanchet, Escape analysis for object oriented languages: application to java, in *Proceedings of OOPSLA '99*, ACM Press, New York, 2000, pp. 20–34.

[7] D.F. Bacon and P.F. Sweeney, Fast static analysis of C++ virtual function calls, in *Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '96)*, ACM Press, New York, 1996, pp. 324–341.

[8] Y. Caseau, Efficient handling of multiple inheritance hierarchies, in *Proceedings of OOPSLA '93*, Vol. 28(10) of *SIGPLAN*, ACM Press, New York, 1993, pp. 271–287.

[9] J.-G. Choi, M. Gupta, M. Serrano, V.C. Sreedhar and S. Midkiff, Escape analysis for java, in *Proceedings of OOPSLA '99*, Denver, November 1999, ACM Press, New York, pp. 1–19.

[10] N.J. Cohen, Type-extension type tests can be performed in constant time, *Trans. Programming Languages Syst.*, 13(4), 626–629, April 1991.

[11] P. Dencker, K. Dürre and J. Heuft, Optimization of parser tables for portable compilers, *Trans. Programming Languages Syst.*, 6(4), 546–572, April 1984.

[12] J. Dean, D. Grove and C. Chambers, Optimization of object-oriented programs using static class hierarchy analysis, in *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP '95)*, *Lecture Notes in Computer Science*, Vol. 952, Springer-Verlag, New York, 1995, pp. 77–101.

[13] D. Grove, G. DeFouw, J. Dean and C. Chambers, Call graph construction in object-oriented languages, in *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA-97)*, *ACM SIGPLAN Notices*, Vol. 32, 10 October, ACM Press, New York, 1997, pp. 108–124.

[14] S. Gehmawat, K.H. Randall and D.J. Scales, Field analysis: getting useful and low-cost interprocedural information, in *Conference on Programming Language Design and Implementation SIGPLAN*, Vol. 35(5), ACM Press, New York, 2000, pp. 334–344.

[15] U. Hölzle, C. Chambers and D. Ungar, Optimizing dynamically-typed object-oriented programming languages with polymorphic inline caches, in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '91)*, *Lecture Notes in Computer Science*, Vol. 512, Springer-Verlag, New York, 1991, pp. 21–38.

[16] K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu and T. Nakatani, A study of devirtualization techniques for a JavaTM just-in-time compiler, in *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Application (OOPSLA-00)*, *ACM SIGPLAN Notices*, Vol. 35.10, ACM Press, New York, 2000, pp. 294–310.

[17] R. Jones and R. Lins, *Garbage Collection*, John Wiley & Sons, New York, 1996.

[18] A. Krall and R. Grafl, CACAO — a 64 bit JavaVM just-in-time compiler, *Concurrency: Pract. Exp.*, 9(11), 1017–1030, 1997.

[19] A. Krall, J. Vitek and N. Horspool, Near optimal hierarchical encoding of types, in *11th European Conference on Object Oriented Programming (ECOOP '97)*, M. Aksit and S. Matsuoka, Eds., *Lecture Notes in Computer Science*, Vol. 1241, Springer-Verlag, New York, 1997, pp. 128–145.

[20] B. Liskov, D. Curtis, M. Day, S. Ghemawat, R. Gruber, P. Johnson and A.C. Myers, Theta reference manual, Technical Report Programming Methodology Group Memo 88, Laboratory for Computer Science, Massachusetts Institute of Technology, February 1995.

[21] A.C. Myers, Bidirectional object layout for separate compilation, in *OOPSLA '95 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications*, ACM Press, New York, 1995, pp. 124–139.

[22] J.R. Rose, Fast dispatch mechanisms for stock hardware, in *OOPSLA '88:* Object-Oriented Programming Systems, Languages and Applications: Conference Proceedings, N. Meyrowitz, Ed., 1988, pp. 27–35.

[23] O. Shivers, Control-Flow Analysis of Higher-Order Languages, Ph.D. thesis, Carnegie-Mellon University, Pittsburgh, PA, May 1991.

[24] V. Sundaresan, L.J. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon and C. Godin, Practical virtual method call resolution for java, in *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Application (OOPSLA-00), ACM Sigplan Notices*, Vol. 35.10, ACM Press, New York, 2000, pp. 264–280.

[25] T. Suganuma, T. Ogasawara, M.T. Yasuekeuchi, M. Kawahito, K. Ishizaki and H. Komatsuatani, Overview of the IBM Java just-in-time compiler, *IBM Syst. J.*, 39(1), 175–193, 2000.

[26] B. Stroustrup, Multiple inheritance for C++, in *Comput. Syst.*, USENIX Association, Ed., 2(4), 367–395, Fall, 1989.

[27] F. Tip and J. Palsberg, Scalable propagation-based call graph construction algorithms, in *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Application (OOPSLA-00), ACM Sigplan Notices*, Vol. 35(10), ACM Press, New York, 2000, pp. 281–293.

[28] J. Vitek and R. Nigel Horspool, Compact Dispatch Tables for Dynamically Typed Object Oriented Languages, in 6th International Conference CC '96, 1996, pp. 307–325.

[29] J. Vitek, N. Horspool and A. Krall, Efficient type inclusion tests, in *Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '97)*, Atlanta, GA, T. Bloom, Ed., ACM Press, New York, 1997, pp. 142–157.

[30] O. Zendra, D. Colnet and S. Collin, Efficient dynamic dispatch without virtual function tables: the small Eiffel compiler, in *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA-97), ACM SIGPLAN Notices*, Vol. 32.10, ACM Press, New York, 1997, pp. 125–141.

7

Data Flow Testing

Neelam Gupta
University of Arizona

Rajiv Gupta
University of Arizona

7.1 Introduction

Although the employment of systematic design and development practices results in increasingly reliable software, some errors are still likely to be present in the software. The goal of testing is to expose hidden errors by exercising the software on a set of test cases. In its simplest form, a test case consists of program inputs and corresponding expected outputs. Once the software has successfully gone through the testing phase, we have a greater degree of confidence in the software's reliability.

Software testing is very labor intensive and hence also expensive. It can account for 50% of the total cost of software development [6]. Therefore, tools that automate one or more aspects of testing can greatly help in managing the overall cost of testing. Testing techniques can be broadly classified into two categories, functional and structural. Functional testing is concerned with functionality instead of implementation of the program. Therefore, it involves exercising different input and output conditions. Structural testing is concerned with testing the implementation of the program by exercising different programming structures used by the program.

The primary focus of this chapter is structural testing. Before we discuss structural testing in greater detail, let us identify the key aspects of a structural testing strategy.

What is the form of test requirements?

We must identify specific program entities in terms of which test requirements of a program can be stated. When the program is run on a test case, we can analyze the program execution to determine the set of test requirements that are exercised by the test case. Some examples of test requirements are program statements, program paths, definition–use associations and definition–use paths.

0-8493-1240-7/03/$0.00+$1.50

How much testing is enough?

We need to determine when the testing process has been completed. For this purpose we define test coverage criteria. A test coverage criterion specifies a minimal set of test requirements that must be collectively exercised by the test cases on which the program is executed during the testing process. Depending on the criterion, a minimal set of test requirements may or may not be unique. The test criterion also helps us guide the testing process. At any point during the testing of a program, our goal is to run the program on test cases that exercise test requirements that are yet to be covered by any of test cases that have already been executed.

What is the testing strategy?

Typically, first unit testing is employed to test all the modules in the program individually and then integration testing is employed to test the interfaces among the modules. There are different ways to organize the integration testing process. Whereas initially a program is fully tested, during the maintenance stages of a program, regression testing techniques are employed to exercise only the test requirements that are impacted by the program changes. Finally, the generation of test cases required to test a given set of test requirements can be conducted by manual means or some automatic techniques can be employed.

The three types of structural testing techniques are control flow based testing, data flow based testing, and mutation testing. Control flow based test coverage criteria express testing requirements in terms of the nodes, edges or paths in the program control flow graph. The data flow based test coverage criteria express testing in terms of definition–use associations present in the program. Mutation testing begins by creating mutants of the original program that are obtained by making simple changes to the original program. The changes made to the original program correspond to the most likely errors that could be present. The goal of testing is to execute the original program and its mutants on test cases that distinguish them from each other. Although the focus of this chapter is on data flow testing, we also briefly comment on control flow testing for comparison purposes.

The steps of the data flow testing are summarized in Figure 7.1. A test coverage criterion is chosen and test cases are generated to adequately test the program under the selected criterion. When the test coverage criterion is satisfied, testing terminates. If the program output differs from the expected output for some test case, the program is debugged and modified and then testing resumes.

The remainder of this chapter is organized as follows. In Section 7.2 we introduce the various test coverage criteria aimed at exercising the program control flow and data flow characteristics. In Section 7.3 we discuss commonly used strategies for complete testing of a program under a data flow based testing criterion. In Section 7.4 we describe regression testing techniques that are employed after a program change is made during program maintenance. The test input generation problem is considered in Section 7.5. We briefly describe some approaches to automating test case generation. Concluding remarks are given in Section 7.6.

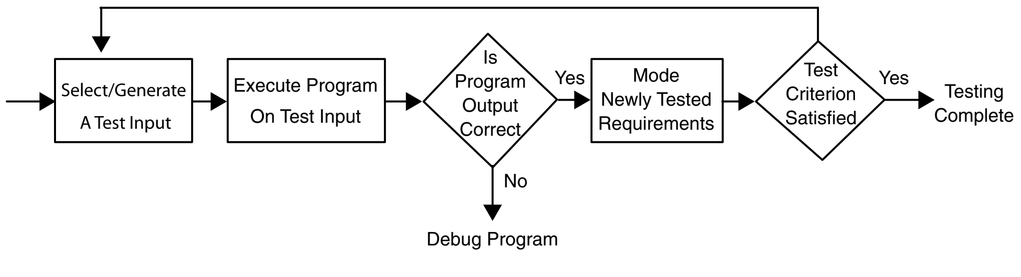

No
Select/Generate A Test Input
Execute Program On Test Input
Is Program Output Correct
Yes
No
Debug Program
Mode Newly Tested Requirements
Test Criterion Satisfied
Yes
Testing Complete

FIGURE 7.1 Testing process.

7.2 Test Coverage Criteria

A test coverage criterion is used to determine whether a program has been adequately tested. It specifies the minimal set of program entities that must be collectively exercised by the test cases on which the program is executed during the testing process. Given a set of test cases, by executing the program on those test cases, and examining the paths followed during the program's executions, we can determine whether a given test coverage criterion is satisfied.

7.2.1 Control Flow Coverage Criteria

A simple form of structural test coverage criteria is based on the characteristics of the program control flow graph. The test coverage criteria included in this category are:

All-nodes test coverage criterion requires that each node in the control flow graph be executed by some test case. Therefore, it is also called *statement testing*.

All-edges test coverage criterion requires that each edge in the control flow graph be traversed during some program execution. This form of testing is also called *branch testing* because each branch outcome is exercised under this criterion. Various variations on branch testing have been proposed to rigorously test the branch predicates [13]. The all-edges criterion subsumes the all-nodes criterion because if all edges in the flow graph are exercised by the test cases, then it is guaranteed that all-nodes are also exercised by the test cases; however, the reverse is not true.

All-paths criterion requires that every complete path (i.e., a path from the entry node to the exit node of the flow graph) in the program be tested. This form of testing is also called *path testing*. The all-paths criterion subsumes the all-edges test criterion. For a program with loops, there may be infinite complete paths. Therefore, proposals have been made to limit the testing to a finite set of paths. A popular technique that falls in this category is McCabe's basis path testing technique. This technique requires a set of paths to be tested that cover all the branches in the program. Therefore, McCabe's basis path testing technique provides exactly the same coverage as the all-edges test coverage criterion.

From the preceding descriptions of the three criterion it is clear that the degree of testing performed differs from one test criterion to another. The all-nodes criterion performs the least amount of testing and is not considered effective in exposing faults because it does not sufficiently exercise interactions between statements. The all-paths criterion is very stringent but it is impractical for realistic programs.

7.2.2 Data Flow Coverage Criteria

Data flow testing was introduced to close the gap between all-edges and all-paths criteria [14, 34, 35]. Data flow testing criteria are aimed at exercising definition–use associations in the program. The exercising of a definition–use association can be viewed as requiring traversal of a selected subpath that originates at the definition, terminates at the use, and is responsible for establishing the definition–use association. Before introducing the data flow based testing criteria, we give the following relevant definitions:

Each occurrence of a variable on the left-hand side of an assignment is a definition of the variable as it updates the value of the variable.

Each occurrence of a variable on the right-hand side of an assignment is called a *computation use*, or a *c-use*.

Each occurrence of a variable in a Boolean predicate results in a pair of predicate uses, or p-uses, corresponding to true and false evaluations of the predicate.

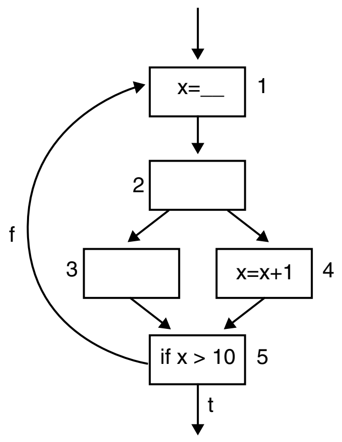


FIGURE 7.2 Illustration of definitions.

A path $(i, n_1, n_2, \ldots, n_m, j)$ is a definition-clear path from the exit of i to entry of j if n_1 through n_m do not contain a definition of x.

Given a definition of x in node n_d and a c-use of x in node n_{c-use}, the presence of a definition-clear path for x from n_d to n_{c-use} establishes the definition-c-use association (n_d, n_{c-use}, x).

Given a definition of x in node n_d and a p-use of x in node n_{p-use}, the presence of a definition-clear path for x from n_d to n_{p-use} establishes a pair of definition-p-use associations $(n_d, (n_{p-use}, t), x)$ and $(n_d, (n_{p-use}, f), x)$ corresponding to true and false evaluations of the predicate in n_{p-use}, respectively.

The paths through the program along which definition–use associations are established are called *du-paths*. A path $(n_1, \ldots, n_j, n_k)$ is a du-path for variable x if n_1 contains a definition of x and either n_k has a c-use of x and $(n_1, \ldots, n_j, n_k)$ is definition-clear simple path for x (i.e., a definition-clear path with all nodes distinct, except possibly the first and last), or n_j is a p-use of x and $(n_1, \ldots, n_j)$ is a definition-clear loop-free path for x (i.e., a definition-clear path in which all nodes are distinct).

The example in Figure 7.2 contains two definitions of x in nodes 1 and 4, one c-use of x in node 4, and a pair of *p-uses* of x due to its use by the predicate in node 5. The paths (1, 2, 3, 5) and (1, 2, 4) are definition clear paths for x. The du-path (1, 2, 4) establishes the definition-c-use association $(1, 4, x)$ while the du-path (1, 2, 3, 5) establishes definition-p-use associations $(1, (5, t), x)$ and $(1, (5, f), x)$.

Given a set of test cases, let P be the set of complete paths exercised by the program executions for these test cases. For each of the data flow based test coverage criterion, the conditions that P must meet for the test criterion to be satisfied are given as follows:

All-defs is satisfied if P includes a definition-clear path from every definition to some corresponding use (c-use or p-use).

All-c-uses is satisfied if P includes a definition-clear path from every definition to all its corresponding c-uses.

All-c-uses or some-p-uses is satisfied if P includes a definition-clear path from every definition to all of its corresponding c-uses. In addition, if a definition has no c-use, then P must include a definition-clear path to some p-use.

All-p-uses is satisfied if P includes a definition-clear path from every definition to all its corresponding p-uses.

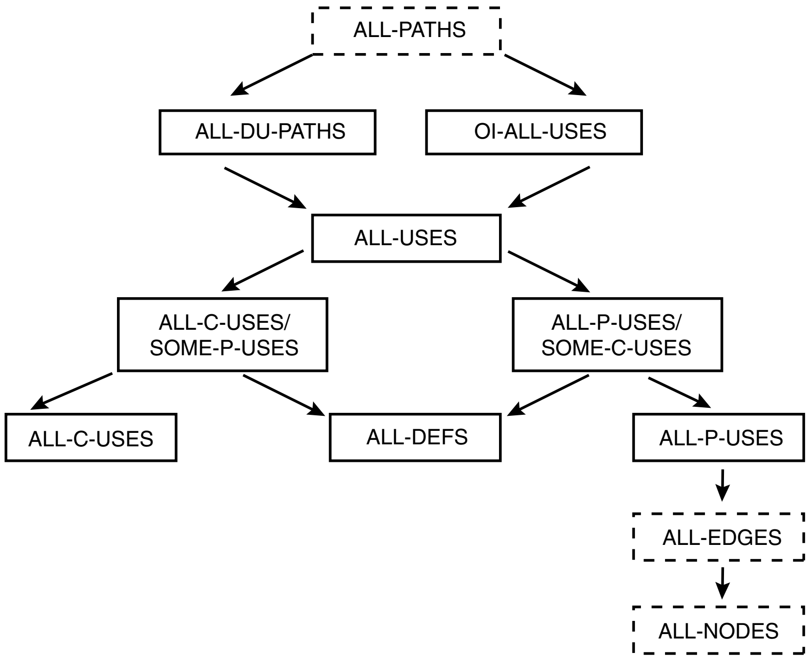


FIGURE 7.3 Test coverage criteria hierarchy.

All-p-uses or some-c-uses is satisfied if *P* includes a definition-clear path from every definition to all of its corresponding p-uses. In addition, if a definition has no p-use, then *P* must include a definition-clear path to some c-use.

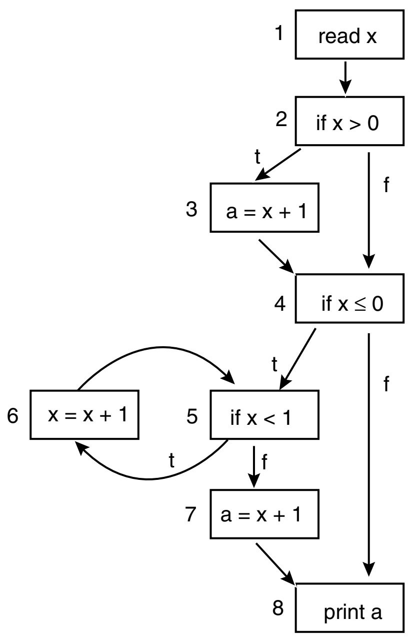
All-uses is satisfied if *P* includes a definition-clear path from every definition to each of its uses including both c-uses and p-uses.

All-du-paths is satisfied if *P* includes all du-paths for each definition. Therefore, if there are multiple paths between a given definition and a use, they must all be included.

Oi-all-uses is satisfied if *P* includes a definition-clear path from every definition to each of its uses including both c-uses and p-uses. Moreover, each use is considered tested only if it is output influencing; that is, it directly or indirectly influences the computation of some program output during the program run [9].

From the preceding definitions of different data flow based criteria it is clear that each criterion identifies a minimal set of definition–use associations that must be exercised by a set of test cases to satisfy the criterion. However, this minimal set is not unique. As was the case with control flow based testing criteria, subsumption relationships also exist between certain data flow based criteria. In Figure 7.3 these subsumption relationships are identified. We have also included the control flow based test coverage criteria in this hierarchy for completeness. Dashed lines are used to distinguish them from the data flow based test coverage criteria.

Let us consider the example in Figure 7.4 to illustrate the test criteria. In addition to the control flow graph, two test inputs and their corresponding complete paths are given in the figure. The two complete paths cover all nodes and edges in the flow graph and therefore satisfy the all-nodes and all-edges criteria. Now let us consider the various data flow based test coverage criteria. The table in Figure 7.4 lists all the definition–use associations present in the program. For each of the data flow based criterion, excluding the all-du-paths and oi-all-uses criteria, a minimal set of associations is identified. If a given set of complete paths exercises the specified associations for a test criterion, then it satisfies the test criterion. In this example, all the specified test criteria are satisfied by the



Test Input	Complete Path
x = 1	P_1: 1-2-3-4-8
x = -1	P_2: 1-2-4-5-6-5-6-5-7-8

Criteria → / Association ↓	all-defs	all-c-uses	all-p-uses	all-c-uses/ some-p-uses	all-p-uses/ some-c-uses	all-uses
(1,(2,t),x)	u		u		u	u
(1,(2,f),x)			u		u	u
(1,3,x)		u		u		u
(1,(4,t),x)			u		u	u
(1,(4,f),x)			u		u	u
(1,(5,t),x)			u		u	u
(1,(5,f),x)			u		u	u
(1,6,x)		u		u		u
(1,7,x)		u		u		u
(3,8,a)	u	u		u	u	u
(7,8,a)	u	u		u	u	u
(6,6,x)	u	u		u		u
(6,7,x)		u		u		u
(6,(5,t),x)			u		u	u
(6,(5,f),x)			u		u	u
Satisfying Paths	$\{P_1,P_2\}$	$\{P_1,P_2\}$	$\{P_1,P_2\}$	$\{P_1,P_2\}$	$\{P_1,P_2\}$	$\{P_1,P_2\}$

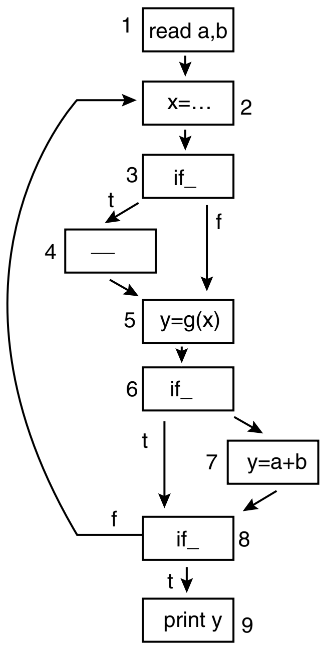
FIGURE 7.4 Example of test criteria.

two complete paths corresponding to the two program inputs. The all-du-paths criteria are also specified by the two complete paths because in this example only a single du-path corresponds to each definition–use association.

From the example in Figure 7.4, we can make the following additional observations:

If a more stringent test coverage criterion is chosen, the number of definition–use associations that must be tested increases. For example, to satisfy the all-defs criterion we need to test only four associations whereas to satisfy the all-uses criterion we must test all associations.

The minimal set of associations that must be tested to satisfy a given criterion is not always unique. For example, to satisfy the all-defs criteria any one of the first nine associations can be selected to test the definition of x in node 1. The test requirements or test cases can be prioritized to



Complete Paths:

P_1 : 1 – 2 – 3 – 4 – 5 – 6 – 8 – 2 – 3 – 5 – 6 – 7 – 8 – 9
P_2 : 1 – 2 – 3 – 5 – 6 – 8 – 9

Complete Path → / *du – paths* for (2,5,x) ↓	P_1	P_2
2-3-4-5	u	
2-3-5	u	u

Complete Path →	P_1	P_2
du – association (2,5,x) influences output *y*		u

FIGURE 7.5 All-du-paths and oi-all-uses coverage criteria.

exploit the preceding characteristic. In particular, priorities can be used to reduce the testing effort [19] or maximize the rate of fault detection [17, 36].

Testing a greater number of associations does not always require execution of greater number of test cases. In our example, although the minimal number of definition–use associations that must be tested to satisfy various criteria varies greatly, all coverage criteria are satisfied by the two test cases. In fact it has been shown that the all-uses criteria are quite practical because relatively few test cases are required to satisfy this criterion [41].

Although it may be possible to find a minimal number of complete paths that satisfy a given test coverage criterion, this approach cannot always be used to minimize the testing effort. This is because some of the complete paths may be infeasible. For example, the complete path 1-2-4-8 for the program in Figure 7.4 is infeasible.

Let us consider another example to illustrate the all-du-paths and oi-all-uses criteria. Figure 7.5 shows two complete paths through the given control flow graph. The definition–use association $(2, 5, x)$ is established along two *du-paths* 2-3-4-5 and 2-3-5. The path P_1 exercises $(2, 5, x)$ along both du-paths and therefore sufficiently tests it to meet the all-du-paths criterion. However, because the value of x at node 5 does not influence the output y along P_1, it does not meet the oi-all-uses criterion for $(2, 5, x)$. On the other hand, although path P_2 exercises only one of the du-paths, the value of x at node 5 does influence the output y. Therefore, although P_2 does not satisfy the all-du-paths criterion for $(2, 5, x)$ it does satisfy the oi-all-uses criterion.

7.3 Complete Testing

Before the software can be deployed it should be throughly tested. For large software, testing is carried out in a number of steps. First, the individual procedures are tested one at a time during unit testing. The interfaces between the procedures are tested next during integration testing. A number of researchers have explored data flow testing methodologies for complete testing. Some of these techniques are suitable for unit testing because they are intraprocedural (i.e., their applicability is restricted to procedure bodies) [4, 15, 31]. Others have developed integration testing techniques that are interprocedural and therefore these techniques consider definition–use associations that extend across procedure boundaries [8, 26].

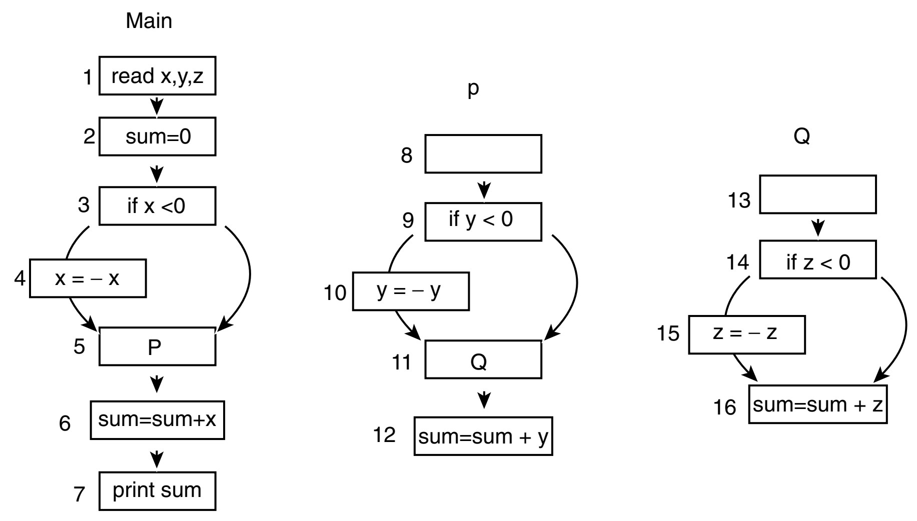
In this section we present an overview of unit and integration testing assuming that the all-uses criterion is employed. During each step in testing we must identify the test requirements in form of definition–use associations that are to be tested. Once the test requirements have been identified, we must generate test cases to exercise the definition–use associations. One approach is to select a path along which at least one untested definition–use association is exercised. Test data are generated that cause the program to take the desired path and the definition–use association is tested. This process is repeatedly applied to test all the test requirements.

Unit testing considers one procedure at a time and exercises the definition–use associations within that procedure. A definition–use association within a procedure, also referred to as a intraprocedural definition–use association, is one for which the definition and the use belong to the procedure and a definition-clear path can be found within the procedure without examining the code belonging to any called procedures. In other words, to identify intraprocedural definition–use associations for a procedure, we conservatively assume that all variables visible to the called procedures are killed by them. A procedure may read values of global variables and may have input parameters. Definitions for input parameters and globals are introduced at the beginning of the procedure in form of read statements. These definitions also give rise to definition–use associations that must be tested. Once the definition–use associations have been computed the testing of the procedure can proceed using the path selection based strategy described earlier.

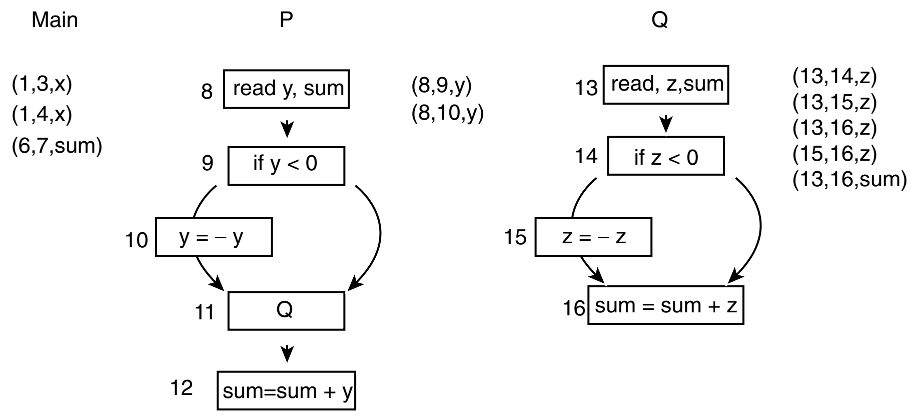
Integration testing exercises the interprocedural definition–use associations during one or more integration steps. An interprocedural definition–use association may be formed in a situation where the definition and use belong to different procedures. It can also be formed in cases where although the definition and use are in the same procedure, all the definition-clear paths between them contain procedure calls. In each integration step one or more procedures are selected for integration. The program call graph is used to drive the selection process. For example, integration can be carried out in a top-down fashion or bottom-up fashion according to the call graph. In each integration step the interprocedural definition–use associations relevant for that step must be tested. The computation of interprocedural definition–use associations can be carried in a couple of different ways. One approach analyzes the complete program and computes all the definition–use associations prior to integration testing [27]. Another approach partially analyzes the program to compute the relevant interprocedural definition–use associations during an integration step using demand-driven data flow analysis [10].

The example in Figure 7.6 illustrates unit and integration testing. The sample program contains four variables that are all globals. The definition–use associations that are generated as requirements for unit testing are given in Figure 7.6(b). To compute definition–use associations to unit test P and Q, definitions for globals referenced by these procedures are introduced in the entry nodes of the procedures. During integration testing we assume that a bottom-up strategy is used. Therefore, we first integrate Q into P and then we integrate P and *Main*. The interprocedural definition–use associations that must be tested during the integration steps are given in Figure 7.6(c).

To carry out the testing of the requirements during unit and integration testing we must generate test inputs. The test data generation problem is discussed later in this chapter. After the program is executed on a test input we would like to identify all the definition–use associations exercised by that input. For this purpose we can instrument the program to either generate an execution trace that can be later analyzed or dynamically track the definition–use associations that are exercised. Techniques used by dynamic slicing algorithms can be used for this purpose [1, 30].



(a) A Sample Program.



(b) Requirements for Unit Testing.

Integrate Q with P	Integrate P with Main
(8,16,sum)	(2,16,sum)
(16,12,sum)	(12,6,sum)
(10,12,y)	(1,9,y)
	(1,10,y)
	(1,14,z)
	(1,15,z)

(c) Requirements for Integration Testing Steps.

FIGURE 7.6 Example of complete testing.

7.4 Regression Testing

Program changes are sometimes required after the deployment of software. The two common reasons for such changes are (1) although the software may have been completely tested prior to its deployment, some errors may only be discovered once the software is in use; and (2) some changes may be made to better meet the needs of the customer. Regression testing is the process of testing the modified parts of the software and ensuring that no new errors have been introduced into previously tested code. Therefore, regression testing must test both the modified code and other parts of the program that may be affected by the program change. Researchers have developed a number of data flow based regression testing techniques [20, 33, 37].

In this section we describe an approach for regression testing proposed by Gupta et al. [20]. In this approach a program change is translated into a series of low-level program edits. Demand-driven static slicing algorithms are used to identify definition–use associations that are affected by each program edit. The complete set of definition–use associations that are affected by a series of program edits corresponding to a program change are retested during regression testing. During regression testing we make use of existing test cases that were developed during the initial exhaustive testing of the software by maintaining the test suite [25]. However, in general, some new test cases may also have to be generated to adequately test the modified code.

The program change is mapped into a series of low-level program edits. We identify a complete set of program edits such that changes to the program can always be mapped into a series of program edits. This list of edits follows, with the edits defined in terms of operations on the program control flow graph:

1. Delete a use of a variable from an assignment statement.
2. Insert a use of a variable in an assignment statement.
3. Delete a definition of a variable from an assignment statement.
4. Insert a definition of a variable in an assignment statement.
5. Delete a use of a variable from a conditional statement.
6. Insert a use of a variable in a conditional statement.
7. Change an operator or a constant in an assignment statement.
8. Change an operator or a constant in a conditional statement.
9. Insert an edge.
10. Delete an edge.
11. Insert a new assignment statement.
12. Insert a new conditional statement.

Let us assume that we used the all-uses criterion during complete testing. During regression testing, given a program edit, the newly created definition–use associations not only should be tested, but also all definition–use associations that are affected directly or indirectly by the program edit must be retested. The definition–use associations that should be tested during regression testing are classified into the following three categories:

New associations. A program edit can cause the creation of new definition–use associations that must be tested. In particular, edit types 2, 3, 4, 6, 9, 11 and 12 can create new definition–use associations.

Value associations. If an existing definition of a variable is directly or indirectly impacted by a program edit, such that the value assigned in the definition is changed, then all definition–use associations for this definition must be retested.

Path associations. If a program edit directly or indirectly affects a predicate, then definition–use associations impacted by the predicate are also tested. A predicate is affected by a program edit if either the edit explicitly modifies the predicate or the predicate contains a use belonging to a new or value association. A definition–use association is impacted by a predicate if the definition in the association is control dependent on the predicate.

The example in Figure 7.7, taken from [20], illustrates the preceding approach. This program iteratively computes the square root of x. It has an error in statement 8 in which $x1$ should be assigned the value of $x3$ instead of $x2$. Correcting the error can be expressed in terms of two low-level edits: deletion of the use of $x2$ from statement 8 followed by the insertion of the use of $x3$ in its place. The definition–use associations that must be tested due to these edits are as follows:

New associations. The insertion of the new use of $x3$ in statement 8 creates a new definition–use association $(5, 8, x3)$.

Value associations. Because the value of $x1$ defined by statement 8 is affected, the value associations $(8, (4, t), x1)$, $(8, (4, f), x1)$, $(8, 5, x1)$, $(8, (6, t), x1)$ and $(8, (6, f), x1)$ must be tested. The use of $x1$ in statement 5 affects the definition of $x3$ and therefore value associations $(5, 7, x3)$, $(5, (6, t), x3)$ and $(5, (6, f), x3)$ must also be tested. Finally, the use of $x3$ in statement 7 affects the definition of $x2$ and thus the value associations $(7, (4, t), x2)$, $(7, (4, f), x2)$ and $(7, 5, x2)$ must also be tested.

Path associations. The preceding value associations indicate that the predicates 4 and 6 are affected by the edits. Therefore, path associations created by definitions that are control dependent on predicates 4 and 6 must be tested. The definitions that are control dependent on these predicates include statements 5, 7 and 8. Because all definition–use associations for these three statements have already been marked for testing as value associations, all path associations are already included in the preceding set of value associations.

The modified program must be analyzed to identify the definition–use associations that have to be tested. One approach for performing such analysis is to exhaustively compute definition–use associations for the program prior to an edit as well as following an edit. By comparing the two the

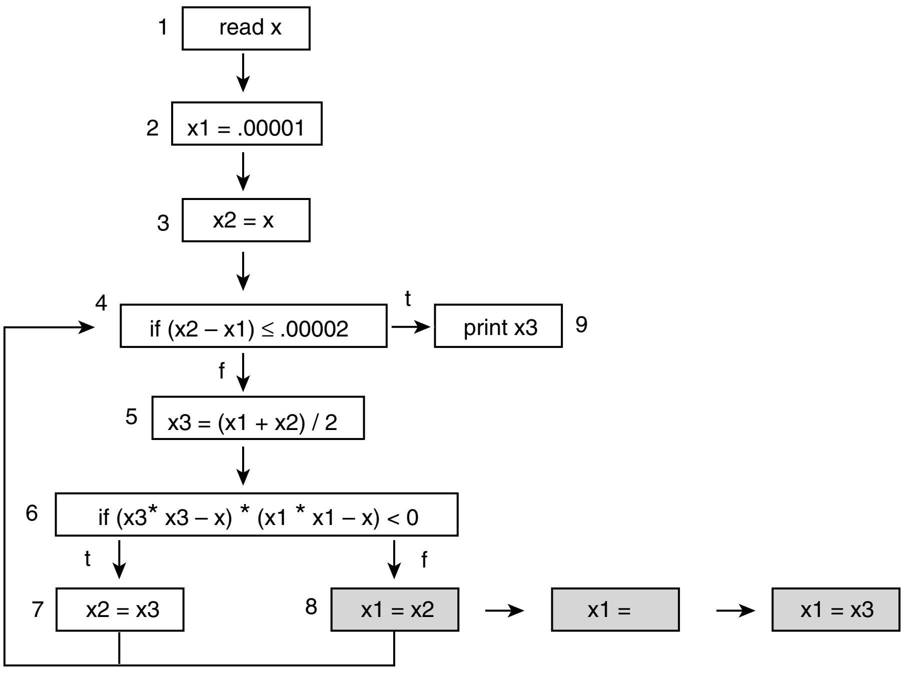


FIGURE 7.7 Testing following program edits.

```
Reaching Defs (v, n){
    Initialize Worklist to contain node n
    While Worklist is not empty do
            remove a node. say m, from Worklist
            if m defines variable v then
                add m to DefsFound
            else
                add predecessor nodes of m to Worklist
            endif
    endwhile
    return (DefsFound)
}
```

(a) Finding Reaching Definitions Using Backward Analysis.

```
ReachableUses (v, n)
    Initial Worklist to contain node n
    While Worklist is not empty do
            remove a node, say m, from Worklist
            if m contains a use of variable v then
                add m to UsesFound
            endif
            if m does not define variable v then
                add successor nodes of m to Worklist
            endif
    endwhile
    return(UsesFound)
}
```

(b) Finding Reachable Uses Using Forward Analysis

FIGURE 7.8 Reaching definitions and reachable uses analysis.

new definition–use associations are identified. By starting at the new associations and performing a forward static slicing operation over the definition–use associations, the value associations are identified. The predicates involved in new and value associations are now known and the definitions that are control dependent on them are found. By performing forward static slicing operations from these definitions all the path associations are found.

Another approach that only partially computes the definition–use associations before and after the program edit is based on demand-driven analysis. Demand-driven analysis algorithms can be devised to identify only those associations that are of interest. As an example consider a situation in which a definition of variable v has been deleted from node n. We would like to identify the new definition–use associations for v that may have been established. By using the algorithms shown in Figure 7.8, we can compute the definitions of v that reach the entry of node n (*ReachingDefs(v,n)*) as well as the uses of v that are reachable from the exit of n (i.e., $ReachableUses(v, n)$). All definition–use associations for v that are established through n are now known. If some of these associations did not exist in the program prior to the edit, then they are new associations. The advantage of this approach is that the new associations have been found without exhaustively analyzing the program. The first algorithm for demand-driven computation of definition–use associations that we are aware of was proposed by Weiser, who used this approach to compute static slices [18, 39]. Since then this approach has been used to deal with program edits in other works [18, 20].

7.5 Test Input Data Generation

One of the most difficult tasks one faces during the initial testing of a program is that of generating test cases to exercise the various test requirements. Whereas during functional testing the test cases are generated from the functional specification, during data flow testing test cases must be generated to provide adequate coverage under the selected test criterion for a given implementation. One by one as the test cases are generated, the program is executed on them and the execution is analyzed to identify the test requirements that are covered by the test case. As testing proceeds, an attempt must be made to generate test cases that exercise test requirements yet to be covered.

7.5.1 Manual Test Input Generation

One approach to test input generation leaves the task of generating test cases to the person testing the software. Even under this situation some of the other tasks can still be carried out automatically. In particular, the determination of overall test requirements, the tracking of test requirements exercised by a given test case and the identification of test requirements that must be exercised to satisfy the test coverage criterion can all be carried out automatically.

The generation of test cases is extremely labor intensive and time consuming. Therefore, the use of static analysis information has been explored to guide the user during the generation of test data [21]. In particular, static information is used to identify paths covering one or more test requirements that are yet to be tested. Another problem that is encountered is due to the presence of infeasible program paths. Because the identification of test requirements is based on static analysis, it is possible that some of the identified test requirements are in fact infeasible; that is, there is no program input on which they are exercised. The generation of infeasible test requirements is unavoidable because static analysis is based on the assumption that all program paths are feasible. The time spent in trying to generate a test case for an infeasible test requirement is clearly wasted. In fact, it is quite possible that the test engineer discovers that a test requirement is infeasible only after spending a great deal of time trying to generate a test case for the requirement.

To reduce the time wasted on infeasible requirements, an approach was proposed by Bodik et al. [2]. A simple static analysis is described to identify some of the infeasible test requirements. These requirements can be removed from consideration during the testing process. The basic idea behind this approach is to identify smallest subpaths through the control flow graph that are infeasible. The traditional data flow analysis for identifying definition–use associations is then modified such that it prevents propagation of data flow information along infeasible subpaths. Consider the example in Figure 7.9 that contains an infeasible definition-c-use association $(1, 4, a)$. This association can be identified as infeasible by the analysis in [2]. In particular, it is determined that if $x < y$ is true, then $x < y + 1$ must also be true; therefore, the subpath 1-2-4 is infeasible.

7.5.2 Automated Test Input Generation

Because test input generation is a time-consuming process, researchers have proposed techniques for automating test input generation [7, 12, 16, 22−24, 29]. In this section we describe an approach that has been proposed to automatically generate test data. We first consider a class of programs in which the branch predicates involve computations that can be expressed as linear functions of the program inputs. Later we show how this condition can be relaxed to allow for more general programs.

The high-level view of the automated test data generation method that we present is as follows. We first select a program path that exercises the test requirement considered. The task of the test generator is to identify a program input that can cause the program execution to follow the selected path. To carry out this task we consider each branch predicate evaluation along the selected program path; then by analyzing the code along the program path, we derive a linear constraint on the

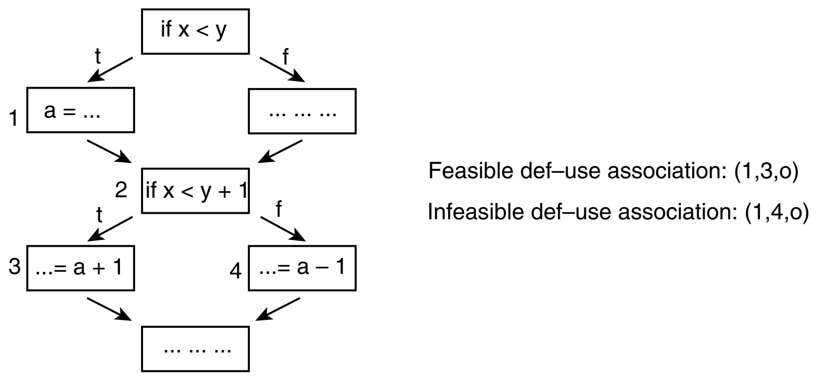


FIGURE 7.9 Feasible vs. infeasible definition–use associations.

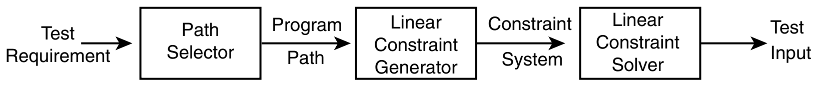


FIGURE 7.10 Automated test data generation and testing.

program's inputs that must be satisfied for that branch predicate to evaluate correctly. This process is repeated for each branch predicate evaluation and a system of linear constraints is obtained. Any constraint solver, such as XA [43] or UNA [23], can then be used to solve the linear constraint system (Figure 7.10).

The preceding approach deals with infeasible path problem quite naturally. If the path chosen is infeasible, then the set of constraints that result are inconsistent and therefore no solution can be found. It is possible that certain constraints on the input domain exist (e.g., an input value may have to be a positive integer to be meaningful). These constraints can also be naturally handled in conjunction with the derived constraints by the constraint solver.

The key step of the preceding process is the derivation of the system of linear constraints from the program. We discuss two approaches to carry out this task: one is based on a symbolic evlauation [3] and the other is based on program execution [24]. We illustrate these approaches through an example program of Figure 7.11, which reads three input values into variables *a*, *b* and *c* and then changes the contents of the variables such that eventually *a* contains the smallest number and *c* contains the largest number. We consider the generation of test data for a couple of paths through this program, one feasible and another infeasible.

7.5.2.1 Symbolic Evaluation-Based Derivation of Linear Constraints

Let us consider the symbolic evaluation approach. For each branch predicate evaluation along the selected path, we prepare a straight-line code segment consisting of all statements along the path except for the earlier branch predicate evaluations. This straight-line code is symbolically executed so that the branch predicate evaluation under consideration can be expressed directly in terms of the program inputs. The desired evaluation of the branch predicate imposes a constraint on the inputs that must be satisfied by the test input. This process is repeated for each branch predicate evaluation along the selected path and a system of constraints is derived.

Figure 7.12 illustrates this approach for the sorting example of Figure 7.11. The first path considered is a feasible path. The code along the path is given and the desired branch predicate evaluations are indicated. For simplicity the statements that implement the loop, causing it to iterate twice, are ignored. No input constraints result from analyzing the loop predicate evaluations because the number of times the loop iterates is fixed and independent of the program input. This fact can also

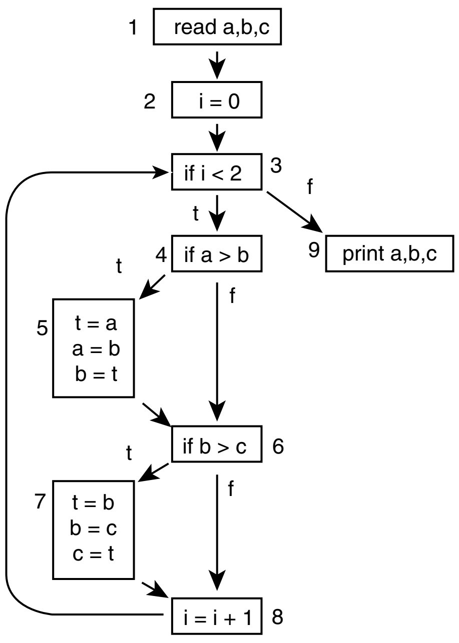


FIGURE 7.11 Example for test data generation.

be easily discovered through symbolic evaluation. When symbolic evaluation of the code along the path to other branch predicate evaluations is carried out, we obtain a system of constraints shown in Figure 7.12. This indicates that for the path to be followed, we must select input values for a, b and c such that $a > b > c$.

If we consider the second path shown in Figure 7.12, we notice that this path only differs from the first path in its evaluation of the last branch predicate (true instead of false). Although the first three constraints are the same, the fourth constraint we obtain is different ($b > a$ instead of $b \not> a$). We can clearly see that the constraint system that we have obtained now has no solution because the constraints corresponding to the first and last branch predicate evaluations cannot be simultaneously satisfied. Therefore, we can conclude that the selected path is infeasible.

7.5.2.2 Execution Based Derivation of Linear Constraints

Next we consider an execution-based approach for deriving the constraints. Given a Boolean predicate, we can write this predicate in a normalized form $P \; relop \; 0$, where $relop$ is a relational operator. Our goal is to find a linear representation for P in terms of the program inputs. If P is a linear function of program inputs, it can be expressed in the following form where i_1 through i_n are the inputs and x_0 through x_n are constants:

$$P = x_0 + x_1 \times i_1 + x_2 \times i_2 + \cdots + x_n \times i_n$$

If we can find the values of the constants in the preceding representation, we have the linear representation of P in terms of the program inputs. To do so we execute the statements along the path leading to P, excluding the prior branch predicates, on $n + 1$ arbitrary input data sets and print out the values of P for each of these input data sets. By substituting the input data values and

Test data for feasible path: 1-2-3(t)-4(t)-5-6(t)-7-8-3(t)-4(t)-5-6(f)-8-3(f)-9

read a, b, c	read a, b, c	read a, b, c	read a, b, c	read a, b, c
if $a >$ [true]	if $a > b$ [true]	$t = a$	$t = a$	$t = a$
$t = a$	$\Downarrow$	$a = b$	$a = b$	$a = b$
$a = b$	$\Downarrow$	$b = t$	$b = t$	$b = t$
$b = t$	$\Downarrow$	if $b > c$ [true]	$t = b$	$t = b$
if $b > c$ [true]	$\Downarrow$	$\Downarrow$	$b = c$	$b = c$
$t = b$	$\Downarrow$	$\Downarrow$	$c = t$	$c = t$
$b = c$	$\Downarrow$	$\Downarrow$	if $a > b$ [true]	$t = a$
$c = t$	$\Downarrow$	$\Downarrow$	$\Downarrow$	$a = b$
if $a > b$ [true]	$\Downarrow$	$\Downarrow$	$\Downarrow$	$b = t$
$t = a$	$\Downarrow$	$\Downarrow$	$\Downarrow$	if $b > c$ [false]
$a = b$	$\Downarrow$	$\Downarrow$	$\Downarrow$	$\Downarrow$
$b = t$	$\Downarrow$	$\Downarrow$	$\Downarrow$	$\Downarrow$
if $b > c$ [false]	$a > b$	$a > c$	$b > c$	$b \not> a$ or $b \leq a$
print a,b,c				

$$\begin{array}{lcl} a > b & & \\ a > c & \Rightarrow & a > b > c \\ b > c & & \\ b \not> a \text{ or } b \leq a & & \end{array}$$

Test data for infeasible path: 1-2-3(t)-4(t)-5-6(t)-7-8-3(t)-4(t)-5-6(t)-7-8-3(f)-9

read a, b, c	read a, b, c	read a, b, c	read a, b, c	read a, b, c
if $a >$ [true]	if $a > b$ [true]	$t = a$	$t = a$	$t = a$
$t = a$	$\Downarrow$	$a = b$	$a = b$	$a = b$
$a = b$	$\Downarrow$	$b = t$	$b = t$	$b = t$
$b = t$	$\Downarrow$	if $b > c$ [true]	$t = b$	$t = b$
if $b > c$ [true]	$\Downarrow$	$\Downarrow$	$b = c$	$b = c$
$t = b$	$\Downarrow$	$\Downarrow$	$c = t$	$c = t$
$b = c$	$\Downarrow$	$\Downarrow$	if $a > b$ [true]	$t = a$
$c = t$	$\Downarrow$	$\Downarrow$	$\Downarrow$	$a = b$
if $a > b$ [true]	$\Downarrow$	$\Downarrow$	$\Downarrow$	$b = t$
$t = a$	$\Downarrow$	$\Downarrow$	$\Downarrow$	if $b > c$ [true]
$a = b$	$\Downarrow$	$\Downarrow$	$\Downarrow$	$\Downarrow$
$b = t$	$\Downarrow$	$\Downarrow$	$\Downarrow$	$\Downarrow$
if $b > c$ [true]	$a > b$	$a > c$	$b > c$	$b > a$
print a,b,c				

$$\begin{array}{lcl} a > b & & \\ a > c & \Rightarrow & \text{Infeasible Path} \\ b > c & & \\ b > a & & \end{array}$$

FIGURE 7.12 Constraint derivation using symbolic evaluation.

the corresponding printed value of P in the preceding equation, we obtain the following system of equations:

$$P[0] = x_0 + x_1 \times i_1[0] + x_2 \times i_2[0] + \cdots + x_n \times i_n[0]$$

$$P[1] = x_0 + x_1 \times i_1[1] + x_2 \times i_2[1] + \cdots + x_n \times i_n[1]$$

$$\ldots$$

$$P[n] = x_0 + x_1 \times i_1[1] + x_2 \times i_2[n] + \cdots + x_n \times i_n[n]$$

Therefore, we have a system of $n + 1$ equations in $n + 1$ unknowns that can be solved to compute the values of x_0 through x_n using standard packages such as MATLAB [42]. The result of the preceding computation is the linear representation of P in terms of program inputs.

Figure 7.13 illustrates the derivation of a predicate expression using the execution-based approach. In this example we derive the linear representation of the last predicate expression corresponding to the feasible path of Figure 7.12. The predicate is represented by $f(a, b, c)$ in terms of the three program inputs. The code along the path is executed on some arbitrarily selected inputs and from the resulting values of f, we obtain the system of equations from which the values of constant coefficients are computed. As we can see, this approach yields the constraint $b \not> a$, which is the same as the earlier representation that we obtained using the symbolic execution approach.

There are some advantages to using the execution-based approach. In the example we considered, it was fairly straightforward to carry out the symbolic evaluation because variable names referenced were statically unambiguous, no array or pointer references existed in the code and finally expressions under manipulation were small and simple. For long paths containing array and pointer references, symbolic evaluation can become impossible to perform. However, the execution-based approach is free of these limitations.

7.5.2.3 Nonlinear Constraints

We have restricted our discussion to situations in which the predicate expressions were linear functions of the program inputs. In general, this may not be the case and some of the predicate expressions may be nonlinear functions of the inputs. The presence of nonlinear predicate expressions poses challenges both in deriving the predicate expressions and solving the nonlinear constraint system. In principle, the symbolic evaluation approach is not restricted to a linear representation. However, the simplification of nonlinear expressions during symbolic evaluation is far more complex. The execution-based approach has a rather strict limitation. It requires that the form of the predicate expression in terms of program inputs be known. However, it is not realistic to assume that this would be the case. Even if the nonlinear predicate expressions are identified, solving the resulting system of nonlinear constraints is a problem.

An approach that addresses the preceding problem was proposed by Gupta et al. in [23, 24]. The predicate expressions are handled as follows:

All predicate expressions are treated uniformly by assuming that they are linear functions of the inputs. The execution-based approach is used to derive these linear representations. If the predicate expression is linear, we have chosen the correct form. However, if a predicate expression has a nonlinear representation, the linear representation that is derived is an approximation of the actual nonlinear representation.

A nonlinear function cannot be accurately approximated by a linear function across the entire input domain. This problem is addressed as follows. Instead of having a single linear representation for the predicate expression for all program inputs, the linear representation is derived with

```
read a, b, c
t = a
a = b
b = t
t = b
b = c
c = t
t = a
a = b
b = t
if b > c [false]
```

if $b > c$ [false]

$\Downarrow$

if $b > c$ [false]

$\Downarrow$

if $f(a, b, c) > 0$ [false]

where, $f(a, b, c) = x_1 \times a + x_2 \times b + x_3 \times c$

$\Downarrow$

```
f(a, b, c){
  read a, b, c
   t = a
   a = b
   b = t
   t = b
   b = c
   c = t
   t = a
   a = b
   b = t
 print (f(a, b, c) = b − c)
}
```

$f(0, 5, 10) = 5$

$f(1, 6, 11) = 5$

$f(2, 8, 10) = 6$

$f(3, 9, 12) = 6$

$$
\begin{aligned}
f(0, 5, 10) = 5 &\Rightarrow x_0 + x_1 \times 0 + x_2 \times 5 + x_3 \times 10 = 5\\
f(1, 6, 11) = 5 &\Rightarrow x_0 + x_1 \times 1 + x_2 \times 6 + x_3 \times 11 = 5\\
f(2, 8, 10) = 6 &\Rightarrow x_0 + x_1 \times 2 + x_2 \times 8 + x_3 \times 10 = 6\\
f(3, 9, 12) = 6 &\Rightarrow x_0 + x_1 \times 3 + x_2 \times 9 + x_3 \times 12 = 6
\end{aligned}
$$

$$
\begin{array}{lcl}
x_0 + 5 \times x_2 + 10 \times x_3 = 5 & & x_0 = 0\\
x_0 + x_1 + 6 \times x_2 + 11 \times x_3 = 5 & \Rightarrow & x_1 = -1\\
x_0 + 2 \times x_1 + 8 \times x_2 + 10 \times x_3 = 6 & & x_2 = 1\\
x_0 + 3 \times x_1 + 9 \times x_2 + 12 \times x_3 = 6 & & x_3 = 0
\end{array}
$$

$$f(a, b, c) = x_0 + x_1 \times a + x_2 \times b + x_3 \times c = b - a \Rightarrow \text{if } b > a \text{ [false]} \Rightarrow b \not> a \text{ or } b \le a$$

FIGURE 7.13 Execution-based constraint derivation.

respect to each program input. In fact, the linear approximation represents the tangent plane to the nonlinear expression for a given input. Because the approximate linear representation is specific to a program input, a specific input must be chosen before the representations can be derived.

The preceding treatment of predicate expressions leads to the overall approach shown in Figure 7.14 that uses an iterative relaxation method to generate test input. Test data generation is initiated with an arbitrarily chosen input from a given domain. This input is then iteratively refined to obtain an input on which all the predicate expressions on the given path evaluate to the desired outcome. In each iteration the program statements relevant to the evaluation of each branch predicates on the path are executed, and a set of linear constraints is derived. The constraints are then solved to obtain the increments for the input. These increments are added to the current input to obtain the input for the next iteration. The relaxation technique used in deriving the constraints provides feedback on the amount by which each input variable should be adjusted for the branches on the path to evaluate to the desired outcome.

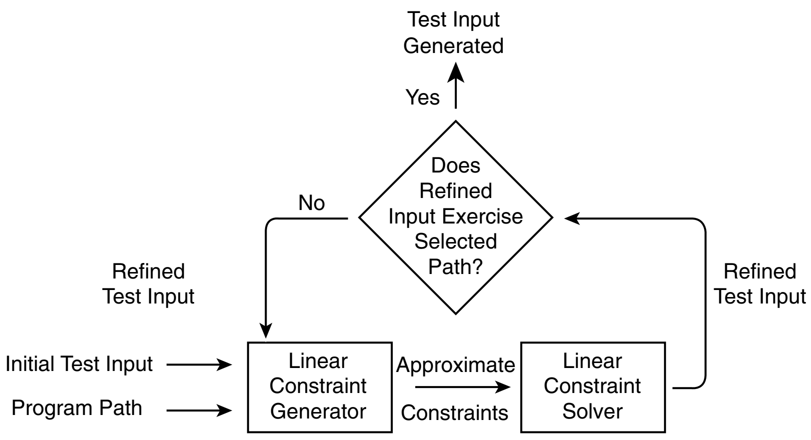


FIGURE 7.14 Iterative refinement of test data.

7.6 Concluding Remarks

In this chapter we provide an overview of different aspects of data flow based testing of programs. We describe the various data flow based test coverage criterion and the relationships among them. The various aspects of data flow testing can be automated to a significant extent. In particular, static data flow analysis is used to identify test requirements, dynamic analysis is used to track test requirements exercised during a given program execution, test requirements that remain to be tested can be automatically tracked and finally automatic determination can be made when enough testing has been performed to satisfy a given test coverage criterion. We also describe techniques that automate the difficult task of generating test cases. Although we consider the testing of sequential programs, researchers are also developing testing strategies for dealing with object-oriented programs [28], and parallel and real-time programs [5, 11, 38].

References

[1] H. Agrawal and J.R. Horgan, Dynamic Program Slicing, ACM SIGPLAN Conference on Programming Language Design and Implementation, June 1990, pp. 246–256.

[2] R. Bodik, R. Gupta and M.L. Soffa, Refining data flow information using infeasible paths, in *ACM SIGSOFT 5th Symposium on Foundations of Software Engineering and 6th European Software Engineering Conference*, LNCS 1301, Springer-Verlag, September 1997, pp. 361–377.

[3] L.A. Clarke, A system to generate test data and symbolically execute programs, *IEEE Trans. Software Eng.*, SE-2(3), 215–222, September 1976.

[4] L.A. Clarke, A. Podgurski, D.J. Richardson and S.J. Zeil, A formal evaluation of data flow path selection criteria, *IEEE Trans. Software Eng.*, SE-15(11), 1318–1332, November 1989.

[5] J.C. Corbett, Timing analysis of Ada testing programs, *IEEE Trans. Software Eng.*, 22(7), July 1996.

[6] R. De Millo, W. McCracken, R. Martin and J. Passafiume, *Software Testing and Evaluation*, Benjamin/Cummings, Menlo Park, CA, 1987.

[7] R.A. DeMillo and A.J. Offutt, Constraint-based automatic test data generation, *IEEE Trans. Software Eng.*, 17(9), 900–910, September 1991.

[8] E. Duesterwald, R. Gupta and M.L. Soffa, A Demand-Driven Analyzer for Data Flow Testing at the Integration Level, SIGSOFT/IEEE 18th International Conference on Software Engineering, 1996, pp. 575–584.

[9] E. Duesterwald, R. Gupta and M.L. Soffa, Rigorous Data Flow Testing through Output Influences, 2nd Irvine Software Symposium, March 1992, 131–145.

[10] E. Duesterwald, R. Gupta and M.L. Soffa, A practical framework for demand-driven interprocedural data flow analysis, *ACM Trans. Programming Languages Syst.*, 19(6), November 1997, 992–1030.

[11] M.B. Dwyer and L.A. Clarke, Data Flow Analysis for Verifying Properties of Concurrent Programs, ACM SIGSOFT International Symposium on Foundations of Software Engineering, December 1994, pp. 62–75.

[12] R. Ferguson and B. Korel, The chaining approach for software test data generation, *ACM Trans. Software Eng. Methodol.*, 5(1), 63–86, January 1996.

[13] P.G. Frankl and E.J. Weyuker, Provable improvements on branch testing, *IEEE Trans. Software Eng.*, 19(10), 962–975, October 1993.

[14] P.G. Frankl and E.J. Weyuker, An applicable family of data flow testing criteria, *IEEE Trans. Software Eng.*, 14(10), 1483–1498, October 1988.

[15] P.G. Frankl and E.J. Weyuker, An Analytical Comparison of the Fault-detecting Ability of Data Flow Testing Techniques, SIGSOFT/IEEE 15th International Conference on Software Engineering, 1993, pp. 415–424.

[16] M.J. Gallagher and V.L. Narsimhan, ADTEST: a test data generation suite for Ada software systems, *IEEE Trans. Software Eng.*, 23(8), August 1997, 473–484.

[17] T.L. Graves, M.J. Harrold, J.-M. Kim, A. Porter and G. Rothermel, An empirical study of regression test selection techniques, *ACM Trans. Software Eng. Methodol.*, 10(2), 184–208, April 2001.

[18] R. Gupta and M.L. Soffa, A Framework for Partial Data Flow Analysis, in IEEE International Conference on Software Maintenance, September 1994, pp. 4–13.

[19] R. Gupta and M.L. Soffa, Priority Based Data Flow Testing, in IEEE International Conference on Software Maintenance, October 1995, pp. 348–357.

[20] R. Gupta, M.J. Harrold and M.L. Soffa, An Approach to Regression Testing Using Slicing, IEEE International Conference on Software Maintenance, November 1992, pp. 299–308.

[21] R. Gupta and M.L. Soffa, Employing static information in the generation of test cases, *J. Software Testing, Verification Reliability*, 3, 29–48, 1993.

[22] N. Gupta, A.P. Mathur and M.L. Soffa, Generating Test Data for Branch Coverage, 15th IEEE International Conference on Automated Software Engineering, September 2000, pp. 219–227.

[23] N. Gupta, A.P. Mathur and M.L. Soffa, UNA Based Iterative Test Data Generation and Its Evaluation, 14th IEEE International Conference on Automated Software Engineering, October 1999, pp. 224–232.

[24] N. Gupta, A.P. Mathur and M.L. Soffa, Automated Test Data Generation Using an Iterative Relaxation Method, ACM SIGSOFT Sixth International Symposium on Foundations of Software Engineering, November 1998, pp. 231–244.

[25] M.J. Harrold, R. Gupta and M.L. Soffa, A methodology for controlling the size of the test suite, *ACM Trans. Software Eng. Methodol.*, 2(3), 270–285, July 1993.

[26] M.J. Harrold and M.L. Soffa, Interprocedural Data Flow Testing, ACM SIGSOFT Third Symposium on Software Testing, Analysis, and Verification, 1989, pp. 158–167.

[27] M.J. Harrold and M.L. Soffa, Efficient computation of interprocedural definition-use chains, *ACM Trans. Programming Languages Syst.*, 16(2), 175–204, March 1994.

[28] P.C. Jorgensen and C. Erickson, Object-oriented integration testing, *Commun. ACM*, 37(9), 30–38, September 1994.

[29] B. Korel, Automated software test data generation, *IEEE Trans. Software Eng.*, 16(8), August 1990, pp. 870–879.

[30] B. Korel and J. Laski, Dynamic program slicing, *Inf. Process. Lett.*, 29, 155–163, October 1988.

[31] J.W. Laski and B. Korel, A data flow oriented program testing strategy, *IEEE Trans. Software Eng.*, SE-9(3), 347–354, May 1983.

[32] T.J. McCabe, Structural Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric, National Bureau of Standards (now NIST), Publication 500-99, Washington, D.C., 1982.

[33] T. Ostrand and E.J. Weyuker, Using Data Flow Analysis for Regression Testing, Sixth Annual Pacific Northwest Software Quality Conference, September 1988, pp. 58–71.

[34] A.S. Parrish and S.H. Zweben, On the relationships among the all-uses, all-du-paths, and all-edges testing criteria, *IEEE Trans. Software Eng.*, 21(12), 1006–1009, December 1995.

[35] S. Rapps and E.J. Weyuker, Selecting software test data using data flow information, *IEEE Trans. Software Eng.*, SE-11(4), April 1985, pp. 367–375.

[36] G. Rothermel, R.H. Untch, C.-C. Chu and M.J. Harrold, Prioritizing test cases for regression testing, *IEEE Trans. Software Eng.*, 27(10), 929–948, October 2001.

[37] A.M. Taha, S.M. Thebut and S.S. Liu, An approach to software fault localization and revalidation based on incremental data flow analysis, *COMPSAC*, 527–534, September 1989.

[38] R.N. Taylor, D.L. Levine and C.D. Kelly, Structural testing of concurrent programs, *IEEE Trans. Software Eng.*, 18(3), March 1992.

[39] M.D. Weiser, Program slicing, *IEEE Trans. Software Eng.*, SE-10(4), 352–357, April 1988.

[40] E.J. Weyuker, More experience with data flow testing, *IEEE Trans. Software Eng.*, 19(9), 912–919, September 1993.

[41] E.J. Weyuker, The cost of data flow testing: an empirical study, *IEEE Trans. Software Eng.*, 16(2), 121–128, February 1990.

[42] MATLAB, *The MathWorks*, *www.mathworks.com.*

[43] XA, *Sunset Software Technology*, *www.sunsetsoft.com.*

8 Program Slicing

G.B. Mund
Indian Institute of Technology, Kharagpur

D. Goswami
Indian Institute of Technology

Rajib Mall
Indian Institute of Technology

8.1 Introduction

Program slicing is a program analysis technique. It can be used to extract the statements of a program relevant for a given computation. The concept of program slicing was introduced by Weiser in his doctoral work [1]. A program can be sliced with respect to some slicing criterion. In Weiser's terminology, a slicing criterion is a pair $\langle p, V\rangle$, where p is a program point of interest and V is a subset of the program variables. If we attach integer labels to all the statements of a program, then a program point of interest could be an integer i representing the label associated with a statement of the program. A slice of a program P with respect to a slicing criterion $\langle p, V\rangle$ is defined as the set of all statements of P that might affect the values of the variables in V used or defined at the program point p. The program slicing technique introduced by Weiser [1] is now called static backward slicing. It is

0-8493-1240-7/03/$0.00+$1.50

```
1.  read(i);
2.  prod := 1;
3.  sum := 0;
4.  while(i < 10) do
    begin
5.      sum := sum + i;
6.      prod : = prod * i;
7.      i := i + 1;
    end
8.  write(sum);
9.  write(prod);
```

(a)

```
1.  read(i);

3.  sum := 0;
4.  while(i < 10) do
    begin
5.      sum := sum + i;

7.      i := i + 1;
    end
8.  write(sum);
```

(b)

FIGURE 8.1 (a) Example program and (b) its slice with respect to the slicing criterion $\langle 8, sum \rangle$.

static in the sense that the slice is independent of the input values to the program. It is backward because the control flow of the program is considered in reverse while constructing the slice.

Consider the example program given in Figure 8.1(a). Slicing it with respect to the slicing criterion $\langle 8, sum \rangle$ would yield the program slice shown in Figure 8.1(b). A semantic relationship exists between the slice and the original program by the fact that both result in computation of the same value of the variable sum for all possible inputs.

The program slicing technique was originally developed to realize automated static code decomposition tools. The primary objective of those tools was to aid program debugging. From this modest beginning, program slicing techniques have now ramified into a powerful set of tools for use in such diverse applications as program understanding, automated computation of several software engineering metrics, dead code elimination, reverse engineering, parallelization, software portability, reusable component generation, etc. [2–12].

A major aim of any slicing technique is to realize as small a slice with respect to a slicing criterion as possible because smaller slices are found to be more useful in different applications. Much of the literature on program slicing is concerned with improving the algorithms for slicing both in terms of reducing the size of the slice and improving the efficiency of slice computation. These works address computation of more precise dependence information and more accurate slices. Tip [11] and Binkley and Gallagher [12] provide comprehensive surveys of the existing paradigms, applications and algorithms for program slicing.

8.1.1 Static and Dynamic Slicing

Static slicing techniques perform static analysis to derive slices. That is, the source code of the program is analyzed and slices are computed that hold good for all possible input values [13]. A static slice conservatively contains all the statements that may affect the value of a variable at a program point for every possible input. Therefore, a static slice may contain some statements that might not be executed during an actual run of the program.

Dynamic slicing makes use of the information about a particular execution of a program [14]. A dynamic slice with respect to a slicing criterion $\langle p, V \rangle$, for a particular execution, contains only those statements that actually affect the values of the variables in V at the program point p. Therefore, dynamic slices are usually smaller than static slices. A comprehensive survey on the existing dynamic program slicing algorithms is reported in Korel and Rilling [15].

8.1.2 Backward and Forward Slicing

As already discussed, a backward slice contains all parts of the program that might directly or indirectly affect the variables at the statement under consideration. Thus, a static backward slice provides the answer to the question: Which statements affect the slicing criterion?

A forward slice with respect to a slicing criterion $\langle p, V\rangle$ contains all parts of the program that might be affected by the variables in V used or defined at the program point p [16]. A forward slice provides the answer to the question: Which statements will be affected by a slicing criterion?

8.1.3 Organization of the Chapter

The remainder of this chapter is organized as follows. In Section 8.2, we discuss the applications of program slicing. In Section 8.3, we examine some basic concepts, notations and terminologies associated with intermediate representations of sequential programs. Section 8.4 presents some basic slicing algorithms for sequential programs. Section 8.5 deals with intermediate representations and slicing of concurrent and distributed programs. In Section 8.6, we cover parallel slicing of sequential and concurrent programs. In Section 8.7, we discuss intermediate representations and slicing of object-oriented programs. Finally, we present our conclusions in Section 8.8.

8.2 Applications of Program Slicing

The utility and power of program slicing comes from its ability to assist software engineers in many tedious and error prone tasks. Important applications of program slicing include debugging, software maintenance and testing, program integration, functional cohesion metric computation, etc. In the following, we briefly discuss these applications of program slicing.

8.2.1 Debugging

Realization of automated tools to help effective program debugging was the original motivation for the development of the static slicing technique. In his doctoral thesis, Weiser provided experimental evidence that programmers unconsciously use a mental form of slicing during program debugging [1]. Locating a bug can be a difficult task when one is confronted with a large program. In such cases, program slicing is useful because it can enable one to ignore many statements while attempting to localize the bug. If a program computes an erroneous value for a variable x, only those statements in its slice would contain the bug; all statements not in the slice can safely be ignored.

The control and data dependences existing in a program are determined during slice computation. A program slicer integrated into a symbolic debugger can help in visualizing control and data dependences. Variants of the basic program slicing technique have been developed to further assist the programmer during debugging: program dicing [17] identifies statements that are likely to contain bugs by using information that some variables fail some tests whereas others pass all tests at some program point. Consider a slice with respect to an incorrectly computed variable at a particular statement. Now consider a correctly computed variable at some program point. Then the bug is likely to be associated with the slice on the incorrectly computed variable minus the slice on the correctly computed variable. This dicing heuristic can be used iteratively to locate a program bug. Several slices may be combined with each other in different ways: the intersection of two slices contains all statements that lead to an error in both test cases. The union of two slices contains all statements that lead to an error in at least one of the test cases. The symmetric difference of two slices contains all statements that lead to an error in exactly one of the test cases.

Another variant of program slicing is program chopping [18, 19]. It identifies statements that lie between two points a and b in the program and are affected by a change at a. Debugging in such a situation should be focused only on those statements between a and b that transmit the change of a to b.

8.2.2 Software Maintenance and Testing

Software maintainers often have to perform regression testing (i.e., retesting a software product after any modifications are carried out to it to ensure that no new bugs have been introduced [20]). Even after a small change, extensive tests may be necessary, requiring running of a large number of test cases. Suppose a program modification requires only changing a statement that defines a variable x at a program point p. If the forward slice with respect to the slicing criterion $\langle p, x\rangle$ is disjoint from the coverage of a regression test t, then it is not necessary to rerun the test t. Let us consider another situation. Suppose a coverage tool reveals that a use of variable x at some program point p has not been tested. What input data is required to cover p? The answer to this question can be provided by examining the backward slice with respect to the slicing criterion $\langle p, x\rangle$. Work has also been reported concerning testing incrementally through an application of program slicing [21]. These applications are discussed in detail in [22, 23].

Software testers have to locate safety critical code and to ascertain its proper functioning throughout the system. Program slicing techniques can be used to locate all the code parts that influence the values of variables that might be part of a safety critical computation. However, these variables that are part of the safety critical computation have to be determined beforehand by domain experts.

One possibility to assure high quality is to incorporate redundancy into the system. If some output values are critical, then these output values should be computed independently. For doing this, one has to ensure that the computation of these values should not depend on the same internal functions, because an error might manifest in both output values in the same way, thereby causing both the parts to fail. An example of such a technique is functional diversity [24]. In this technique, multiple algorithms are used for the same purpose. Thus, the same critical output values are computed using different internal functions. Program slicing can be used to determine the logical independence of the slices with respect to the output values computing the same result.

8.2.3 Program Integration

Programmers often face the problem of integrating several variants of a base program. To achieve integration, the first step possibly is to look for textual differences between the variants. Semantics-based program integration is a technique that attempts to create an integrated program that incorporates the changed computations of the variants as well as the computations of the base program that are preserved in all variants [25]. Consider a program *Called Base*. Let A and B be two variants of Base created by modifying separate copies of Base. The set of preserved components consists of those components of Base that are neither affected in A nor in B. This set precisely consists of the components having the same slices in Base, A and B. Horwitz et al. presented an algorithm for semantics-based program integration that creates the integrated program by merging the Base program, and its variants A and B [25]. The integrated program is produced through the following steps: (1) building dependence graphs $D1$, $D2$ and $D3$, which represent Base, A, and B respectively; (2) obtaining a dependence graph of the merged program by taking the graph union of the following: symmetric difference of $D1$ and $D2$, symmetric difference of $D1$ and $D3$ and induced graph on the preserved components; (3) testing the merged graph for certain interference criteria; and (4) reconstructing a program from the merged graph.

8.2.4 Functional Cohesion Metric Computation

The cohesion metric measures the relatedness of the different parts of some component. A highly cohesive software module is a module that has one function and is indivisible. For developing an effective functional cohesion metric, Beiman and Ott define data slices that consist of data tokens (instead of statements) [26]. Data tokens may be variables and constant definitions and references. Data slices are computed for each output of a procedure (e.g., output to a file, output parameter and assignment to a global variable). The tokens that are common to more than one data slice are the connections between the slices; they are the glue that bind the slices together. The tokens that are present in every data slice of a function are called *superglue*. Strong functional cohesion can be expressed as the ratio of superglue tokens to the total number of tokens in the slice, whereas weak functional cohesion is the ratio of glue tokens to the total number of tokens. The adhesiveness of a token is another measure expressing how many slices are glued together by that token.

8.2.5 Other Applications of Program Slicing

Program slicing methods have been used in several other applications such as tuning of compilers, compiler optimizations, parallelization of sequential programs, detection of dead code, determination of uninitialized variables, software portability analysis, program understanding, reverse engineering and program specialization and reuse. These applications are discussed in some detail in [2, 4–12, 27].

8.3 Intermediate Program Representation

To compute a slice, it is first required to transform the program code into a suitable intermediate representation. In this section we present a few basic concepts, notations and terminologies associated with intermediate program representation that are used later in this chapter. A common cornerstone for most of the slicing algorithms is that programs are represented by a directed graph, which captures the notion of data dependence and control dependence.

DEFINITION 8.1 (directed graph or graph). *A directed graph G is a pair (N, E) where N is a finite nonempty set of nodes, and $E \subseteq N \times N$ is a set of directed edges between the nodes. Each edge denoted by (x, y) or $x \rightarrow y$ leaves the source node x and enters the sink node y, making x a predecessor of y, and y a successor of x.*

The number of predecessors of a node is its in-degree, and the number of successors of the node is its out-degree. A path from a node x_1 to a node x_k in a graph $G = (N, E)$ is a sequence of nodes $(x_1, x_2, \ldots, x_k)$ such that $(x_i, x_{i+1}) \in E$ for every i, $1 \leq i \leq k - 1$.

DEFINITION 8.2 (flow graph). *A flow graph is a quadruple $(N, E, Start, Stop)$ where (N, E) is a graph, Start $\in N$ is a distinguished node of in-degree 0 called the start node, Stop $\in N$ is a distinguished node of out-degree 0 called the stop node, there is a path from the Start to every other node in the graph, and there is a path from every other node in the graph to the Stop.*

If x and y are two nodes in a flow graph, then x dominates y iff every path from Start to y passes through x; y postdominates x iff every path from x to Stop passes through y. Finding post dominators of a flow graph is equivalent to finding the dominators of the reverse flow graph. In a reverse flow graph, the direction of every edge of the original flow graph is reversed and the *Start* and *Stop* labels are interchanged. We call x the immediate dominator of y, iff x is a dominator of y, $x \neq y$, and no other node z dominates y and is dominated by x. The dominator tree of a directed graph G with

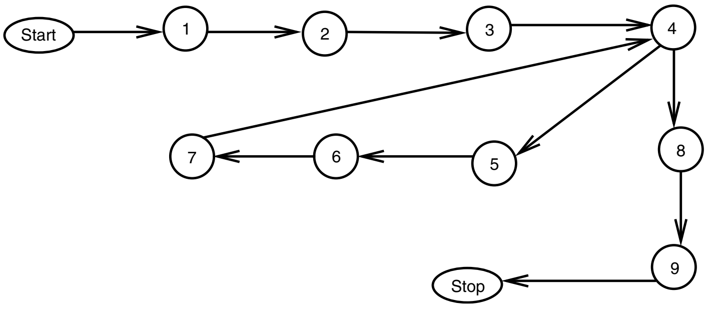


FIGURE 8.2 CFG of the example program given in Figure 8.1(a).

entry node *Start* is the tree that consists of the nodes of G, has the root *Start* and has an edge between nodes x and y, if x immediately dominates y.

8.3.1 Control Flow Graph

Definition 8.3 (control flow graph). *A control flow graph* $(CFG) G$ *of a program* P *is a flow graph* $(N, E, Start, Stop)$*, where each node* $n \in N$ *represents either a statement or a control predicate. An edge* $(m, n) \in E$ *indicates the possible flow of control from the node m to the node n. Nodes Start and Stop are unique nodes representing entry and exit of the program P, respectively.*

Note that the existence of an edge (x, y) in the CFG does not mean that control must transfer from x to y during program execution. Figure 8.2 represents the CFG of the example program given in Figure 8.1(a). CFGs model the branching structure of the program. They can be built while parsing the source code using algorithms that have linear time complexity in the size of the program.

8.3.2 Data Dependence Graph

Data flow describes the flow of the values of variables from the points of their definitions to the points where they are used. In the following, we describe how data flow information can be computed for structured programming languages. A data dependence of a node x to another node y means that the program computation might be changed if the relative order of the nodes x and y are reversed. The direction of the data dependence from a node x to a node y indicates the flow of the value of some variable defined at node x to the node y. The value computed at node y depends on some value of the variable defined at node x that may reach node y. Aho et al. [28] use the term *reaching definition* to express the fact that the value of a variable defined at some node j may be used at another node i. That is, node j is a reaching definition for an execution of node i if node i is data dependent on node j. The precise computation of reaching definitions is one of the goals of data flow analysis. Let m be a node of a CFG G. The sets **def(m)** and **ref(m)** denote the sets of variables defined and referenced at the node m, respectively.

Definition 8.4 (Data Dependence). *Let G be the CFG of a program P. A node n is said to be* ***data dependent*** *on a node m if there exists a variable x of the program P such that the following hold:*

1. $x \in def(m)$.
2. $x \in ref(n)$.
3. *An execution path exists from m to n along which there is no intervening definition of x.*

Definition 8.5 (data dependence graph). *The data dependence graph (DDG) G of a program P is a graph* $G = (N, E)$*, where each node* $n \in N$ *represents either a statement or a predicate of the program. An edge* (m, n) *indicates that n is data dependent on m.*

8.3.3 Control Dependence Graph

In a control dependence graph (CDG), the notion of control dependence is used to represent the relations between program entities arising out of control flow.

DEFINITION 8.6 (control dependence). *Let G be the control flow graph of a program P. Let x and y be nodes in G. Node y is control-dependent on node x if the following hold:*

1. *A directed path Q exists from x to y.*
2. *Node y postdominates every z in Q (excluding x and y).*
3. *Node y does not postdominate x.*

If y is control-dependent on x, then x must have multiple successors. Following one path from x results in execution of y whereas following other paths may result in no execution of y.

DEFINITION 8.7 (control dependence graph). *The CDG over a CFG G is the graph defined over all nodes of G in which a directed edge exists from node x to node y iff y is control dependent on x.*

The CDG compactly encodes the required order of execution of the statements of a program. A node evaluating a condition on which the execution of the other nodes depends has to be executed first. The later nodes are therefore control dependent on the condition node.

8.3.4 Program Dependence Graph

Ottenstein and Ottenstein presented a new mechanism of program representation called *program dependence graph* (PDG) [29]. Unlike flow graphs, an important feature of the PDG is that it explicitly represents both control and data dependences in a single program representation. Because program slicing requires both kinds of dependences, the PDG has been adopted as an appropriate representation for use with slicing algorithms. A PDG models a program as a graph in which the nodes represent either statements or predicates, and the edges represent data or control dependences.

DEFINITION 8.8 (program dependence graph). *The PDG of a program P is the union of a pair of graphs : the DDG of P and the CDG of P.*

Note that the DDG and CDG are subgraphs of a PDG. Figure 8.3 represents the PDG of the example program given in Figure 8.1(a).

8.3.5 System Dependence Graph

The PDG of a program combines the control dependences and data dependences into a common framework. The PDG has been found to be suitable for intraprocedural slicing. However, it cannot handle procedure calls. Horwitz et al. enhanced the PDG representation to facilitate interprocedural slicing [30]. They introduced the system dependence graph (SDG) representation that models the main program together with all nonnested procedures. This graph is very similar to the PDG. Indeed, a PDG of the main program is a subgraph of the SDG. In other words, for a program without procedure calls, the PDG and the SDG are identical. The technique for constructing an SDG consists of first constructing a PDG for every procedure, including the main procedure, and then adding auxiliary dependence edges that link the various subgraphs together. This results in a program representation that includes the information necessary for slicing across procedure boundaries.

An SDG includes several types of nodes to model procedure calls and parameter passing:

- *Call-site nodes.* These represent the procedure call statements in a program.
- *Actual-in* and *actual-out nodes.* These represent the input and output parameters at the call sites. They are control dependent on the call-site nodes.

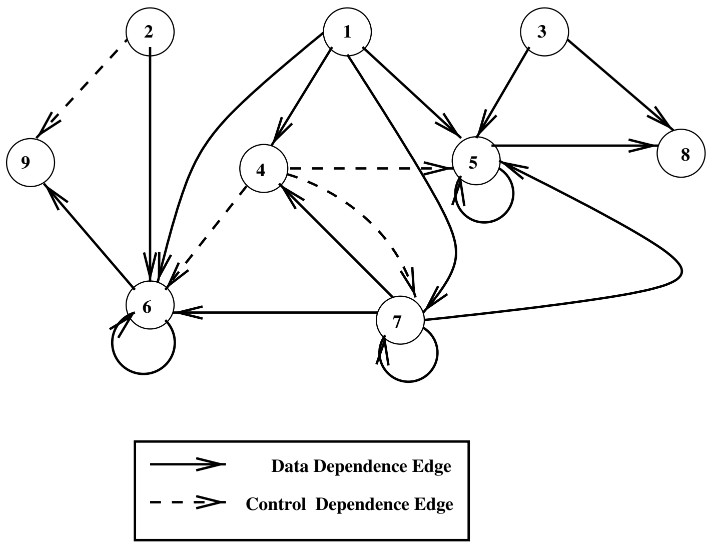


FIGURE 8.3 PDG of the example program given in Figure 8.1(a).

```
main( )                 add(a, b)          inc(z)
begin                   begin              begin
s=0;                       a=a+b;             return add(z, 1)
i=1;                       return a        end
while (i < 10) do       end
begin
   add(s, i);
   inc(i)
end
write(s)
end
```

FIGURE 8.4 Example program consisting of a main program and two procedures.

- *Formal-in* and *formal-out nodes*. These represent the input and output parameters at the called procedure. They are control dependent on the procedure's entry node.

Control dependence edges and data dependence edges are used to link the individual PDGs in an SDG. The additional edges to link the PDGs together are as follows:

- *Call edges*. These link the call-site nodes with the procedure entry nodes.
- *Parameter-in edges*. These link the actual-in nodes with the formal-in nodes.
- *Parameter-out edges*. These link the formal-out nodes with the actual-out nodes.

Finally, summary edges are used to represent the transitive dependences that arise due to calls. A summary edge is added from an actual-in node A to an actual-out node B, if a path of control, data and summary edges exist in the called procedure from the corresponding formal-in node A' to the formal-out node B'. Figure 8.5 represents the SDG of the example program shown in Figure 8.4.

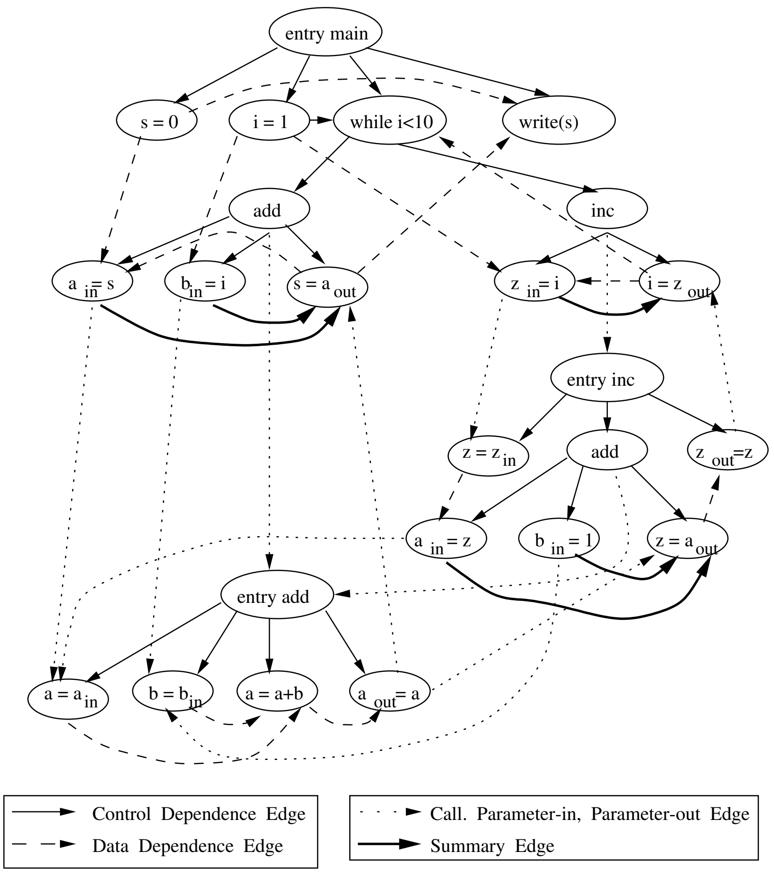


FIGURE 8.5 SDG of the example program shown in Figure 8.4.

8.4 Basic Slicing Algorithms: An Overview

This section presents an overview of the basic program slicing techniques and includes a brief history of their development. We start with the original approach of Weiser [1] where slicing is considered as a data flow analysis problem, and then examine the slicing techniques where slicing is seen as a graph reachability problem.

8.4.1 Slicing Using Data Flow Analysis

8.4.1.1 Weiser's Algorithm

Weiser used a CFG as an intermediate representation for his slicing algorithm [1]. Let v be a variable and n be a statement (node) of a program P, and S be the slice with respect to the slicing criterion $\langle n, v\rangle$. Consider a node $m \in S$. Weiser defined the set of *relevant variables* for the node m, $relevant(m)$, as the set of variables of the program P whose values (transitively) affect the computation of the value of the variable v at the node n. Consider the example program shown in Figure 8.1(a) and the slice with respect to the slicing criterion $\langle 9, prod\rangle$. For this example, $relevant(8) = \{prod\}$, $relevant(7) = \{prod\}$, and $relevant(6) = \{prod, i\}$. Computing a slice from a CFG requires computation of the data flow information about the set of relevant variables at each node. That is, slices can be computed by solving a set of data and control flow equations derived directly from the CFG of the program undergoing slicing.

In Weiser's approach, every slice is computed from scratch. That is, no information obtained during any previous computation of slices is used. This is a serious disadvantage of his algorithm. It has been shown that computation of static slices using his algorithm requires $O(n^2e)$ time, where n is the number of nodes and e is the number of edges in the CFG [1].

Weiser's algorithm [1] for computing static slices cannot handle programs with multiple procedures. Later in [13], Weiser presented algorithms for interprocedural slicing.

8.4.1.1.1 ***Dynamic Slicing.***

Consider the example program given in Figure 8.6. The static slice of the program with respect to the criterion $\langle 11, z \rangle$ includes the whole program. Suppose the program is executed and the value entered for the variable i was 3. Now, if the value of z printed at the end of the program is not as expected, then we can infer that the program contains a bug. To locate the bug a dynamic slice can be constructed as shown in Figure 8.6(b). The dynamic slice only identifies those statements that contribute to the value of the variable z when the input $i = 3$ is supplied to the program. Locating the bug using the dynamic slice is thus easier than examining the original program or the corresponding static slice because the number of statements included in the dynamic slice is normally much less. A dynamic slice is said to be precise if it includes only those statements that actually affect the value of a variable at a program point for the given execution.

Korel and Laski extended Weiser's CFG based static slicing algorithm to compute dynamic slice [14]. They computed dynamic slices by solving the associated data flow equations. However, their dynamic slice may not be optimal. They require that if any one occurrence of a statement in the execution trace is included in the slice, then all other occurrences of that statement should be automatically included in the slice, even when the value of the variable in the slicing criterion under consideration is not affected by other occurrences. Their method needs $O(N)$ space to store the execution history, and $O(N^2)$ space to store the dynamic flow data, where N is the number of statements executed during the run of the program. Note that for programs containing loops, N may be unbounded. This is a major shortcoming of their method.

8.4.2 Slicing Using Graph-Reachability Analysis

Ottenstein and Ottenstein [29] defined slicing as a reachability problem in the dependence graph representation of a program. A directed graph is used as an intermediate representation of the program. This directed graph models the control or data dependences among the program entities. Slices can be computed by traversing along the dependence edges of this intermediate representation. An important advantage of this approach is that data flow analysis has to be performed only once, and that the information can be used for computing all slices.

8.4.2.1 Intraprocedural Slicing: Program Dependence Graph

Ottenstein and Ottenstein introduced PDG as an intermediate program representation [29]. We have already discussed the definitions of these types of graphs in Section 8.3.4 . They demonstrated how the PDG could be used as the basis of a new slicing algorithm. Their algorithm produced smaller slices than Weiser's algorithm. This method differed from Weiser's in an important way: it used a single reachability pass of a PDG compared with Weiser's incremental flow analysis. Ottenstein and Ottenstein presented a linear time solution for intraprocedural static slicing in terms of graph reachability in the PDG [29]. The construction of the PDG of a program requires $O(n^2)$ time, where n is the number of statements in the program. Once the PDG is constructed, the slice with respect to a slicing criterion can be computed in $O(n + e)$ time, where n is the number of nodes and e is the

```
1.  read(i);
2.  n = 3;
3.  z = 1;
4.  while(i < n) do
     begin
5.      read(x);
6.      if(x < 0) then
7.          y = f1(x);
        else
8.          y = f2(x);
9.      z = f3(y);
10.     i = i + 1;
     end;
11. write(z);
```

(a)

```
3.  z = 1;
```

(b)

FIGURE 8.6 (a) Example program and (b) its dynamic slice with respect to the slicing criterion $\langle 11, z \rangle$ for the input value $i = 3$.

number of edges in the PDG. The process of building the PDG of a program involves computation and storage of most of the information needed for generating slices of the program.

8.4.2.1.1 *Dynamic Slicing Using PDG.*

Agrawal and Horgan [31] were the first to present algorithms for finding dynamic program slices using PDG as the intermediate program representation. In their approach, they construct a DDG of the program at runtime using its PDG. Construction of DDG involves creating a new node for each occurrence of a statement in the execution history, along with its associated control and dependence edges [31]. The disadvantage of using the DDG is that the number of nodes in a DDG is equal to the number of executed statements (length of execution), which may be unbounded for programs having loops.

Agrawal and Horgan [31] proposed to reduce the number of nodes in the DDG by merging nodes for which the transitive dependences map to the same set of statements. In other words, a new node is introduced only if it can create a new dynamic slice. This check involves some runtime overhead. The resulting reduced graph is called the *reduced dynamic dependence graph* (RDDG).The size of the RDDG is proportional to the number of dynamic slices that can arise during execution of the program. Note that in the worst case, the number of dynamic slices of a program having n statements is $O(2^n)$ [11].

The example given in Figure 8.7 shows a program having $O(2^n)$ different dynamic slices. The program reads n distinct numbers $x_1, x_2, \ldots, x_n$. Then, for every possible subset $S \subseteq \{x_1, x_2, \ldots, x_n\}$, it finds the sum of all the elements of the subset S. Note that in each iteration of the outer loop, the slice with respect to write(y) contains all the statements read(x_i) for which $x_i \in S$. Because a set with n elements has 2^n subsets, the example of Figure 8.7 has $O(2^n)$ different dynamic slices. Thus, the worst-case space complexity of the RDDG-based algorithm of Agrawal and Horgan [31] is exponential in the number of statements of the program.

Mund et al. [32] have proposed an efficient intraprocedural dynamic slicing algorithm. They used the PDG as an intermediate program representation, and modified it by introducing the concepts of

```
/* This program reads n numbers and prints the sum of various */
/* subsets of the set of the read numbers.*/
                integer y, i, x1,..., xn;
                moreSubsets, finished : boolean;
                read(x1);
                 ......
                read(xn);
                moreSubsets=true;
                while (moreSubsets) do
                      finished=false;
                      y=0;
                      while not(finished) do   /*  this loop finds the  */
                           read(i);            /*  sum of the elements  */
                           case(i) of          /*  of a subset of { x1,..., xn}.  */
                             1:  y=y+x1;

                                 .......

                             n: y=y+xn;
                           endcase;
                           read(finished);
                      endwhile;
                      write(y);                  /* Output sum of the current subset */
                      read(moreSubsets);         /* Find out whether there are any more subsets . */
                endwhile.
```

FIGURE 8.7 Program having exponential (in the number of statements) number of dynamic slices.

a stable edge and an unstable edge. Let S be a statement in a program P. An outgoing dependence edge (S, S_i) in the PDG G_P of P is said to be an unstable edge if there exists an outgoing dependence edge (S, S_j) with $S_i \neq S_j$ such that the statements S_i and S_j both define the same variable used at S.

An edge that is not unstable is said to be an stable edge. If (x, y) is an unstable edge, then it does not mean that node x can have dependence on node y in all executions of the program. That is, in some execution of the program node x may have dependence on node y, and in some other execution of the program node x may not have dependence on node y. Figure 8.8 represents the modified program dependence graph (MPDG) of the example program given in Figure 8.6(a). The MPDG of a program does not distinguish data and control dependence edges. It contains two types of edges: stable edges and unstable edges. In the MPDG of Figure 8.8, the edges (4,1), (4,10), (9,7), (9,8), (10,1), (10,10), (11, 3) and (11, 9) are unstable edges, with all other edges stable. The algorithm of Mund et al. [32] is based on marking and unmarking the unstable edges as and when the dependences arise and cease at runtime. Their algorithm does not need to create any new node at runtime. They have shown that their algorithm always finds a precise dynamic slice with respect to a given slicing criterion. Further, they have shown that their algorithm is more time efficient than the existing ones, and that the worst-case space complexity of their algorithm is $O(n^2)$, where n is the number of statements in the program.

8.4.2.2 Interprocedural Slicing: System Dependence Graph

As described earlier, the notion of PDG was extended by Horwitz et al. into an SDG to represent multiprocedure programs [30]. Interprocedural slicing can be implemented as a reachability problem over the SDG. Horwitz et al. developed a two-phase algorithm that computes precise interprocedural

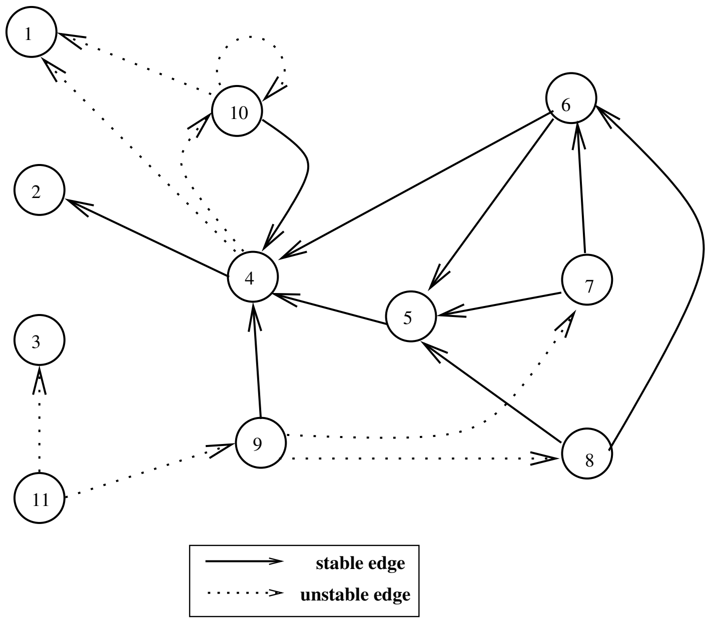


FIGURE 8.8 MPDG of the example program given in Figure 8.6(a). An edge (x, y) represents data or control dependence of the node x on the node y.

slices [30]. To compute a slice with respect to a node n in a procedure P requires two phases of computations that perform the following:

- In the first phase, all edges except the parameter-out edges are followed backward starting with the node n in procedure P. All nodes that reach n and are in P itself or in procedures that (transitively) call P are marked. That is, the traversal ascends from procedure P upward to the procedures that called P. Because parameter-out edges are not followed, phase 1 does not descend into procedures called by P. The effects of such procedures are not ignored. Summary edges from actual-in nodes to actual-out nodes cause nodes to be included in the slice that would only be reached through the procedure call, though the graph traversal does not actually descend into the called procedure. The marked nodes represent all nodes that are part of the calling context of P and may influence n.
- In the second phase, all edges except parameter-in and call edges are followed backward starting from all nodes that have been marked during phase 1. Because parameter-in edges and call edges are not followed, the traversal does not ascend into calling procedures. Again, the summary edges simulate the effects of the calling procedures. The marked nodes represent all nodes in the called procedures that induce summary edges.

8.5 Slicing of Concurrent and Distributed Programs

In this section, we first discuss some basic issues associated with concurrent programming. Later, we discuss how these issues have been addressed in computation of slices of concurrent programs.

The basic unit of concurrent programming is the process (also called *task* in the literature). A process is an execution of a program or a section of a program. Multiple processes can be executing the same program (or a section of the program) simultaneously. A set of processes can execute on one or more processors. In the limiting case of a single processor, all processes are interleaved or time shared on this processor. *Concurrent program* is a generic term that is used to describe any program involving potential parallel behavior. Parallel and distributed programs are subclasses of concurrent programs that are designed for execution in specific parallel processing environments.

8.5.1 Program Properties

8.5.1.1 Nondeterminism

A sequential program imposes a total ordering on the actions it performs. In a concurrent program, an uncertainty exists over the precise order of occurrence of some events. This property of a concurrent program is referred to as *nondeterminism*. A consequence of nondeterminism is that when a concurrent program is executed repeatedly, it may take different execution paths even when operating on the same input data.

8.5.1.2 Process Interaction

A concurrent program normally involves process interaction. This occurs for two main reasons:

- Processes compete for exclusive access to shared resources, such as physical devices or data, and therefore need to coordinate access to the resource.
- Processes communicate to exchange data.

In both the preceding cases, it is necessary for the processes concerned to synchronize their execution, either to avoid conflict, when acquiring resources, or to make contact, when exchanging data. Processes can interact in one of two ways: through shared variables, or by message passing. Process interaction may be explicit within a program description or may occur implicitly when the program is executed.

A process wishing to use a shared resource must first acquire the resource, that is, obtain permission to access it. When the resource is no longer required, it is released. If a process is unable to acquire a resource, its execution is usually suspended until that resource is available. Resources should be administered so that no process is delayed unduly.

8.5.1.3 A Coding View

The main concerns in the representation of concurrent programs are:

- Representation of processes
- Representation of process interactions

Concurrent behavior may be expressed directly in a programming notation or implemented by system calls. In a programming notation, a process is usually described in a program block and process instances are created through declaration or invocation references to that block. Process interaction is achieved via shared variables or by message passing from one process to another.

8.5.1.4 Interaction via Shared Variables

A commonly used mechanism for enforcing mutual exclusion is through use of semaphores. Entry to and exit from a critical region is controlled by using P and V operations, respectively. This notation was proposed by Dijkstra [33], and the operations can be read as “wait if necessary” and “signal” (the letters actually represent Dutch words meaning pass and release). Some semaphores are defined to give access to competing processes based on their arrival order. The original definition, however, does not stipulate an order. The less strict definition gives greater flexibility to the implementor but forces the program designer to find other means of managing queues of waiting processes.

8.5.1.5 Interaction by Message Passing

Process interaction through message passing is very popular. This model has been adopted by the major concurrent programming languages [34]. It is also a model amenable to implementation in a distributed environment. Within the general scheme of message passing, there are two alternatives:

- *Synchronous*. The sending process is blocked until the receiving process has accepted the message (by implicit or by some explicit operation).
- *Asynchronous*. The sender does not wait for the message to be received but continues immediately. This is sometimes called a *nonblocking*, or *no-wait send*.

Synchronous message passing, by definition, involves a synchronization as well as a communication operation. Because the sender process is blocked while awaiting receipt of the message, there can be at most one spending message from a given sender to a given receiver, with no ordering relation assumed between messages sent by different processes. The buffering problem is simple because the number of sending messages is bounded.

In asynchronous message passing, the pending messages are buffered transparently, leading to potential unreliability in case of a full buffer. For most applications, synchronous message passing is thought to be the easier method to understand and use, and is more reliable as well. Asynchronous message passing allows a higher degree of concurrency.

8.5.2 Concurrency at the Operating System Level

In this section we confine our attention to the discussion on the UNIX operating system. UNIX [35, 36] is not a single operating system but an entire family of operating systems. The discussion here is based chiefly on the POSIX standard, which describes a common, portable, UNIX programmer's interface.

UNIX uses a pair of system calls, fork and exec, for process creation and activation. The fork call creates a copy of the forking process with its own address space. The exec call is invoked by either the original or a copied process to replace its own virtual memory space with the new program, which is loaded into memory, destroying the memory image of the calling process. The parent process of a process terminating by using the exit system call can wait on the termination event of its child process by using the wait system call. Process synchronization is implemented using semaphores. Interprocess communication is achieved through shared memory and message-passing mechanisms [37]. A shared memory segment is created using the *shmget* function. It returns an identifier to the segment. The system call *shmat* is used to map a segment to the address space of a particular process. Message queues are created by using *msgget* function. Messages are sent by using the *msgsnd* function and these messages get stored in the message queue. The *msgrcv* function is used by a process to receive a message addressed to it from the message queue.

8.5.3 Slicing of Concurrent Programs

Research in slicing of concurrent programs is scarcely reported in the literature. In the following, we review the reported work in static and dynamic slicing of concurrent programs.

8.5.3.1 Static Slicing

Cheng [38] generalized the notion of CFG and a PDG to a nondeterministic parallel control flow net and a program dependence net (PDN), respectively. In addition to edges for data dependence and control dependence, PDN may also contain edges for selection dependences, synchronization dependences and communication dependences. Selection dependence is similar to control dependence but involves nondeterministic selection statements, such as the ALT statement of Occam-2.

Synchronization dependence reflects the fact that the start or termination of the execution of a statement depends on the start or termination of the execution of another statement. Communication dependence corresponds to a situation where a value computed at one point in the program influences the value computed at another point through interprocess communication. Static slices are computed by solving for the reachability problem in a PDN. However, Cheng did not precisely define the semantics of synchronization and communication dependences, or state or prove any property of the slices computed by his algorithm [38].

Goswami et al. [39] presented an algorithm for computing static slices of concurrent programs in a UNIX process environment. They introduced the concept of a concurrent program dependence graph (CPDG), and constructed the graph representation of a concurrent program through three hierarchical levels: process graph, concurrency graph and CPDG. A process graph captures the basic process structure of a concurrent program, and represents process creation, termination and joining of processes. A process node consists of a sequence of statements of a concurrent program that would be executed by a process.

Krinke [40] proposed a method for slicing threaded programs. Krinke's work extends the structures of CFG and PDG for threaded programs with interference. She defines *interference* as data flow that is introduced through use of variables common to parallel executing statements. In [40], Krinke proposed a slicing algorithm to compute slices from the new constructs for threaded programs called as *threaded*-PDG and *threaded*-CFG. Nanda and Ramesh [41] later pointed out some inaccuracies in the slicing algorithm of Krinke and proposed some improvements for it. Their algorithm has a worst-case complexity of $O(N^t)$, where N is the number of nodes in the graph and t is the number of threads in the program. They also proposed three optimizations to reduce this exponential complexity.

A process graph captures only the basic process structure of a program. This has been extended to capture other UNIX programming mechanisms such as interprocess communication and synchronization. A concurrency graph is a refinement of a process graph where the process nodes of the process graph containing message passing statements are split up into three different kinds of nodes, namely, send node, receive node and statement node. The significance of these nodes and the construction procedure of these nodes are explained in the following:

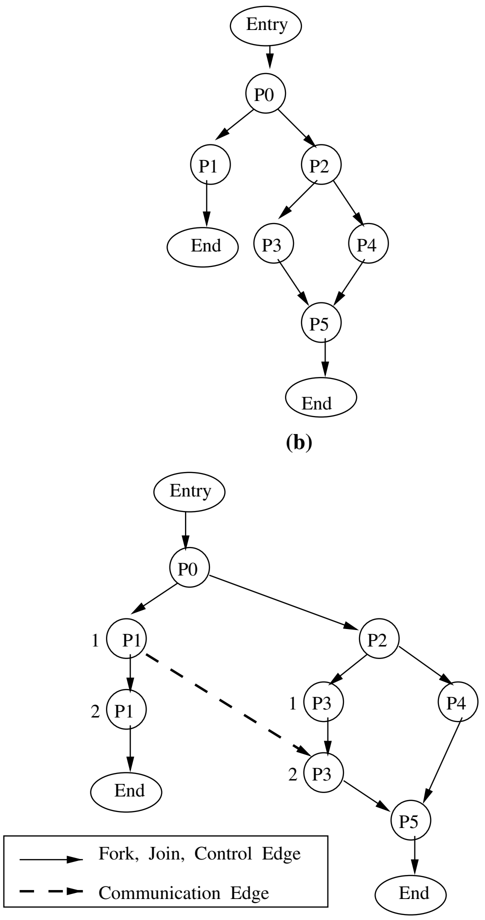
- *Send node*. A send node consists of a sequence of statements that ends with a *msgsend* statement.
- *Receive node*. A receive node consists of a sequence of statements that begins with a *msgrecv* statement.
- *Statement node*. A statement node consists of a sequence of statements without any message passing statement.

Each node of the concurrency graph is called a *concurrent component*. A concurrency graph captures the dependencies among different components arising due to message-passing communications among them. However, components may also interact through other forms of communication such as shared variables. Access to shared variables may either be unsynchronized or synchronized using semaphores. Further, to compute a slice, in addition to representing concurrency and interprocess communication aspects, one needs to represent all traditional (sequential) program instructions. To achieve this, they extended the concurrency graph to construct a third level graph called CPDG. Consider the example program given in Figure 8.9(a). Its process graph, concurrency graph and CPDG are shown in Figures 8.9(b), 8.9(c) and 8.10, respectively. Once the CPDG is constructed, slices can be computed through simple graph reachability analysis.

Goswami et al. [39] implemented a static slicing tool that supports an option to view slices of programs at different levels, that is, process level, concurrent component level or code level. They reported on the basis of implementation experience that their approach of hierarchical presentation of the slicing information helps the users get a better understanding of the behaviors of concurrent programs.

```
    main()
    { /* P0 */
    int i, j, x, n;
    x = shmat(...);
1.  x = 0;
2.  scanf("%d", &n);
3.  i = 1;
4.  j = 0;
5.  if (fork() == 0)
    { /* P1 */
6.      x = x + n;
7.      j = i + x;
8.      msgsend(m1, j);
9.      for(i=1; i<n; i++)
10          j++;
    }
    else { /* P2 */
11.     if (fork() == 0)
        { /* P3 */
12.         i = i +1;
13.         msgrecv(m1, j);
14.         i = i +j;
15.         x--;
        }
        else { /* P4 */
16.         if (n>0)
17.             x--;
            else
18.             x++;
19.         n++;
20.         wait(0);
        }
        /* P5 */
21.     printf("%d", n);
    }
    }
```

(a)



(c)

FIGURE 8.9 (a) Example program; (b) process graph and (c) concurrency graph.

8.5.3.2 Dynamic Slicing

Korel and Ferguson extended the dynamic slicing method of [14, 42] to distributed programs with Ada-type rendezvous communication [43]. For a distributed program, the execution history is formalized as a distributed program path which, for each task, comprises (1) the sequences of statements (trajectory) executed by it, and (2) a sequence of triples (A, C, B) identifying each rendezvous in which the task is involved.

A dynamic slicing criterion of a distributed program specifies: (1) the inputs to each task, (2) a distributed program path P, (3) a task W, (4) a statement occurrence q in the trajectory of w and (5) a variable v. A dynamic slice with respect to such a criterion is an executable projection of the program that is obtained by deleting statements from it. However, the computed slice is only guaranteed to preserve the behavior of the program if the rendezvous in the slice occurs in the same relative order as in the program.

Duesterwald, Gupta and Soffa present a dependence graph based algorithm for computing dynamic slices of distributed programs [44]. They introduce a DDG for representing distributed programs. A DDG contains a single vertex for each statement and predicate in a program. Control dependences between statements are determined statically, prior to execution. Edges for data and communication

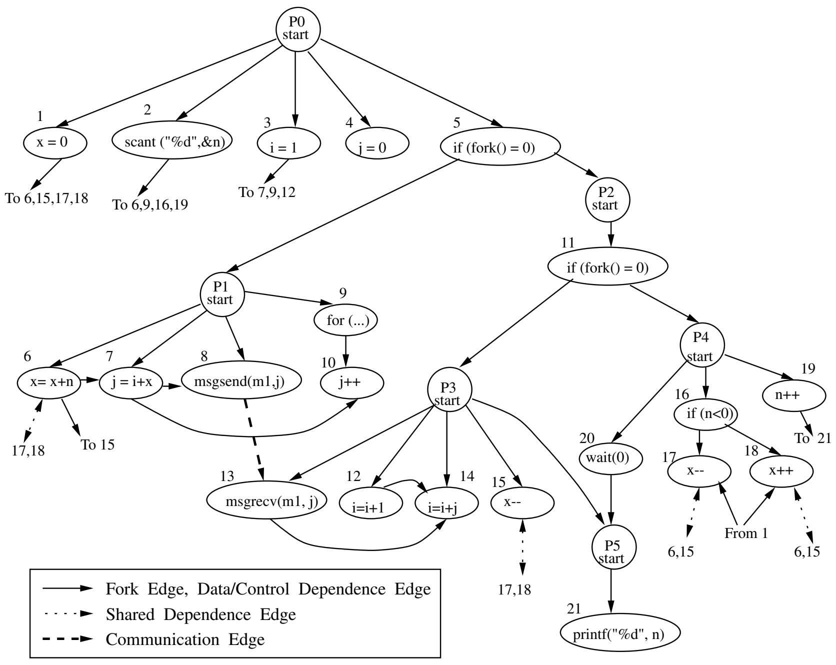


FIGURE 8.10 CPDG for the example program shown in Figure 8.9(a).

dependences are added to the graph at runtime. Slices are computed in the usual way by determining the set of DDG vertices for which the vertices specified in the criterion can be reached. Both the construction of the DDG and the computation of slices are performed in a distributed manner. Thus, a separate DDG construction process and slicing process are assigned to each process P_i in the program. The different processes communicate when a send or receive statement is encountered. However, due to the fact that a single vertex is used for all occurrences of a statement in the execution history, inaccurate slices may be computed in the presence of loops.

Cheng presents an alternative graph-based algorithm for computing dynamic slices of distributed and concurrent programs [38]. Cheng's algorithm is basically a generalization of the initial approach proposed by Agrawal and Horgan in [31]: the PDN vertices corresponding to executed statements are marked, and the static slicing algorithm is applied to the PDN subgraph induced by the marked vertices. This, however, yields inaccurate slices in presence of loops.

In [45], Goswami and Mall extended their static slicing framework [39] to compute dynamic slices of concurrent programs. They introduced the notion of the dynamic program dependence graph (DPDG) to represent various intra- and interprocess dependences of concurrent programs. They constructed the DPDG of a concurrent program through three hierarchical stages. At compile time, a dynamic process graph and a static program dependence graph (SPDG) are constructed. The dynamic process graph is the most abstract representation of a concurrent program. It captures the basic information about processes that are obtained through a static analysis of the program code. The SPDG of a concurrent program represents the static part of data and control dependences of the program. Trace files are generated at runtime to record the information regarding the relevant events that occur during the execution of concurrent programs. By using the information stored in the trace files, the dynamic process graph is refined to realize a dynamic concurrency graph. The SPDG, the information stored in the trace files and the dynamic concurrency graph are then used to construct the DPDG. The DPDG of a concurrent program represents dynamic information concerning fork and join, semaphore and shared dependences and communication dependences due to message passing in addition to data and control dependences. After construction of the DPDG of a concurrent program, dynamic slices can be computed using some simple graph-reachability algorithm.

The dynamic slicing algorithm of Goswami and Mall [45] can handle both shared memory and message-passing constructs. They have shown that their dynamic slicing algorithm computes more precise dynamic slices than the dynamic slicing algorithms of Cheng [38] and Duesterwald, Gupta and Soffa [44].

8.6 Parallelization of Slicing

Parallel algorithms have the potential to be faster than their sequential counterparts because the computation work can be shared by many computing agents all executing at the same time. Also, for large programs, sequential algorithms become very slow. Slicing algorithms for concurrent programs are highly compute-intensive because the graphs required for intermediate representations of the programs often become very large for practical problems. Therefore, parallelization of slicing algorithms seems to be an attractive option to improve efficiency. In the following, we review the research results in parallelization of slicing algorithms for sequential and concurrent programs.

8.6.1 Parallel Slicing of Sequential Programs

In [46], Harman et al. presented a parallel-slicing algorithm to compute intraprocedural slices for sequential programs. In their method, a process network is constructed from the program to be sliced. A process network is a network of concurrent processes. It is represented as a directed graph in which nodes represent processes and edges represent communication channels among processes.

The process network is constructed using the CFG of the program. The reverse control flow graph (RCFG) is constructed by reversing the direction of every edge in the CFG. The topology of the process network is obtained from the RCFG, with one process for each of its nodes and with communication channels corresponding to its edges. The edges entering a node i represent input to process i, and the edges leaving node i represent outputs from process i.

To compute a slice for the slicing criterion $\langle n, V\rangle$, where V is a set of variables of the program and n is a node of the CFG of the program, network communication is initiated by outputing the message V from the process n of the process network. Messages then are generated and passed around the network until it eventually stabilizes, that is, when no new message arrives from any node. The algorithm computes the slice of a program by including the set of nodes, with identifiers that are input to the entry node of the process network. The parallel-slicing algorithm has been shown to be correct and finitely terminating [46]. Implementation details of the algorithm have not been reported in [46].

8.6.2 Parallel Slicing of Concurrent Programs

In [47], Haldar et al. extended the parallel static slicing algorithm of Harman et al. [46] for sequential programs to concurrent programs. They introduced the concept of concurrent control flow graph (CCFG). The CCFG of a concurrent program consists of the CFGs of all the processes, with nodes and edges added to represent interprocess communications. Note that fork edges in a process graph represent flow of control among processes in a concurrent program. When a process forks, it creates a child process and executes concurrently with the child. Thus, a fork edge in a process can be used to represent parallel flow of control. Process graph and concurrency graphs are constructed as already discussed in the context of static slicing of concurrent programs. For every node x of the process graph, a CFG is constructed from the process represented by node x. The CCFG is then constructed by interconnecting the individual CFGs.

The algorithm of Haldar et al. [47] first constructs the process network of a given concurrent program. The topology of the process network is given by the reverse concurrent control flow graph (RCCFG). The RCCFG is constructed from the CCFG by reversing the direction of all the edges. Every node of the RCCFG represents a process and the edges represent communication channels among processes. Consider a slicing criterion $\langle P, s, V \rangle$ of a concurrent program, where P is a process, s is a statement in the process P and V is a set of variables of the program. To compute a slice with respect to this criterion, the process network is first initiated. Let n be the node in the CCFG corresponding to the statement s in the process P, and m be the process in the process network corresponding to the CCFG node m. The process network is initiated by transmitting the message $\{m, V\}$ on all output channels of m. Each process in the process network repeatedly sends and receives messages until the network stabilizes. The network stabilizes when no messages are generated in the whole network. The set of all node identifiers that reach the *entry* node gives the required static slice. Haldar et al. [47] proved that their algorithm is correct and finite terminating. The steps for computation of slices are summarized as follows:

1. Construct the hierarchical CCFG for the concurrent program.
2. Reverse the CCFG.
3. Compile the RCCFG into a process network.
4. Initiate network communication by outputting the message $\{s, v\}$ from the process in the process network representing statement s in the process P, where $\langle P, s, v \rangle$ is the slicing criterion.
5. Continue the process of message generation until no new messages are generated in the network.
6. Add to the slice all those statements with node identifiers that have reached the *entry* node of the CCFG.

8.6.2.1 Implementation Results

Haldar et al. [47] implemented their parallel algorithm for computing dynamic slices of concurrent programs in Digital–UNIX environment. They considered a subset of C language with UNIX primitives for process creation and interprocess communications. Standard UNIX tools *Lex* and *Yacc* have been used for lexical analysis and parsing of the source code. The information stored with each node of RCCFG representing a statement s are node *id*, statement type, $ref(s)$, $def(s)$, $C(s)$, input channels (numbers and types) and output channels (numbers and types).

A major aim of their implementation was to investigate the achieved speedup in computing slices. They examined their algorithm with several input concurrent programs. They have reported that the lengths of the input programs were in between 30 to 100 lines. They considered a subset of C programming language in writing these programs.

They have reported the following encouraging results of the implementation. The speedup achieved for different programs — in a 2-processor environment is between 1.13 and 1.56, in 3-processor environment is between 1.175 and 1.81 and in 4-processor environment is between 1.255 and 2.08. For the same number of processors used, speedup varies for different program samples. It is seen that speedup is more for larger programs. This may be due to the fact that the number of nodes in the process network for larger programs is higher compared with smaller programs, leading to higher utilization of processors. Their implementation supported up to a 4-processor environment and considered small size input programs (up to 100 lines).

8.7 Slicing of Object-Oriented Programs

Object-oriented programming languages have become very popular during the last decade. The concepts of classes, inheritance, polymorphism and dynamic binding are the basic strengths of object-oriented programming languages. On the other hand, these concepts raise new challenges for program slicing. Intermediate representations for object-oriented programs need to model classes, objects, inheritance, scoping, persistence, polymorphism and dynamic binding effectively. In the literature, research efforts to slice object-oriented programs are scarely reported. In the following, we briefly review the reported work on static and dynamic slicing of object-oriented programs.

8.7.1 Static Slicing of Object-Oriented Programs

Several researchers have extended the concepts of intermediate procedural program representation to intermediate object-oriented program representation. Kung et al. [48–50] presented a representation for object-oriented software. Their model consists of an object relation diagram and a block branch diagram. The object relation diagram of an object-oriented program provides static structural information on the relationships existing between objects. It models the relationship that exists between classes such as inheritance, aggregation and association. The block branch diagram of an object-oriented program contains the CFG of each of the class methods, and presents a static implementation view of the program. Harrold and Rothermel [51] presented the concept of call graph. A call graph provides a static view of the relationship between object classes. A call graph is an interprocedural program representation in which nodes represent individual methods and edges represent call sites. However, a call graph does not represent important object-oriented concepts such as inheritance, polymorphism and dynamic binding.

Krishnaswamy [52] introduced the concept of the object-oriented program dependence graph (OPDG). The OPDG of an object-oriented program represents control flow, data dependences and control dependences. The OPDG representation of an object-oriented program is constructed in three layers, namely: class hierarchy subgraph (CHS), control dependence subgraph (CDS) and data dependence subgraph (DDS). The CHS represents inheritance relationship between classes, and the composition of methods into a class. A CHS contains a single class header node and a method header node for each method that is defined in the class. Inheritance relationships are represented by edges connecting class headers. Every method header is connected to the class header by a membership edge. Subclass representations do not repeat representations of methods that are already defined in the superclasses. Inheritance edges of a CHS connect the class header node of a derived class to the class header nodes of its superclasses. Inherited membership edges connect the class header node of the derived class to the method header nodes of the methods that it inherits. A CDS represents the static control dependence relationships that exist within and among the different methods of a class. The DDS represents the data dependence relationship among the statements and predicates of the program. The OPDG of an object-oriented program is the union of the three subgraphs: CHS, CDS and DDS. Slices can be computed using OPDG as a graph-reachability problem.

The OPDG of an object-oriented program is constructed as the classes are compiled and hence it captures the complete class representations. The main advantage of OPDG representation over other representations is that the representation has to be generated only once during the entire life of the class. It does not need to be changed as long as the class definition remains unchanged. Figure 8.11(b) represents the CHS of the example program of Figure 8.11(a).

In [53], Larsen and Harrold extended the concept of SDG to represent some of the features of object-oriented programs. They introduced the notions of the CDG, class call graph, CCFG and interclass dependence graph. A CDG captures the control and data dependence among statements in a single class hierarchy. It connects individual PDGs for methods that are members of the class. A CDG uses a polymorphic choice node to represent the dynamic choice among the possible destinations. Such a node has all edges incident to subgraphs representing calls to each possible destination. A class call graph captures the calling relationship among methods in a class hierarchy. As the CHS, it contains class header nodes, method header nodes, virtual method header nodes, membership edges, inheritance edges and inherited membership

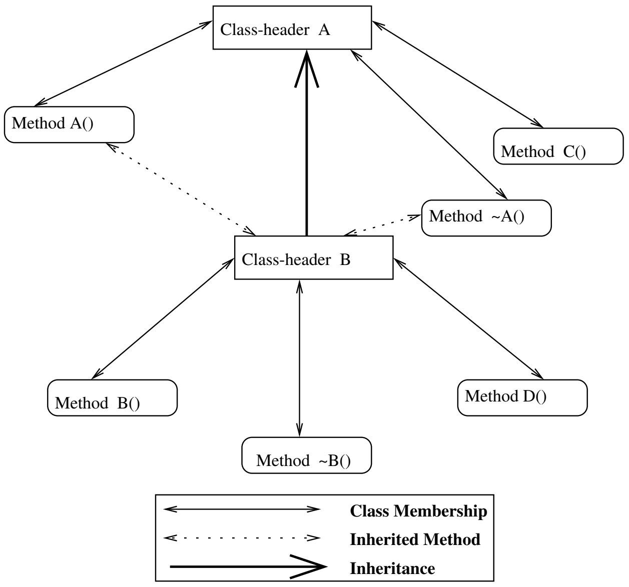
```
Class  A
  {
   Public:
      A( );
       void ~A( );
    Private:
       void   C( );

  }

 Class B:  Public A

  {
    B( );
   ~B( );
    void D( );
  }
```

(a)



(b)

FIGURE 8.11 (a) Object-oriented program, and (b) its CHS.

edges. A class call graph also includes edges that represent method calls. A CCFG captures the static control flow relationships that exist within and among methods of the class. It consists of a class call graph in which each method header node is replaced by the CFG for its associated method. An interclass dependence graph captures the control and data dependences for interacting classes that are not in the same hierarchy. In object-oriented programs, a composite class may instantiate its component class either through a declaration or by use of an operation, such as *new*. Larsen and Harrold [53] constructed SDGs for these individual classes, groups of interacting classes and finally the complete object-oriented program. Slices are computed using a graph-reachability algorithm. Some results on intermediate representations and slicing of concurrent object-oriented and multithreaded programs have been reported in the literature [54–57].

8.7.2 Dynamic Slicing of Object-Oriented Programs

Zhao [58] presented an algorithm for dynamic slicing of object-oriented programs, and adopted the following concepts:

- A slicing criterion for an object-oriented program is of the form (s, v, t, i), where s is a statement in the program, v is a variable used at s and t is an execution trace of the program with input i.
- A dynamic slice of an object-oriented program on a given slicing criterion (s, v, t, i) consists of all statements in the program that actually affected the value of the variable v at the statement s.

Zhao [58] introduced the concept of dynamic object-oriented dependence graph (DODG). Construction of DODG involves creating a new node for each occurrence of a statement in the execution history, and creating all the dependence edges associated with the occurrence at runtime. His method of construction of the DODG of an object-oriented program is based on performing dynamic analysis of data and control flow of the program. This is similar to the methods of Agrawal and Horgan [31] and Agrawal et al. [59] for constructing DDGs of procedural programs. Computation of dynamic slices using the DODG is carried out as a graph-reachability problem. Implementation details of the algorithm have not been reported in [58].

8.8 Conclusions

We started with a discussion on the basic concepts and terminologies used in the area of program slicing. We also reviewed the recent work in the area of program slicing, including slicing of sequential, concurrent and object-oriented programs. An interesting trend that is visible in the slicing area is its step-lock advancements to the program slicing techniques with the programming language trends. By starting with the basic sequential program constructs researchers are now trying to address various issues of slicing distributed object-oriented programs. Also, because modern software products often require programs with millions of lines of code, development of parallel algorithms for slicing has assumed importance to reduce the slicing time.

References

[1] M. Weiser, Program Slices: Formal, Psychological, and Practical Investigations of An Automatic Program Abstraction Method, Ph.D. thesis, University of Michigan, Ann Arbor, MI, 1979.

[2] M. Weiser, Reconstructing sequential behavior from parallel behavior projections, *Inf. Process. Lett.*, 17(10), 129–135, 1983.

[3] K. Gallagher and J. Lyle, Using program slicing in software maintenance, *IEEE Trans. Software Eng.*, SE-17(8), 751–761, August 1991.

[4] J.A. Beck, Interface Slicing: A Static Program Analysis Tool for Software Engineering, Ph.D. thesis, West Virginia University, Morgantown, WV, April 1993.

[5] A.D. Lucia, A.R. Fasolino and M. Munro, Understanding function behaviors through program slicing, in *Proceedings of the Fourth IEEE Workshop on Program Comprehension, Berlin, Germany, March 1996*, IEEE Computer Society Press, Los Alamitos, CA, 1996, pp. 9–18.

[6] M. Harman, S. Danicic and Y. Sivagurunathan, Program comprehension assisted by slicing and transformation, in Proceedings of First UK Workshop on Program Comprehension, Durham University, U.K., M. Munro, Ed., July 1995.

[7] J. Beck and D. Eichmann, Program and Interface Slicing for Reverse Engineering, in Proceedings of the IEEE/ACM 15th International Conference on Software Engineering, ICSE ’93, Baltimore, 1993, pp. 509–518.

[8] A. Cimitile, A.D. Lucia and M. Munro, A specification driven slicing process for identifying reusable functions, *Software Maintenance: Res. Pract.*, 8, 145–178, 1996.

[9] L.M. Ott and J.J. Thuss, Effects of Software Changes in Module Cohesion, in Proceedings of the Conference on Software Maintenance, November 1992, pp. 345–353.

[10] L.M. Ott and J.J. Thuss, Slice based metrics for estimating cohesion, in *Proceedings of the IEEE-CS International Metrics Symposium, Baltimore, Maryland*, IEEE Computer Society Press, Los Alamotos, CA, May 1993, pp. 71–81.

[11] F. Tip, A survey of program slicing techniques, *J. Programming Languages*, 3(3), 121–189, September 1995.

[12] D. Binkley and K.B. Gallagher, Program slicing, *Adv. Comput.*, M. Zelkowitz, Ed., Academic Press, San Diego, CA, 43, 1–50, 1996.

[13] M. Weiser, Program slicing, *IEEE Trans. Software Eng.*, 10(4), 352–357, July 1984.

[14] B. Korel and J. Laski, Dynamic program slicing, *Inf. Process. Lett.*, 29(3), 155–163, October 1988.

[15] B. Korel and J. Rilling, Dynamic program slicing methods, *Inf. Software Technol.*, 40, 647–659, 1998.

[16] J.-F. Bergeretti and B. Carré, Information-flow and data-flow analysis of while-programs, *ACM Trans. Programming Languages Sys.*, 7(1), 37–61, January 1985.

[17] J.R. Lyle and M.D. Weiser, Automatic program bug location by program slicing, in Proceedings of the Second International Conference on Computers and Applications, Peking, China, June 1987, pp. 877–882.

[18] D. Jackson and E.J. Rollins, A new model of program dependences for reverse engineering, in Proceedings of the Second ACM SIGSOFT Symposium on the Foundations of Software Engineering, New Orleans, LA, December 1994, pp. 2–10.

[19] T. Reps and G. Rosay, Precise interprocedural chopping, in Proceedings of the Third ACM Symposium on the Foundations of Software Engineering, Washington, D.C., October 1995, pp. 41–52.

[20] R. Mall, *Fundamentals of Software Engineering*, Prentice Hall of India, New Delhi, 1999.

[21] S. Bates and S. Horwitz, Incremental program testing using program dependence graphs, *in Conference Record of the Twentieth ACM Symposium on Principles of Programming Languages*, ACM Press, New York, 1993, pp. 384–396.

[22] R. Gupta, M.J. Harrold and M.L. Soffa, Program slicing-based regression testing techniques, *J. Software Testing, Verification Reliability*, 6(2), June 1996.

[23] I. Forgacs and A. Bertolino, Feasible test path selection by principal slicing, in *Proceedings of the 6th European Software Engineering Conference (ESEC/FSE97), Lecture Notes in Computer Science*, 1301, Springer-Verlag, New York, September 1997.

[24] J.R. Lyle, D.R. Wallce, J.R. Graham, K.B. Gallagher, J.E. Poole and D.W. Binkley, A Case Tool to Evaluate Functional Diversity in High Integrity Software, U.S Department of Commerce, Technology Administration, National Institute of Standards and Technology, Computer Systems Laboratory, Gaithersburg, MD, 1995.

[25] S. Horwitz, J. Prins and T. Reps, Integrating non-interfering versions of programs, *ACM Trans. Programming Languages Syst.*, 11(3), 345–387, July 1989.

[26] J.M. Beiman and L.M. Ott, Measuring functional cohesion, *IEEE Trans. Software Eng.*, 20(8), 644–657, August 1994.

[27] J. Ferrante, K. Ottenstein and J. Warren, The program dependence graph and its use in optimization, *ACM Trans. Programming Languages Syst.*, 9(3), 319–349, 1987.

[28] A.V. Aho, R. Sethi and J.D. Ullman, *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, MA, 1986.

[29] K. Ottenstein and L. Ottenstein, The program dependence graph in software development environment, *in Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, SIGPLAN Notices*, 19(5), 177–184, 1984.

[30] S. Horwitz, T. Reps and D. Binkley, Interprocedural slicing using dependence graphs, *ACM Trans. Programming Languages Syst.*, 12(1), 26–61, January 1990.

[31] H. Agrawal and J. Horgan, Dynamic program slicing, *in Proceedings of the ACM SIGPLAN '90 Conf. on Programming Language Design and Implementation, SIGPLAN Notices, Analysis and Verification*, White Plains, New York, 25(6), 246–256, June 1990.

[32] G.B. Mund, R. Mall and S. Sarkar, An efficient dynamic program slicing technique, in *Information and Software Technology*, Elsevier Press, 44(2002), 123–132.

[33] E.W. Dijkstra, Cooperating sequential processes, *Programming Languages*, F. Genuys, Ed., Academic Press, 1968, pp. 43–112.

[34] A. Burns, *Concurrent Programming in Ada*, Cambridge University Press, Cambridge, 1985.

[35] B.W. Kernighan and R. Pike, *The UNIX Programming Environment*, Prentice Hall, Englewood Cliffs, NJ, 1984.

[36] M. Rochkind, *Advanced UNIX Programming*, Prentice Hall, Englewood Cliffs, NJ, 1985.

[37] M.J. Bach, *The Design of the Unix Operating System*, Prentice Hall India, New Delhi, 1986.

[38] J. Cheng, Slicing concurrent programs — a graph theoretical approach, in *Proceedings of the 1st International Workshop on Automated and Algorithmic Debugging*, P. Fritzson, Ed., *Lecture Notes in Computer Science*, Springer-Verlag, Vol. 749, New York, 1993, pp. 223–240.

[39] D. Goswami, R. Mall and P. Chatterjee, Static slicing in UNIX process environment, *Software — Pract. Exp.*, 30(1), 17–36, January 2000.

[40] J. Krinke, Static Slicing of Threaded Programs, in Proceedings of ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE '98, 1998, pp. 35–42,

[41] M.G. Nanda and S. Ramesh, Slicing Concurrent Programs, in Proceedings of the ACM International Symposium on Software Testing and Analysis, ISSTA-2000, August 2000.

[42] B. Korel and J. Laski, Dynamic slicing of computer programs, *J. Syst. Software*, 13, 187–195, 1990.

[43] B. Korel and R. Ferguson, Dynamic slicing of distributed programs, *Appl. Math. Comput. Sci.*, 2(2), 199–215, 1992.

[44] E. Duesterwald, R. Gupta and M.L. Soffa, Distributed slicing and partial re-execution for distributed programs, in *Proceedings of the 5th Workshop on Languages and Compilers for Parallel Computing, Lecture Notes in Computer Science*, Vol. 757, Springer-Verlag, New York, 1992, pp. 329–337.

[45] D. Goswami and R. Mall, Dynamic slicing of concurrent programs, in *Proceedings of 7th International Conference on High Performance Computing, Lecture Notes in Computer Science*, Vol. 1970, Springer-Verlag, New York, 2000, pp. 15–26.

[46] M. Harman, S. Danicic and Y. Sivagurunathan, A parallel algorithm for static program slicing, *Inf. Process. Lett.*, 56(6), 307–313, 1996.

[47] M. Haldar, D. Goswami and R. Mall, Static slicing of shared memory parallel programs, in *Proceedings of 7th International Conference on Advanced Computing, Pune, India*, December 1998.

[48] D. Kung, J. Gao, P. Hsia, Y. Toyoshima and C. Chen, Design Recovery for Software Testing of Object-Oriented Programs, in Working Conference on Reverse Engineering, May 1993, pp. 202–211.

[49] D. Kung, J. Gao, P. Hsia, Y. Toyoshima, F. Wen and C. Chen, Change Impact Identification in Object-Oriented Software Maintenance, in International Conference on Software Maintenance, September 1994, pp. 202–211.

[50] D. Kung, J. Gao, P. Hsia, Y. Toyoshima, F. Wen and C. Chen, Firewall regression testing and software maintenance of object-oriented systems, *Object-Oriented Programming*, 1994.

[51] M.J. Harrold and G. Rothermel, Performing Data Flow Testing on Classes, Second ACM SIGSOFT Symposium on the Foundation of Software Engineering, December 1994, pp. 154–163.

[52] A. Krishnaswamy, Program Slicing: An Application of Object-Oriented Program Dependency Graphs, Technical Report TR-94-108, Department of Computer Science, Clemson University, 1994.

[53] L. Larsen and M.J. Harrold, Slicing Object-Oriented Software, in Proceedings of the 18th International Conference On Software Engineering, Germany, March 1996, pp. 495–505.

[54] J. Krinke, Static Slicing of Threaded Programs, in Program Analysis for Software Tools and Engineering (PASTE '98), ACMSOFT, 1998, pp. 35–42.

[55] J. Zhao, J. Cheng and K. Ushijima, A Dependence-Based Program Representation for Concurrent Object-Oriented Software Maintenance, in Proceedings of 2nd Euromicro Conference on Software Maintenance and Reengineering (CSMR '98), Italy, 1998, pp. 60–66.

[56] J. Zhao, Slicing concurrent Java programs, in Proceedings of the IEEE International Workshop on Program Compression, 1999.

[57] J. Zaho, Multithreaded Dependence Graphs for Concurrent Java Programs, in Proceedings of the 1999 International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE '99), 1999.

[58] J. Zhao, Dynamic Slicing of Object-Oriented Programs, Technical report SE-98-119, Information Processing Society of Japan, 1998, pp. 17–23.

[59] H. Agrawal, R.A. Demillo and E.H. Spafford, Dynamic Slicing in the Presence of Unconstrained Pointers, in Proceedings of the ACM 4th Symposium on Testing, Analysis, and Verification (TAV4), 1991, pp. 60–73.

9 Debuggers for Programming Languages

Sanjeev Kumar Aggarwal
Indian Institute of Technology, Kanpur

M. Sarath Kumar
Indian Institute of Technology, Kanpur

9.1 Introduction

A symbolic debugger is one of the most important tools in a software development environment. It is normally used to fix logical errors in a program. A debugger is used when a program does not produce expected behavior during execution, and the reasons for the unexpected behavior are not clear. The program may give unexpected results or may prematurely terminate with an error condition.

0-8493-1240-7/03/$0.00+$1.50

A symbolic debugger helps a programmer to examine the values of variables while a program is running. It also allows the programmer to change the value of a variable, suspend execution of a program and resume execution from a previously suspended session. It can monitor the state of the variables, may allow the programmer to change the value of a variable and may allow the programmer to test the effects of the change without going through the edit session. Symbolic debuggers work with the names of variables in the programs and not with the memory addresses of variables.

A debugger, usually:

- Displays program source code and permits browsing through the code
- Sets break points to suspend execution of program at break points
- Examines values of variables at the break points
- Executes the program by executing one statement at a time
- Examines and changes the value of a variable during execution
- Examines currently active routines and processes
- Examines the stack and symbol table
- Captures signals sent to the operating system by the program

Debuggers are complex pieces of software and require support from the compiler, assembler, linker and operating system (OS) to operate.

Debuggers control the program to be debugged by using special facilities provided by the hardware and OS. For example, Pentium provides interrupt instruction (INT), and UNIX environment provides ptrace () system call. The most common features used to control program execution are break points and single stepping.

As software systems are becoming larger and more complex, debuggers become very important tools in identifying and fixing software defects. Debuggers are used right from the beginning when the program is at the design stage and very little code has been developed until the time the complete software has been developed. Debuggers are used at intermediate steps to test modules and to check whether a module is ready for integration with the rest of the system. After the complete system is deployed, debuggers are used by developers who maintain and enhance the system to understand the intricacies of the code.

The oldest and perhaps the most commonly used technique for debugging is to insert print statements at the places where a bug is suspected to be present. This technique is both quick and effective for small programs. However, for large software systems an interactive debugger with graphic user interface support is used. In addition, other techniques such as inserting assertions in the code, printing to log files, looking at function call and process stacks and taking postmortem dumps are used.

An important method of debugging is to use symbolic debuggers. These debuggers interact with the developer using symbolic names defined in the program to be debugged. The user gets a view as if the program is directly executed on the target machine. Symbolic debuggers are the most convenient and the most frequently used debuggers. They have also been the most effective tools for debugging.

Historically, postmortem dumps are the oldest tools used for program debugging. These were followed by machine level debuggers and symbolic debuggers. The current state-of-the-art debuggers provide graphic user interfaces, and form part of a complete program development environment integrating editors, compiler and interpreter, debugger, etc.

Current state-of-the-art debuggers provide either command line interface (gdb on UNIX) or complete graphic user interface (GUI) environment (on Windows). Both support a rich set of features. Usually, command line debuggers come as stand-alone modules. However, GUI-based debuggers normally come with an integrated program development environment consisting of editor, compiler, linker, etc. Modern debugging environments also support logging calls to user, system functions and profiling.

Debuggers control execution of programs to be debugged by using break points and single stepping. Single stepping may be either at the instruction level or at the statement level. Break points and single stepping are the most basic features of debuggers.

Break points are supported by placing a special code in the code sequence to be executed. When the special code is executed, it causes a trap or interrupt to occur. The trap is reported to the debugger by the processor and the OS. Most modern processors and OSs provide special features that are used by debuggers.

Single stepping is a method that controls the execution of the program to be debugged at the instruction and statement level. It allows the debugger to control the processor executing the program to be debugged. Again, most modern processors and OSs provide special features that are used by debuggers to do single stepping.

Once an execution stops due to either the break point or the exception condition, the debugger, depending on the functionality built into it, reports to the user the location where the program stopped. It also reports back why the program stopped, makes available stack trace, values of variables, values of registers and contents of memory locations. It also allows the user to change the contents of variables, registers and memory locations to fix the exception conditions. The execution then continues from this point. This saves edit–compile cycles.

Debuggers modify the code of the program to be debugged. In the case of the UNIX OS, the code to be debugged is executed in the process space of the debugger. This may make the program unstable. It is desirable that the changes to the code to be debugged are minimized. The inserted code should not change the functionality of the code to be debugged. Thus something like the Heisenberg uncertainty principle is applied in the context of debugging [8]. Both the debugger and the code to be debugged reside in the same memory that in turn is controlled by the OS. In the UNIX environment both the debugger and the code to be debugged run as independent processes. Because the debugger process cannot change the process space of the code to be debugged, the debugger copies the code into its own process space. This can affect the behavior of the debugged program.

Today, compiler optimizations are an integral part of compilers. The optimizations reorder, eliminate and duplicate code. This procedure may alter the program control flow. Therefore, symbolic debuggers cannot mimic the exact behavior of the source code. However, a debugger is expected to provide correct mapping between the line numbers in the source code and the object code locations, and between variable names and corresponding memory locations. The optimizing transformations affect these mappings. Therefore, support must be built both in the compilers and the debuggers for debugging optimized code.

Debugging becomes easier if the original source code can be mapped directly to the translated machine code. The developer has a view of the original source code normally written in high-level language. In most cases, machine level instructions may not be useful or meaningful to the developer. Therefore, most developers prefer to work with the original source code when debugging. This becomes increasingly important as the systems become larger and more complex. The developer should get an illusion of the code directly executed on a machine whose machine language is the same as the source language. For this reason interpreters provide good debugging support with much less effort. Compilers have to extract and store extensive information from the source code, and the way the source code maps onto the target machine. However, situations arise where the developer needs to get access to machine-specific details such as register values, process and function stack and memory dump. The source level debugger must have built-in support to provide the low-level machine specific details.

A stand-alone debugger like gdb is used only for program debugging. It does not support any other activity in the program development cycle. An integrated development environment that includes a debugger enhances productivity. Therefore, there is great merit in integrating a debugger with the rest of the environment. In such environments the editor can be used to set break points. When the program stops during execution, control can come back to the editor at the location where the program stopped. Another advantage of an integrated environment is that compiler data structures such as symbol tables become available to the debugger. Also, when expressions need to be evaluated, these would be interpreted using the same compiler that generated the code. This makes the program behavior consistent.

Debuggers are also offered with interpreted languages. Such environments provide better support to debuggers when compared with compiled environments. Such debuggers are easier to use because they provide immediate feedback about any change. Also, they do not require support from the hardware and the OS.

A good discussion about debuggers may be found in [16, 21]. Other useful sources of information about debuggers are [5, 6]. Information about some commercial debuggers is available from [2, 10, 14, 20, 22].

The rest of the chapter is organized as follows: Section 9.2 describes debugger architecture; Sections 9.3 to 9.5 discuss hardware, OSs and compiler and linker support issues related to a debugger; Sections 9.6 to 9.8 examine issues related to break points, program stack and expression evaluation; Section 9.9 covers the design of a typical debugger; and Section 9.10 discusses issues related to debugging optimized code.

9.2 Debugger Architecture

A debugger requires support from the underlying OS. Therefore, it must be closely tied with the OS. At the top level it must have a user interface that presents a set of features to the user. The interface is used by the user to gather information and to modify the values of the variables. Figure 9.1 shows the architecture of a debugger.

A debugger presents several views to the user. It presents the source that is the most important view required for debugging. This view also presents the editor window to the user. Other views may be the machine level execution view, about the modules loaded during execution, errors encountered during execution, threads and process view and any other information that may be useful for the user in program debugging.

A very important piece of information required for debugging is the stack information. This gives the function and procedure stack frame. It consists of stack frames, with each frame an activation record. If the program halts with an error, the activation record on the top of the stack frame immediately gives information about the function in which the program stopped.

Other very important information concerns the break points set by the user. Breakpoints allow the programmer to control execution. These are the points where the programmer wishes the application to halt for examining the values of variables.

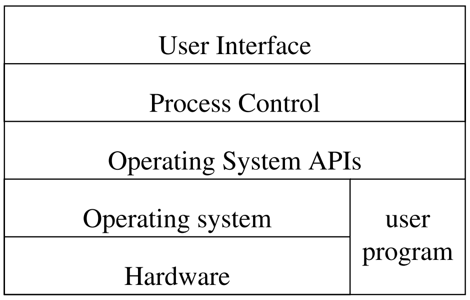


FIGURE 9.1 Architecture of a debugger.

The machine level execution view is important for any large piece of software. Although the source level view (symbolic view) is normally enough for debugging, it is not complete. Any large application is bound to interact with the OS and system libraries. This occasionally requires going through machine instructions. Often a problem can be solved by examining the assembly level code and the values in the registers. A machine level execution view normally shows the assembly code, the contents of the registers, the memory dump, the various system flags and the hardware stack.

Process control takes place in the layer above OS application program interfaces (APIs). The application to be debugged is a process. This layer is responsible for process creation and symbol table access. The symbol table provides mapping between the source statement and the address of the executable instructions. This information is necessary for setting up break points. The symbol table also provides the type information that is required for mapping the bit sequences back to values.

The execution control sequence of the debugger presents a source code execution view of the program. This controls the execution to the next break point, single stepping through a machine level instruction or statement, handling exceptions and evaluating functions.

The expression interpreter evaluates values of expressions and functions. It must use, as far as possible, the compiler parser for expression evaluation. It must be ensured that the expression evaluator uses the built-in parser, symbol table, values in memory and machine execution engine to evaluate expressions.

The debugger accesses the user program through a set of APIs provided by the OS. The APIs must provide a way to access (both for read and write) the memory of the process, and to control the execution (through break points and single stepping) of the process. These APIs should not alter the process to be debugged so that they have minimal impact on the application involved.

9.3 Hardware Support

The minimum support a hardware must provide to a debugger is to specify a break point. The break point is a location in the code such that execution halts when the control reaches this location. This support is normally provided through an illegal instruction that can be written into the code of the program. This instruction generates an interrupt that informs the OS that a break point has occurred. No additional facility is required for single stepping because it can be modeled as a special case of break pointing.

Most processors provide special instructions to support break points. Such instructions generate a trap to the OS when executed. The information in turn is passed on to the debugger. Normally, these instructions are kept of minimum possible size — one byte on Intel processors to one word on Compaq Alpha processors. The debugger is provided support by the OS so that it can read and modify the process space of the program to be debugged. The debugger reads the instruction at the address where the break point is to be set and replaces it with the trap instruction. The original instruction at the address is saved by the debugger. During execution of the code, when a trap instruction (corresponding to a break point) is encountered, control is transferred to the debugger. The debugger takes appropriate action at this point. When the debugger wants the code to continue execution beyond the break point, it replaces the trap instruction by the earlier saved instruction, sets the computer back to the saved instruction and continues the execution.

Normally, single stepping is simulated using break points. Most modern processors do not provide any special support for single stepping. Wherever the debugger has to execute a single instruction, it is assumed that the next instruction is a break point instruction.

9.4 Operating System Support

Interaction between the debugger and the OS is crucial support required for debugging. The debugger must be able to modify the process space of the program involved in debugging, to inform the OS about the program, to inform the OS to continue to execute the program and to collect information about the program when it stops because of a fault. Normally, all OSs provide this kind of support for debugging programs. The support is provided through an API.

A debugger is neither a part of the OS nor a privileged application. It uses system calls that are especially provided by the OS for this purpose. These system calls ensure that whenever an event such as a break point or trap occurs in the program to be debugged, the information is passed on to the debugger and control is transferred to the debugger. In the process of transferring control to the debugger, the OS modifies its internal data structures to reflect that the control of the program is with the debugger. UNIX ptrace() is an example of such a system call.

The following appears in the manual pages of Linux:

The ptrace system call provides a way by which a parent process can observe and control the execution of another process, and examine and change its core image and registers. It is primarily used to implement breakpoint debugging and system call tracing.

The parent can initiate a trace by calling fork and having the resulting child do a PTRACE_TRACEME, followed (typically) by an exec. Alternatively, the parent may commence trace of an existing process using PTRACE_ATTACH.

While being traced, the child process stops each time a signal is delivered, even if the signal is being ignored. The parent is notified at its next wait and may inspect and modify the child process while it is stopped. The parent then causes the child to continue, optionally ignoring the delivered signal (or even delivering a different signal instead).

When the parent is finished tracing, it can terminate the child with PTRACE_KILL or cause it to continue executing in a normal, untraced mode via PTRACE_DETACH.

Parameter PTRACE_TRACEME indicates that this process is to be traced by its parent. Any signal (except SIGKILL) delivered to this process causes it to stop and its parent to be notified via wait. Also, all subsequent calls to exec by this process causes a SIGTRAP to be sent to it, giving the parent a chance to gain control before the new program begins execution.

Refer to UNIX manual pages for details of ptrace().

9.5 Compiler and Linker Support

Most modern compilers provide support for debugging. Normally, the object programs generated by compilers do not have information such as line numbers, variable names, variable types and structures. However, for symbolic debugging this information plays a very crucial role. When a program is compiled for use in debugging, information is generated by the compiler and embedded in the object code. When an error occurs during the execution of the program, the information and the control are transferred to the debugger. This additional information usually makes the object code longer and increases the execution time. For almost all applications, debugging is turned off while generating the production quality object code.

A symbolic debugger must be able to give a mapping between source line numbers and object code addresses. This helps the programmer in setting up break points using the source code view, and the debugger puts the break point at the corresponding location in the object code. Line number information is simple to generate. However, in case of optimizing compilers it becomes difficult to provide the mapping between line numbers and object code. Optimizing compilers move the code around and may remove part of the code. This changes the sequence of code in the object file, and it can no longer be matched with source line numbers. This issue is discussed in Section 9.10 on debugging optimized code.

The other information required for symbolic debugging is variable names, their types and addresses. The compiler must generate this information. This information is put in the symbol tables, which are not removed by the linkers. The tables include source line number to address mapping, file names, symbol names, symbol types, symbol addresses, lexical scoping, etc.

Typically, compilers generate debug information for all the variables in the program including the variables in the header files. Because many program files have common header files, compilers generate duplicate information. Linkers normally remove the duplicate information before generating final debuggable executable code [13].

9.6 Break Points

Break pointing is central to the debugging process. It must be represented at two levels, the source program level and the executable code level. At the source program level it is associated with the source code view and corresponds to a line in the source code. At the executable program level it is associated with the machine code and corresponds to an address in memory.

In addition to the break points set by the user each debugger uses internal break points that are set by the debugger. Single stepping uses a sequence of internal break points to stop the program after each instruction. Another functionality that requires internal break points is step over. Step over executes a function to completion without going inside the function code. This is achieved by setting an internal break point at the return address of the function.

The break point mechanism is also used for logging information. Break points do not always need to be used for stopping execution of code. They can also be used for gathering information at certain points in the program. The generated information is used later for postmortem debugging. This is similar to inserting print statements in the code to gather information. However, using the breakpoint mechanism has obvious advantages. It does not require modification and recompilation of the code. The break point mechanism can also be used for counting how many times control passes through a break point.

Break points do not always need to be trap instructions inserted in the code undergoing debugging. Any other valid instruction can be used. This instruction can be used for code patching. By using this mechanism, changes can be made to the code without changing and recompiling the source code. Therefore, debuggers can also be used as profilers.

9.7 Program Stack

When a program is stopped at an error and control is transferred to the debugger, it is important to know the program context. The context information includes line number, function name and file name with complete directory path information. Another very important context of information is the calling sequence of the functions. This sequence gives information about which functions have been invoked in the program before the fault occurred. Part of this information is available in the program stack.

Whenever a function is invoked, its activation record is pushed on the stack. After the function returns control to the caller, the activation record is removed from the stack. At any point of time the stack has the information of the active functions in the program. Each activation record is identified by two pointers: stack pointer and frame pointer. Activation records also contain values of local variables, parameters, return addresses, pointers to nonlocal variables, previous stack pointer, frame pointer and saved registers.

The complete call sequence information is stored in another data structure called the call tree. This structure keeps information about all the procedures that have been invoked before the current activation. However, information such as local data, pointers to nonlocal data, stack and frame pointers is no longer available for procedures that are not currently active.

9.8 Expression Evaluation

An important part of debugging is to set break points and transfer control to the debugger either when a break point is encountered or when an error occurs. However, another major activity starts after the debugger gets control of the program. The user may want to evaluate an expression at this point.

The symbolic debugger must evaluate the value of an expression using the same identifiers that occur in the source program. Therefore, a debugger must have a built-in expression interpreter. The expression interpreter must conform to the syntax and semantics of the source language. However, this interpreter should not create its own variable space. It must use the values of variables from the locations at which the variables are stored. The name-to-location mapping is available in the debugger symbol tables. This ensures that no inconsistency exists in the values of the variables.

Normally, debuggers try to adapt the syntax and semantic analysis phases of the compiler to build the expression interpreter. This ensures that the syntax and semantic checks performed by the expression interpreter are the same as the ones performed by the compiler. Also, it saves on the effort involved in building and testing the expression interpreter.

9.9 Design of a Typical Debugger

The top level of a debugger consists of four modules:

- Query processor
- Symbol resolver
- Expression interpreter
- Debug support interface

Figure 9.2 shows top level design of a debugger.

Descriptions of these four modules are given in the following subsections.

9.9.1 Query Processor

The query processor module addresses all the issues concerning the commands accepted and is the user interface module. This processor also validates the user command and, depending on the nature of the request, invokes the other modules as and when required. Figure 9.3 shows structure of the query processor.

The command preprocessor validates the debugger commands. The debugger command is passed to the lexical analysis phase only if it is valid. Lexical analysis reads the user debugger commands and translates them into a linear sequence of lexical symbols (lexemes or tokens). The tokens are all debugger commands, numbers, identifiers and strings (everything until the end of the line). Syntax analysis checks that the tokens output by the lexical analyzer occur in patterns permitted by the syntax of the debugger command language. All syntactic errors are detected in this phase. The grammar used to parse the commands is the main design issue. The view command processor processes all the debugger view commands. It consists essentially of text editing and string manipulation routines. The record/playback command processor consists of routines that handle record and playback commands.

9.9.2 Symbol Resolver

The symbol resolver module addresses issues concerning symbol accessibility and the line–address mapping. It handles all queries related to symbolic references of source program entities and their corresponding addresses in the target system. Figure 9.4 shows the structure of the symbol resolver.

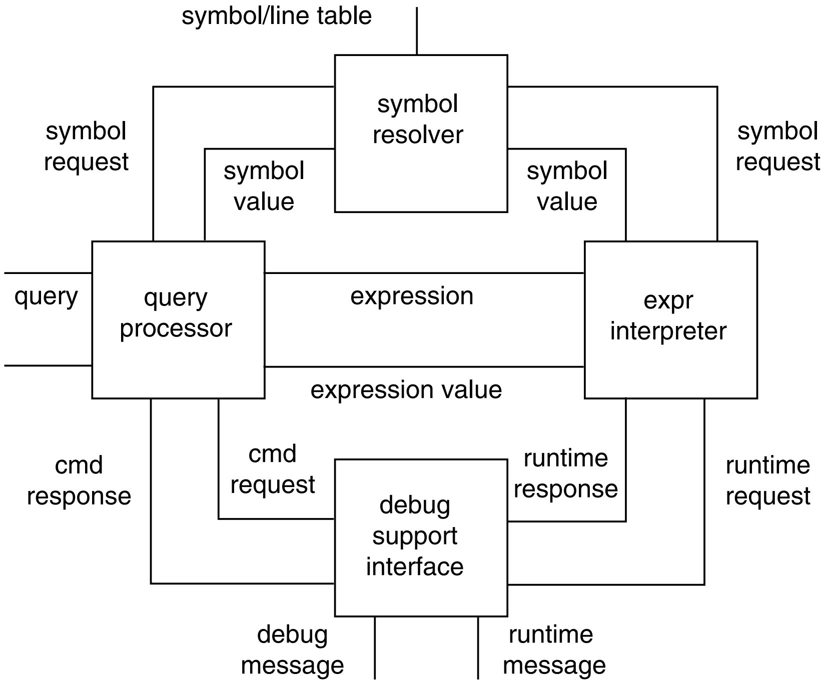


FIGURE 9.2 Top-level design of a debugger.

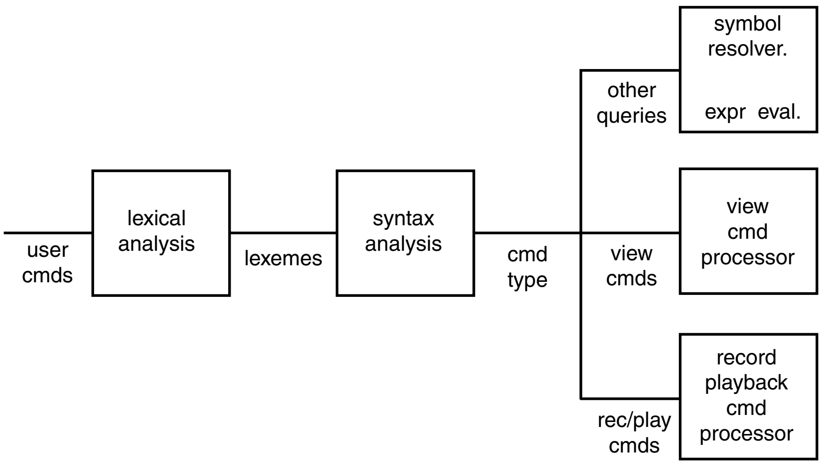


FIGURE 9.3 Structure of the query processor.

9.9.2.1 Debugger Tables

Debugger tables provide interface between the debugger and the linker. The following tables may contain the interface information:

Symbol table
String table
Line table
More table

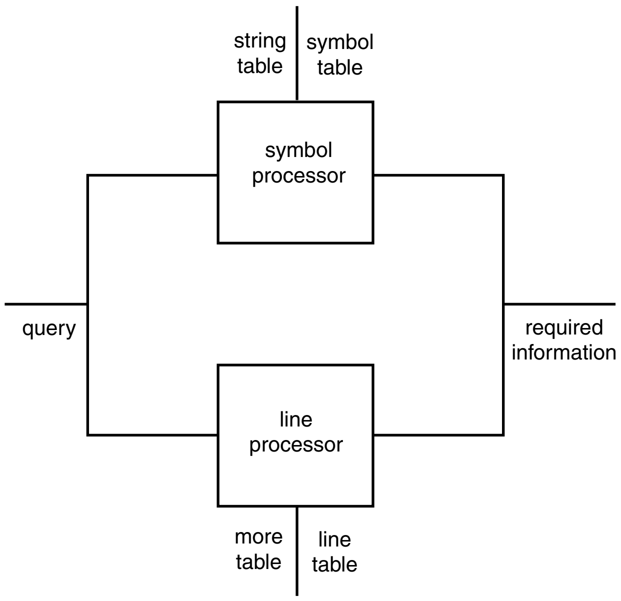


FIGURE 9.4 Structure of the symbol resolver.

Descriptions of these tables follow:

The symbol table consists of one symbol table per file. It consists of main and auxiliary entries. These entries correspond to each symbol that is defined and referenced in the source file. In addition, it also contains some special symbols that are used by the table searching algorithms. The symbol table reflects the source file structure with respect to scope and visibility.

The string table contains symbol names that do not fit into the symbol table. The symbol names are terminated by NULL character.

The line table gives the mapping of line numbers of source files to the corresponding physical addresses. The line table consists of the following fields: *Address* contains the physical address of a line. *Line number* contains the source line number relative to the beginning of the file. The first entry of the table contains the symbol table index of the file name. *More table offset* contains an offset into the more table that contains more information that is required for single-step operation.

The more table contains information necessary for single-step operation. It has the following format:

Number of successor lines	Number of procedures	Successor line addresses	Symbol table indices

Each line has a set of successor lines and zero or more procedure calls. The first two fields in each entry of the more table contains the number of successor lines and the number of procedure calls in the line. The variable part of each entry consists of the successor line addresses and symbol table pointers for each procedure. The successor line of all RETURN statements and of the last executable statement of the procedure is specified as −1.

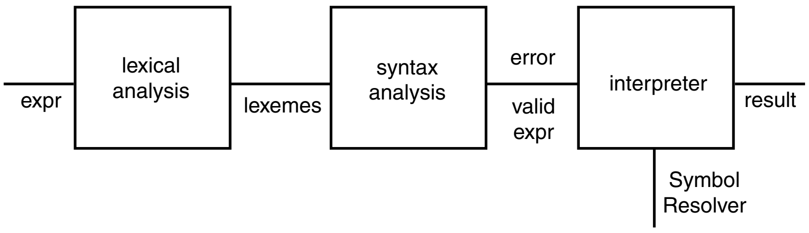


FIGURE 9.5 Decomposition of the expression interpreter module.

9.9.3 Expression Interpreter

This module is invoked whenever a part of the query has an expression and the examine debugger command is used. This command enables the user to examine program entities, to evaluate expressions and to deposit values. This expression following the examine command is parsed and interpretively evaluated. The results are communicated to the invoking module. This module invokes the symbol resolver if required (e.g., for getting the address maps of the referenced variables). The grammar of the expressions is derived from the compiler. The interpretive evaluation is performed during a recursive traversal of the expression. The module can be decomposed as shown in Figure 9.5.

9.9.3.1 Lexical Analysis

The function of this lexical analyzer is to read the given expression and translate it into a linear sequence of lexical symbols such as identifiers, constants, strings and separators. This lexical analyzer is the same as the one used by the compiler.

9.9.3.2 Syntax Analysis

The function of the syntax analyzer is to check that the tokens output by the lexical analyzer occur in patterns permitted by the syntax of the expressions. The grammar of the expressions is derived from the compiler. The type of each nonterminal in the expression parser consists of a structure that contains the address, type, size and other information of the result.

9.9.3.3 Interpreter

This phase performs interpretive evaluation during a recursive traversal of the expression. Only basic-type checking of operands is performed. The symbol resolver is invoked to get symbol information. The module makes use of the runtime routines that are used by the compiler for similar operations.

9.9.4 Debug Support Interface

The debug interface contains a library of procedures that would communicate to the debug support.

9.10 Debugging Optimized Code

Compiler optimizations are an integral part of all modern compilers. However, optimizations reorder, eliminate and duplicate code. Therefore, source level debuggers working with optimized code cannot mimic the behavior of the original unoptimized program. The perturbations caused by the optimizations prevent most source level debuggers from providing support for debugging optimized code. The traditional ways of designing production code is to first test and debug the unoptimized code and then optimize the code to get the performance enhancements and release the product.

Many situations occur where debugging optimized code is necessary and desirable, as follows:

- A program may run correctly when compiled without optimizations but may fail when compiled enabling optimizations. This can occur even though the optimizer is correct. Compiler optimizations are correctness preserving transformations for a correct program. Correctness preserving transformations are not guaranteed to preserve the behavior of a program that behaves differently for different executions. For example, programs using values of uninitialized variables behave differently for different executions.
  - Reordering of the statements may cause underflow or overflow.
  - Exceptions may occur because the value of the uninitialized variable changes when we change the data layout of the program. An array out of bound exception might occur due to change in the data layout of the program caused by optimizations.
- Final production code is generally an optimized version. One should be able to debug this optimized code by looking at the core file and a bug report.
- Programs with timing and instability problems behave differently when compiled with optimizations and without optimizations. In such cases we should have the ability to debug optimized code.
- There may be constraints on the space and time requirements of a program. This situation mainly occurs in designing embedded software systems. The optimized version of a program may satisfy the requirements; when we need to debug such a program, we should have the ability to debug optimized code.
- An optimizing compiler may have bugs.

9.10.1 Previous Work

Hennessey [9] addressed the problems in debugging of optimized code. He introduced the terms *nonresident* and *noncurrent* variables. At a break point, a variable is said to be nonresident if no runtime location corresponds to the variable. At a break point, a variable is said to be noncurrent if a runtime location corresponds to the variable, but the value of the variable is not the expected value of the variable. Hennessey showed how to detect variables that were nonresident and noncurrent at a break point when local optimizations were performed.

Zellweger [23] addressed the problem of correctly mapping the break points set in the source code to the corresponding places in optimized object code. Simmons et al. [7] showed that when debugging optimized code, providing the same behavior as an unoptimized program is not feasible and is impractical. Copperman [3] investigated the problem of detecting endangered variables caused by global optimizations. Reza and Gross [17–19] described ways to find endangered and noncurrent variables and to recover the expected values of the variables. They also addressed the code location problem in detail. Patil et al. [12] described a new framework for debugging optimized code. Kumar [11] addressed the problem of how to overcome the perturbations caused by optimizations and change the value of variables at debug time.

9.11 Perturbations Caused by Optimizations

The debugger maintains source-to-object and object-to-source mappings to set break points and report asynchronous break points (runtime exceptions, user interrupts). In source-to-object mapping, for each source statement the debugger stores the object code locations where the break point can be mapped in the object code. The debugger uses source-to-object mapping to set control break points.

In object-to-source mapping, the debugger stores the source statements corresponding to each object instruction. The debugger uses the object to source mapping to report asynchronous break points.

While debugging unoptimized code, source-to-object and object-to-source mappings are the same. For each source statement there is a fixed object code location where the break point set on that source statement can be mapped. For each group of object statements a corresponding unique source statement exists. The object-to-source and source-to-object mappings are both one-to-one mappings. Optimizations reorder, eliminate and duplicate source statements. Therefore, the source-to-object mapping and the object-to-source mappings become different. Both the mappings are no longer one-to-one mappings. For each source statement more than one object code location may exist; in such a case the debugger has to decide to which object code location a break point has to be mapped. The problem of correctly setting the break points in optimized code is called the *code location problem.*

In unoptimized code, object instructions are executed in the order specified in the source program. Thus, the variables have expected values. Optimizations eliminate and reorder the execution of the statements. The order in which the object instructions execute is different from the source execution order and this causes the variables to have values different from expected values. In unoptimized code each variable has a runtime location corresponding to that variable throughout the scope of the variable. Due to optimizations, a variable may not be allocated a runtime location, or the runtime location allocated to a variable may be reused to store the value of another variable. When the debugger wants to display the value of a variable, no runtime location exists for the variable, or more than one location might contain the value of the variable. The debugger cannot decide which location contains the expected value of the variable. These problems are called *data value problems*.

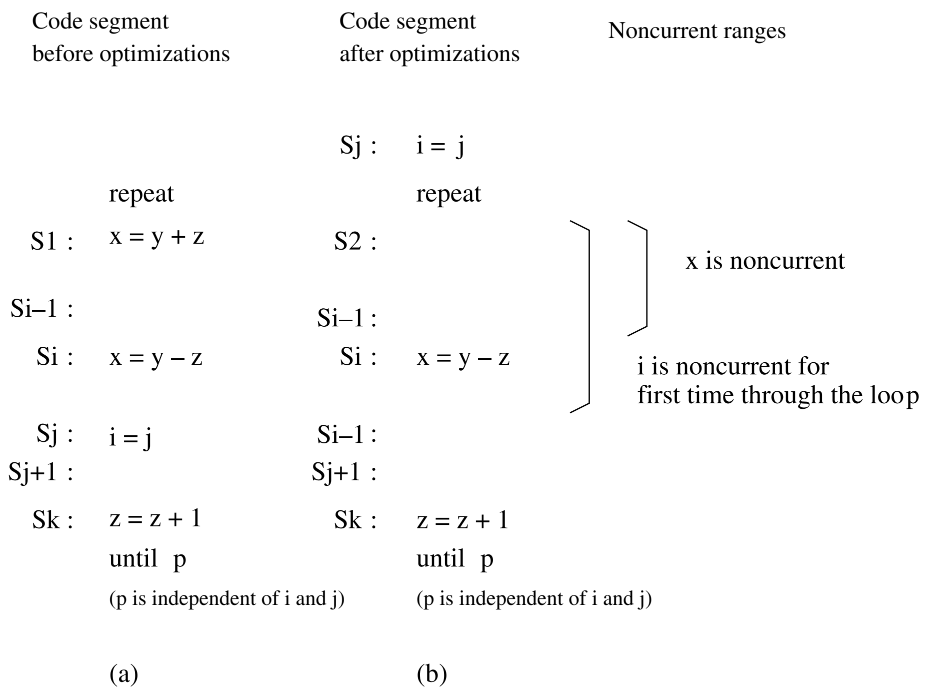
9.11.1 Example: Dead Code Elimination and Loop Invariant Code Motion

In Figure 9.6 a simple example depicts how optimizations affect debugging of optimized code. Figure 9.6(b) is the resultant code after performing dead code elimination and loop invariant code motion on the code in Figure 9.6(a). The optimizations performed cause both code location problems and data value problems.

9.11.1.1 Code Location Problems

Statement S_j is moved out of the loop due to loop invariant code motion. When the user sets a break point at S_j, the debugger should find the appropriate location in the object code to map this break point. If the debugger sets the break point at statement S_j in Figure 9.6(b), then the break point is triggered only once and all the variables assigned in statements S_1 to S_{j-1} are noncurrent. If the debugger sets the break point at statement S_{j-1} in Figure 9.6(b), then the break point is triggered the expected number of times, but the variable *i* is noncurrent at the break point. If the break point is set at statement S_{j+1} in Figure 9.6(b), then the break point is triggered the expected number of times and the variable *i* is current; however, if the statement S_j is the last statement of the basic block, then there is no statement after S_j. The effect is summarized as follows:

Break Point in Figure 9.6(a)	Break Point in Figure 9.6(b)	Number of Times Break Point Is Triggered	Endangered Variables at Break Point
S_j	S_j	Once	Variables assigned in S_1 to S_{j-1}
S_j	S_{j-1}	n Times	Variables assigned in S_j
S_j	S_{j+1}	n Times	None


Code segment before optimizations
Code segment after optimizations
Noncurrent ranges
x is noncurrent
i is noncurrent for first time through the loop
(a)
(b)

```
             (a)                          (b)

                                  Sj :    i = j
          repeat                          repeat
  S1 :    x = y + z               S2 :
Si–1 :                          Si–1 :
  Si :    x = y – z               Si :    x = y – z
  Sj :    i = j                 Si–1 :
Sj+1 :                          Sj+1 :
  Sk :    z = z + 1               Sk :    z = z + 1
          until  p                        until  p
          (p is independent of i and j)   (p is independent of i and j)
```

FIGURE 9.6 Sample code to demonstrate the perturbations caused by optimizations.

9.11.1.2 Data Value Problems

By assuming that there are no uses of x in statements S_2 to S_{i-1}, the assignment to x in S_1 is dead and may be eliminated. As the statement S_1 is deleted, at all break points set at statements S_2 to S_{i-1} the value of x is noncurrent. Statement S_j is moved out of the loop. This causes the value of i to be noncurrent from statements S_2 to S_{j-1} for the first time through the loop.

9.11.2 Approaches to Debugging Optimized Code

The major approaches to debugging optimized code are:

- Ignore the fact that optimizations have been performed resulting in anomalous behavior.
- Try to hide all optimizations performed and provide the expected behavior.
- Expose all the optimizations that are performed and leave it to the user to figure out how optimizations affect the program.
- Provide truthful behavior.

9.11.2.1 Ignoring Optimizations

In this approach, the debugger does not consider whether optimizations are performed. The debugger just shows the current execution state of the program. When the value of a variable is queried, the debugger displays the value present in the runtime location corresponding to the variable and this value may be different from the expected value of the variable. The execution path followed may be different from the expected execution path. These types of debuggers are easy to design and are less helpful when debugging optimized code.

Some debuggers allow debugging without considering whether optimizations are performed. This kind of debugger shows values different from the expected values of the variables and execution paths different from the expected paths. This type of the debugger is easy to design, but not of much use to a novice user when debugging optimized code.

9.11.2.2 Providing Expected Behavior

This is the ideal approach for debugging optimized code. In this approach the debugger manages the effects of the optimizations. It provides the illusion that one source statement is executed at a time and gives the expected values of the variables. The debugger executes the object statements in the expected source order to give expected values of the variables. As optimizations reorder, duplicate and eliminate source statements, it is not possible to execute the statements in the expected source order without making changes to the data layout and code segment of the program.

The debuggers built using this approach use several techniques to provide the expected results. They constrain the number of locations where the debugger can be invoked, constrain some of the optimizations, add instrumentation code and disturb the data layout and code segment of the object code. These debuggers are invasive, because they disturb the data layout and code segment of a program.

Patil et al. [12] proposed a new framework for debugging optimized code. They showed that with perturbations to data layout and code segment one can provide expected behavior for all optimizations including global optimizations. This scheme did not allow the value of the variable to be changed at debug time. When the user sets a break point at a source statement, the debugger takes control of the program early before executing any of the statements that have to be executed after this statement according to the source execution order. After this, it emulates only those instructions that have to be executed before the statement according to the source execution order and reports that the break point has occurred. Because all the statements that have to be executed before the break point have executed and none of the statements that have to be executed after it have executed, the state of the program is the expected state and thus provides the expected behavior.

9.11.2.3 Exposing Optimizations to the User

In this approach all the optimizations performed are exposed to the user in source-level terms. The debugger shows a source-level view of the optimized program. This approach is especially useful when the debugger performs loop optimizations such as loop peeling, loop interchanging and loop tiling. This approach has many disadvantages. The user is expected to have full knowledge of optimizations. There are some optimizations, such as instruction scheduling and register allocation, with effects that are difficult to express in source level terms.

9.11.2.4 Providing Truthful Behavior

In this approach the debugger detects when optimizations do not affect debugging and hides the optimizations. When the debugger is unable to hide the optimizations, it exposes the optimizations to the user. When optimizations are to be exposed to the user, this approach manages the effects of the optimizations by relating runtime values in the optimized code to source level values. The burden of analysis falls on the debugger and less knowledge of the optimizations is required on the part of the debugger user. Hennessey [9], Zellweger [23], Ruscetta et al. [4], Copperman [3] and Reza [17] use this approach.

9.12 Data Value Problems

Data value problems are problems arising in reporting the value of a source variable at a break point. Optimizations change the order of execution of the statements. Changing the order of execution of source assignment statements causes variables to have different values from the expected values.

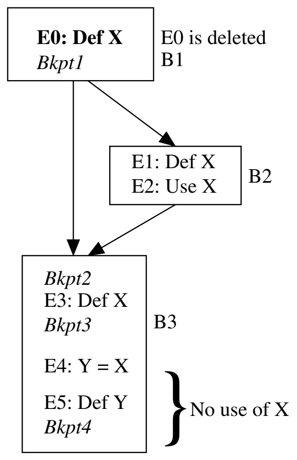


FIGURE 9.7

When the user asks the debugger to display the value of a variable, the debugger reports the state of the variable as nonresident, noncurrent, current or suspect:

- At a break point, a variable is said to be nonresident if there is no runtime location containing the value of the variable and hence no runtime value for the variable. For example, in Figure 9.7, no uses of X exist after $E4$. Thus, the same register is shared by X and Y. The next definition of Y after $E4$ assigns a new value of Y to the register shared by both X and Y. Hence, no runtime value exists for X and it is nonresident at a break point *Bkpt*4 set after $E5$.
- At a break point, a variable is said to be noncurrent if it is resident and the value of the variable is not the expected value. For example, in Figure 9.7 $E0$ is dead code and is eliminated. The value of X present at *Bkpt*1 is not the expected value and hence X is noncurrent.
- A variable is said to be current if the variable is resident and the value is the same as the expected value. At break point *Bkpt*3 the value of X is the expected value. Hence, X is current.
- A variable is said to be suspect if it is resident and the debugger cannot determine if it is current or noncurrent. In Figure 9.7 the variable X is current at *Bkpt*2 if the path taken is $B1$, $B2$ and $B3$. X is noncurrent if the path followed is $B1$ and $B3$. Variables that are suspect and noncurrent are called *endangered* variables.

The data value problem can be subdivided into the problem of finding nonresident variables and the problem of finding endangered variables at a break point.

9.12.1 Detecting Nonresident Variables

A variable is said to be nonresident at a break point if no runtime location corresponds to a variable at that break point, or multiple runtime locations contain the value of the variable and the debugger are unable to decide which location contains the expected value of the variable. Reza [17] proposed an approach to detect nonresident variables. When no optimizations are performed, all variables are assigned unique memory locations except for register variables that are stored in registers. The mapping from variables to runtime locations is one to one. Optimizations such as register allocation and register coalescing disturb this one-to-one mapping and cause nonresident variables.

Register allocation tries to use registers to store the values of frequently accessed variables, constants and temporaries to reduce their access time. We concentrate only on source variables, because only these variables are visible at the source level. The register allocator takes the live ranges of the variables and assigns registers to these live ranges using some heuristics. Some register allocation methods split the live ranges of variables and assign each split range a register or a memory location independent of other live ranges of the same variable. Additional code is added to transfer runtime value from runtime location of one split range to the runtime location of another split range when control moves from one to the other.

The variables that are not allocated a register are allocated a memory location on the activation stack. Sharing of the memory locations by nonoverlapping live ranges reduces the size of the activation stack. Splitting the live ranges of a variable makes variable to runtime location mappings one to many. Assigning the same memory location and register to different nonoverlapping live ranges makes variable to runtime location mappings many to one.

Register coalescing [15] eliminates move instructions from one variable to another by allocating the same register to both these variables if the live ranges of these variables are nonoverlapping. Register coalescing makes variable-to-runtime location mapping many to one and causes nonresident variables.

9.12.2 Detecting Nonresident Variables Caused by Register Allocation and Register Coalescing

A variable is said to be nonresident outside the live range of that variable. This can be detected by looking at the live range information in the symbol table. This approach is simple but misses some opportunities. A storage location allocated to a variable holds the value of the variable during the live range of the variable. Even after the live range of a variable, the storage location allocated to it holds the value of the variable until a new variable is assigned to this location. A definition (assignment to a variable) is called an *evicting definition* if it assigns the value of the variable to a location that previously contained the value of another variable.

A location may contain the value of a variable even after the live range of that variable. Improved response can be given to the user if all evicting definitions are found. A variable *V* whose value is found in location *L* is resident even after the live range of *V* until an evicting definition assigns to location *V*. In Figure 9.8(c) register *r*2 is used to store the value of *y*. The live range of *y* is from *S*2 to *S*4. The value of *r*2 is changed by statement *S*6. The value of *y* is present in *r*2 until *S*6, which is outside the live range of *y*. The evicting definition is *S*6.

A variable *V* is resident in location *L* at a break point *B* if an assignment to *V* exists along all paths from the start of the program to *B*, the value of the variable is stored in *L*, and no evicting definition exists for location *L* after the last occurrence of assignment to *V*. Data flow analysis is performed

```
S1 :    x = 2          s1 = 2          r1 = 2
S2 :    y = 4          s2 = 4          r2 = 4
S3 :    w = x * y      s3 = s1 * s2    r3 = r1 * r2
S4 :    z = x + 1      s4 = s1 + 1     r3 = r1 + 1
S5 :    u = x * 2      s5 = s1 * 2     r1 = r1 * 2
S6 :    x = z * 4      s1 = s2 * 4     r2 = r3 * 4

        (a)            (b)             (c)
```

(a) A simple example of register allocation; (b) symbolic register assignment for it; (c) an allocation for it with three registers assuming y and w are dead on exit from this code

FIGURE 9.8 Sample code to show evicting definitions.

to get the list of variables and corresponding runtime locations that reach a particular point in the program.

9.12.2.1 Data Flow Analysis

A residence is a tuple $\langle V, L\rangle$. It indicates that a variable V is stored at location L. A basic block B generates a residence $\langle V, L\rangle$ if an assignment to V exists in the block B and no evicting definitions of L exist after the last such occurrence of assignment to V. A basic block kills a residence $\langle V, L\rangle$ if an evicting definition of location L is in the block.

DEFINITION 9.1 (AvailResGen). *AvailResGen indicates the set of residences generated by a basic block.*

DEFINITION 9.2 (AvailResKill). *AvailResKill indicates the set of residences killed by a basic block.*

DEFINITION 9.3 (AvailResIn). *AvailResIn indicates the set of residences that reach a basic block.*

DEFINITION 9.4 (AvailResOut). *AvailResOut indicates the set of residences that are available after a basic block.*

Initially *AvailResGen* and *AvailResKill* sets are computed for all the basic blocks. The sets *AvailResIn* and *AvailResOut* of all basic blocks are found by solving the data flow Equations (9.1) and (9.2):

$$AvailResIn(I) = \bigcap_{J \in Pred(I)} AvailResOut(J) \tag{9.1}$$

$$AvailResOut(I) = (AvailResIn(I) - AvailResKill(I)) \bigcup AvailResGen(I) \tag{9.2}$$

An iterative approach is used to solve the data flow equations. The algorithm for finding the *AvailResIn* and *AvailResOut* sets of a basic block given next takes *AvailResKill* and *AvailResGen* sets of each basic block as input and generates *AvailResIn* and *AvailResOut* sets of each basic block:

```
for each basic block B do $AvailResOut[B] = AvailResGen[B]$
change = true
while change do begin
        change = false
        for each basic block B do begin
                $AvailResIn(B) = \bigcap_{J \in Pred(B)} AvailResOut(J)$
                oldout = $AvailResOut[B]$
                $AvailResOut(B) = (AvailResIn(B) - AvailResKill(B)) \bigcup AvailResGen(B)$
                if $AvailResOut[B] \neq$ oldout then change = true
        end
end
```

9.12.3 Detecting Endangered Variables

Optimizations that affect assignments to source variables cause endangered variables. Reza gives an approach [17] to detect endangered variables.

9.12.3.1 Optimizations That Cause Endangered Variables

Optimizations that eliminate and move source assignment statements cause endangered variables.

9.12.3.2 Code Elimination

Optimizations such as dead code elimination delete assignments to source variables. Eliminating assignment statements causes endangered variables. The user expects the variable to have the value that would have been assigned by the deleted assignment statement, whereas the actual value is different. The programmer does not generally add dead code; it results from performing optimizations such as constant propagation, copy propagation or induction variable strength reduction.

9.12.3.3 Code Motion

Optimizations such as loop invariant code motion, code hoisting and instruction scheduling move statements from one location to another location. Moving the statements causes the actual execution order to be different from the expected execution order. Code motion optimizations move code either in the direction of control flow or in the direction opposite to control flow, creating endangered variables.

Two key transformations, code hoisting and dead code elimination, capture data value problems caused by all the global transformations.

9.12.3.4 Optimizations That Do Not Cause Endangered Variables

Some optimizations do not cause endangered variables. Optimizations such as constant folding, induction variable strength reduction and copy propagation do not affect assignments to source variables. These optimizations create new opportunities for dead code elimination indirectly contributing to creation of endangered variables. Optimizations that affect assignments to compiler temporaries do not cause any endangered variables.

9.12.3.5 Endangered Variables Caused by Instruction Scheduling

Instruction scheduling reorders and interleaves the execution of source statements. Program semantics defines an ordering on the execution of the statements, which is the same as the expected execution order. Instruction scheduling moves assignment statements and causes endangered variables. Endangered variables occur due to either early execution or delayed execution of source assignment statements. At a break point B, if assignment to a variable V has been executed early, then the variable is said to be a *roll back variable* indicated by BackVar(V,B). At a break point B, if assignment to a source variable V is delayed to be executed after the break point, then V is called a *roll forward variable* indicated by ForwardVar(V,B).

Endangered variables can be detected by finding all source assignment statements that have executed out of order. The debugger maintains source execution order and object execution order to find the assignment statements that have executed out of order. Source execution order refers to the order in which the source assignment statements have to be executed. Each source assignment statement S is assigned a sequence number seq(S) to indicate the source execution order. For all source assignment statements that immediately follow a source assignment statement S_i the sequence number seq(S_i) + 1 is assigned. For two statements S_1 and S_2, if seq(S_1) $<$ seq(S_2), then S_1 has to execute before S_2 and if seq(S_1) $>$ seq(S_2) then S_1 has to execute after S_2. If seq(S_1) = seq(S_2), then no fixed ordering is between S_1 and S_2.

DEFINITION 9.5 (source assignment descriptor). *Source assignment descriptor for an object instruction O is a tuple $\langle I, N, V\rangle$ where I is the object instruction, N is the sequence number of the source instruction for which O is generated and V is the variable that is assigned in O.*

DEFINITION 9.6 (INST()). *For a source assignment descriptor D, INST(D) returns the object instruction in the descriptor.*

Definition 9.7 (seq()). *For a source assignment descriptor D, seq(D) returns the sequence number of the source instruction for which the object instruction of D is generated.*

Definition 9.8 (VAR()). *For a source assignment descriptor D, VAR(D) returns the variable that is assigned by the object instruction of D.*

Object execution order refers to the order in which the instructions generated for the source assignment statements execute. For each object instruction O that corresponds to a source assignment statement, a source assignment descriptor is added.

A break point B is represented by $\langle S, O\rangle$ where S is the source statement where the break point is set in the source and O is the object instruction where the break point is mapped in the object code. All roll back and roll forward variables at a break point are found and by using this information it can be determined whether a variable is endangered at the break point.

A variable is said to be a roll forward variable at a break point $B\langle S, O\rangle$ if an assignment with assignment descriptor D that has to be executed before the break point according to the source execution order (seq(D) $<$ seq(S)) is slated to be executed after the break point (INST(D) $\geq$ O). Equations (9.3) and (9.4) can be used to find the roll back and roll forward variables, as follows:

$$((seq(D) < seq(S))\&(INST(D) >= O)) => ForwawdVar(VAR(D), B) \tag{9.3}$$

$$((seq(D) > seq(S))\&(INST(D) < O)) => BackVar(VAR(D), B) \tag{9.4}$$

9.12.3.6 Endangered Variables Caused by Local Optimizations

Local optimizations cause endangered variables by eliminating assignment statements. Assignment statements can be eliminated if an assignment is already available or an assignment is a dead assignment.

An assignment to a variable V is called an *available assignment statement* if on all paths from the starting point of the program to this point, an assignment statement is the same as this and no further assignments are made to V and all other variables present in the assignment statement after the last such occurrence of the assignment to V. Elimination of available assignment statements does not cause any endangered variables, because the deleted assignment statement assigns a value that is already available.

9.12.3.7 Elimination of Dead Assignments

An assignment statement S that assigns to a variable V is a dead assignment statement if there is no use of V until the next definition of V or the end of the program. Elimination of dead assignment statements causes endangered variables. The variable is expected to have the value that would have been assigned by the deleted assignment statement. At all break points set between the deleted statement and the next assignment to the variable V, the value of V is different from the expected value.

In the code given in Figure 9.9 the assignment statement S_I is dead as there are no uses of x in statements S_{I+1} to S_{J-1}. Eliminating S_I makes x endangered at break points set at statements S_{I+1} to S_{J-1}. For every dead assignment statement A that is deleted, a dead assignment descriptor $\langle I, N, V\rangle$ is defined. In the descriptor I is the object instruction corresponding to the instruction A', which is the next assignment statement that assigns to V after A, N is the sequence number of the

```
          ...
SI:    X = I + J
          ...              ...
SJ:    X = I * J     Ij:   Rx = Ri * Rj
          ...              ...

                           After useless code elimination
       Sample code         and code generation

          (a)              (b)
```

FIGURE 9.9 Sample code to demonstrate effects of dead code elimination.

source instruction A, and V is the variable assigned in A. At a break point $B\langle S, O\rangle$ a dead assignment descriptor D creates an endangered variable if the following conditions hold:

1. The source statement where the break point is set comes after the dead assignment statement according to the source execution order.
2. The object instruction generated for the next source assignment that assigns to the same variable as dead assignment comes after the object instruction where the break point is mapped.
3. The variable is not a roll forward variable at the break point.

These conditions are checked by:

$$((seq(D) < seq(S))\&(INST(D) \geq O)\& \sim ForwardVar(Var(D), B))$$
$$\Rightarrow EndangeredDead(Var(D), B)$$

All the endangered variables due to dead code elimination at a break point are found and the information is used to find whether a variable is endangered due to local dead code elimination at the break point.

9.12.3.8 Detecting Endangered Variables Caused by Global Optimizations

Optimizations such as code hoisting, code sinking, loop invariant code motion, global dead code elimination and global instruction scheduling move and delete source assignment statements causing data value problems. Concentrating on two key transformations code hoisting and global dead code elimination captures data value problems caused by all the global optimizations.

9.12.3.9 Detecting Endangered Variables Caused by Code Hoisting

Code hoisting is a form of partial redundancy elimination. Adding expressions at some places to make other partially redundant expressions fully redundant is called *code hoisting* [1]. Code hoisting moves expressions in the direction opposite to the control flow. Some of the assignment statements are executed earlier than expected, causing endangered variables. Expressions that are added by code hoisting are called *hoisted expressions*. Expressions that become redundant after code hoisting and are deleted are called *redundant expressions*.

An example in Figure 9.10 shows how code hoisting affects debugging. In the sample code shown, expression $X = U - V$ is executed twice if the path followed is $B0$, $B1$ and $B3$ and the same expression is evaluated once when the path is $B0$, $B2$ and $B3$. Expression $E2$ is partially redundant. As part of code hoisting the expression $X = U - V$ is moved ahead into block $B2$, making partially redundant expression $E2$ fully redundant and a candidate for further optimizations. Expression $E2$ becomes an available expression after code hoisting and is deleted.

If a break point is set after $E3$ in $B2$, then the value of X is noncurrent because the value that is present is the value assigned by $E3$ whereas the expected value is the value assigned by $E0$. If a break

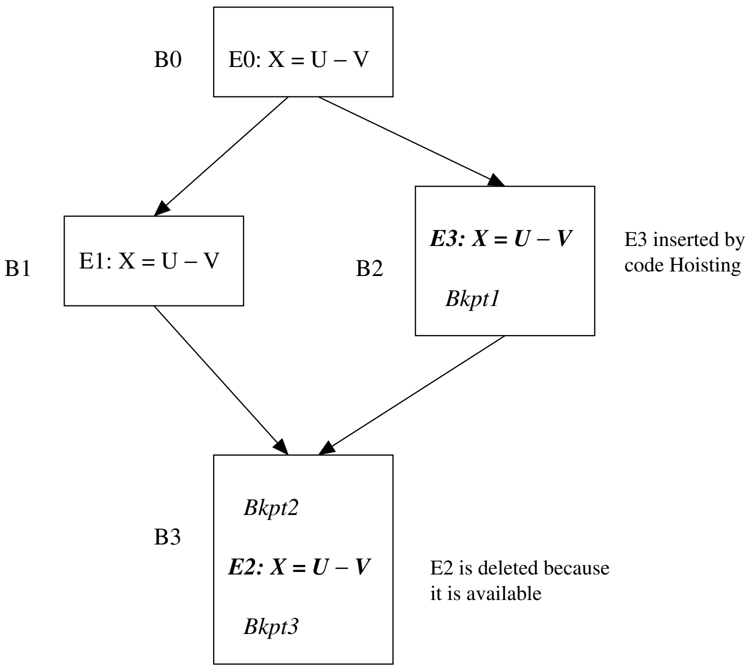


FIGURE 9.10 How code hoisting affects debugging.

point is set before $E2$ in $B3$ then the variable X is current if the path followed is $B0$, $B1$ and $B3$ and the variable X is noncurrent if the path followed is $B0$, $B2$ and $B3$ forcing the X to be suspect. This information is summarized as follows:

Break Points Set at	State of X
Bkpt1	X is noncurrent
Bkpt2	X is suspect
Bkpt3	X is current

Code hoisting causes both suspected and noncurrent variables. When we set a break point after $E3$ in $B2$, the variable x is noncurrent. The value present in x is the value that is assigned by $E3$, whereas the expected value is the value that is assigned by $E0$. For every hoisted expressions E_h a hoist descriptor is added containing the information of the variable that is assigned by E_h, the corresponding redundant expression E_r and the line number of the source instruction for which E_r is generated. For all redundant expressions E_r that are deleted by code hoisting, a redundant descriptor is added containing the information of line number of the source instruction to which it corresponds and the variable that is assigned.

All hoist expressions that reach a particular instruction are found using global data flow analysis similar to reaching definition analysis. After finding the data flow information, a variable V is said to be noncurrent at a break point if on all paths from the starting point of the program to this break point there is a hoisted expression that assigns to V and the corresponding redundant expression is not executed. If the preceding case holds for some paths, then the variable V is said to be suspect.

9.12.3.10 Data Flow Analysis

Definition 9.9 (HoistGen). *HoistGen indicates the hoisted expressions generated by a block.*

Definition 9.10 (HoistKill). *HoistKill indicates the hoisted expressions killed by a block.*

Definition 9.11 (AllHoistReachIn). *AllHoistReachIn indicates the hoisted expressions that reach a basic block along all paths from the starting of the program to the basic block.*

Definition 9.12 (AllHoistReachOut). *AllHoistReachOut indicates the hoisted expressions that are present after a basic block.*

Definition 9.13 (AnyHoistReachIn). *AnyHoistReachIn indicates the hoisted expressions that reach a basic block along any paths from the starting of the program to the basic block.*

Definition 9.14 (AnyHoistReachOut). *AnyHoistReachOut indicates the hoisted expressions that are present after a basic block.*

A basic block generates a hoisted expression E_h, assigning to V if a hoisted expression exists in the basic block and no other assignment statement, another hoisted expression or a redundant expression assigns to V. HoistGen(B) indicates the hoisted expressions generated by block B. A basic block kills a hoisted assignment statement if a statement or a redundant expression assigns to the same variable as that of the hoisted expression. HoistKill(B) indicates the hoist expressions that are killed by block B. The iterative method similar to the one presented earlier is used to solve the following data flow equations:

$$AllHoistReachOut(I) = (AllHoistReachIn(I) - HoistKill(I)) \bigcup HoistGen(I) \tag{9.5}$$

$$AnyHoistReachOut(I) = (AnyHoistReachIn(I) - HoistKill(I)) \bigcup HoistGen(I) \tag{9.6}$$

$$AllHoistReachIn(I) = \bigcap_{J \in Pred(I)} AllHoistReachOut(J) \tag{9.7}$$

$$AnyHoistReachIn(I) = \bigcup_{J \in Pred(I)} AnyHoistReachOut(J) \tag{9.8}$$

9.12.3.11 Detecting Endangered Variables Caused by Global Dead Code Elimination

Dead code elimination deletes all assignment statements that assign to a variable if the value that is assigned in this assignment is not used later. If an assignment statement that assigns to V is deleted, then the variable V is noncurrent until the next definition of V. The following example shows how dead code elimination affects debugging.

In the sample piece of code given in Figure 9.11, expression $E0$ is dead because there are no uses of x until the next definition of x. At a break point set after $E0$ in $B0$, variable x is noncurrent. At a break point set in block $B1$ variable x is noncurrent. At a break point set before $E2$ in $B2$ x is noncurrent. At a break point set after $E2$ in $B2$ x is current. At a break point set before $E1$ in $B3$, x is current if the path followed is $B0$, $B2$ and $B3$; and x is noncurrent if the path followed is

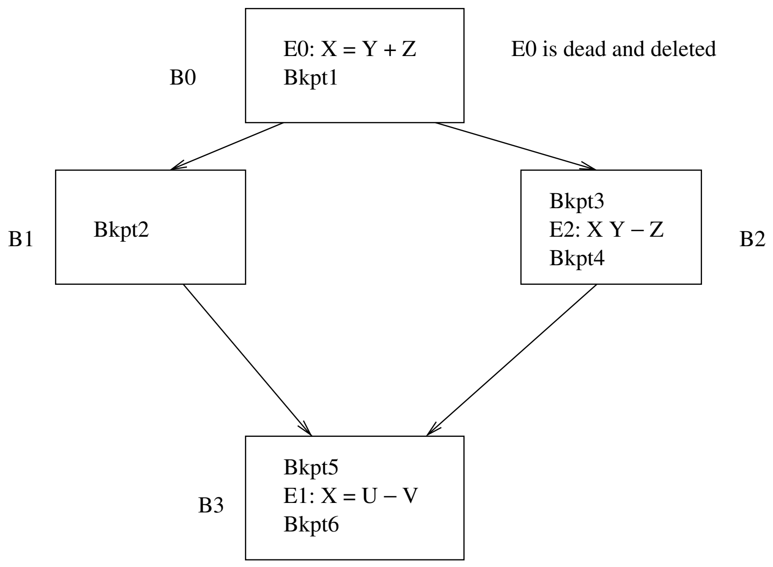


FIGURE 9.11 Sanple piece of code.

*B*0, *B*1 and *B*3 making *x* suspect. If a break point is set after *E*1 in *B*3, then *x* is current as the value of *x* is the value assigned by *E*1. The effect is summarized in the following:

Break Points Set at	State of *X*
Bkpt1	*X* is noncurrent
Bkpt2	*X* is noncurrent
Bkpt3	*X* is noncurrent
Bkpt4	*X* is current
Bkpt5	*X* is suspect
Bkpt6	*X* is current

To determine whether a variable *V* is endangered due to dead code elimination at a break point it is found if a dead assignment statement existed along the path and no other definition of *V* has occurred after the last such dead assignment statement. For each assignment statement that is eliminated due to dead code elimination a dead assignment descriptor is added containing the information of the variable assigned, and the sequence number of the source assignment statement corresponds to the deleted assignment statement. All dead assignment statements that reach a particular point along all paths are found to say whether a variable is noncurrent due to dead code elimination. All dead assignment statements that reach a particular break point along any paths are found to say whether a variable is suspect due to dead code elimination.

Global data flow analysis is used to find the required sets. The data flow analysis similar to that for finding endangered variables due to code hoisting is used.

9.13 Code Location Problem

Optimizations such as loop invariant code motion, dead code elimination and instruction scheduling insert, delete and move code and make mapping of break points difficult. The debugger has to correctly map the break points and check that the break points are triggered the expected number of times and at expected source locations. Optimizations that involve code elimination, code duplication and code motion affect mapping of the break points. The object code locations where the source break points are mapped influence the number of variables that are nonresident and endangered at a break point.

The debugger uses source-to-object mapping and object-to-source mapping to map the source break points and report asynchronous break points. Compilers generally use different intermediate representations to perform different optimizations. Initially, when the source program is compiled and converted into some intermediate representation, the compiler has to construct the source-to-object and object-to-source mappings. The compiler updates the mappings when optimizations are performed. When the intermediate representations are lowered, the compiler propagates the mappings to the lower level until the final object code is obtained.

9.13.1 Code Elimination

Optimizations like dead code elimination remove a source statement completely. No object code corresponds to the deleted source statement. When the user sets a source level break point at a statement that has been removed, the debugger has to determine the appropriate location in the object code where the break point should be mapped. The following cases arise:

1. In the simple case at least one source statement occurs after the deleted statement in the same basic block. In this case the debugger sets the break point at the object code generated for the next source statement.
2. The statement that is deleted is the last statement in the basic block. In this case the debugger maps the break point to the starting instruction of all the successor basic blocks of the basic block containing the deleted statement. The break point is reported more than the expected number of times.
3. The statement that is deleted is the only statement in the basic block and the whole basic block is deleted. In this case the break point is set at the starting instructions of all the successor blocks.

The example given in Figure 9.12 illustrates the first two cases. Figure 9.12(b) shows the code obtained after performing dead code elimination on code given in Figure 9.12(a). *S*3 is the only statement present in block *B*2. After dead code elimination block *B*2 is completely deleted. If the

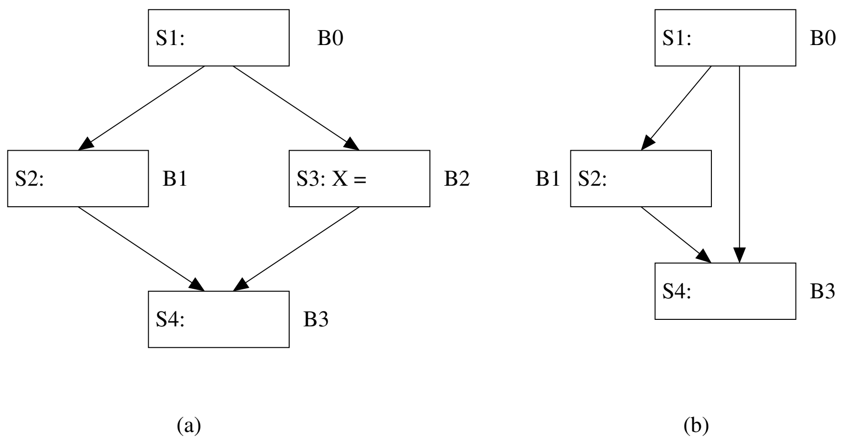


FIGURE 9.12 Code elimination; (b) shows the code obtained after performing dead code elimination on code given in (a).

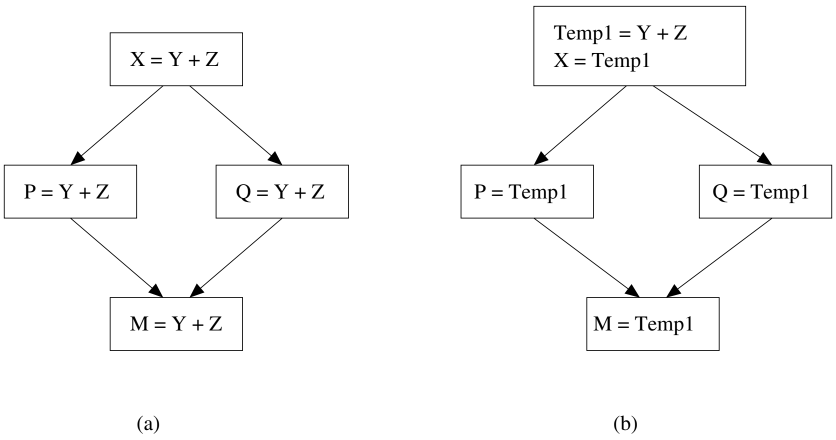


FIGURE 9.13 Common subexpression elimination is performed on the code shown in (a), resulting in the code shown in (b).

user sets a break point at $S3$, and the debugger maps the break point to the starting instruction of block $B3$, then the break point is triggered more than the expected number of times. The break point is triggered whenever control reaches block $B3$ but according to the source semantics the break point should be triggered only when the control reaches $S3$. We can set a conditional break point based on the condition in $S1$ such that the break point is triggered the expected number of times. When code elimination is performed, the compiler updates the source-to-object and the object-to-source mappings to reflect the changes introduced by code elimination.

9.13.2 Code Insertion

Optimizations such as code hoisting and common subexpression elimination insert new code. The code that is added should be corresponding to a source statement so that if an asynchronous break occurs at the newly added code, then the break can be reported at the appropriate source statement. Runtime exceptions and user signals are called asynchronous breaks. The compiler updates the source-to-object and object-to-source mapping by corresponding the newly added code to some source statement.

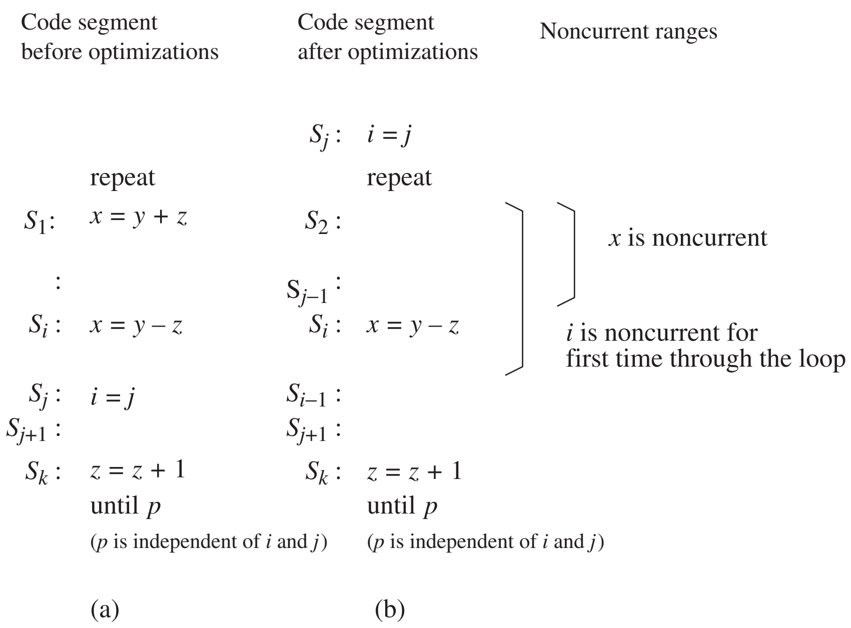
In the example shown in Figure 9.13 common subexpression elimination is performed on the code shown in Figure 9.13(a), resulting in the code shown in Figure 9.13(b). The statement $Temp1 = y + z$ is the newly added statement. When an asynchronous break occurs while evaluating $y + z$ in $Temp1 = y + z$, the debugger reports that an asynchronous break has occurred at statement $S1$ in the source.

9.13.3 Code Motion

Many optimizations such as loop invariant code motion and partial redundancy elimination move code from one location to another location. When code motion is performed, the compiler updates the source-to-object and object-to-source mappings. When a break point is set at a statement S that is moved by code motion, then the debugger can map the break point at:

1. The object code corresponding to the moved statement
2. The object code generated for the statement after the moved statement
3. The object code generated for the statement immediately preceding the moved statement

The choice of where the break point is mapped influences the number of locations that are endangered and noncurrent.


Code segment before optimizations
Code segment after optimizations
Noncurrent ranges
x is noncurrent
i is noncurrent for first time through the loop
(a)
(b)

```
                                    Sj:  i = j
         repeat                          repeat
   S1:   x = y + z                  S2:
    :                             Sj-1:
   Si:   x = y - z                  Si:  x = y - z
   Sj:   i = j                    Si-1:
 Sj+1:                            Sj+1:
   Sk:   z = z + 1                  Sk:  z = z + 1
         until p                         until p
         (p is independent of i and j)   (p is independent of i and j)
```

FIGURE 9.14 Example to show how break point mapping affects data value problem.

When the debugger sets the break point at statement S_j in Figure 9.14(b), the break point occurs only once and all variables assigned in statements S_1 to S_{j-1} are noncurrent. When the debugger sets the break point at statement S_{j-1} in Figure 9.14(b), the break point is triggered the expected number of times but variable i is noncurrent at the break point. If the break point is set at statement S_{j+1} in Figure 9.14(b), then the break point is triggered the expected number of times and the variable i is current. The effect is summarized in the following:

Break Point in Figure 9.14(a)	Break Point in Figure 9.14(b)	Number of Times Break Point Is Triggered	Endangered Variables at Break Point
S_j	S_j	Once	Variables assigned in S_1 to S_{j-1}
S_j	S_{j-1}	n Times	Variables assigned in S_j
S_j	S_{j+1}	n Times	None

9.13.4 Instruction Scheduling

Instruction scheduling interleaves the object code generated for different source statements. Due to this, the object instructions generated for a source statement are not consecutive. The object instructions generated for a source instruction are split into different ranges and object instructions corresponding to any source statement can come in between these ranges. If the user sets a break point on an instruction with object instructions that are split into different ranges, the debugger faces the problem of where to map the break point.

The debugger can set the break point at the first object instruction generated for the source statement or at the first side-effecting instruction generated for an instruction. Generally, assignments, function calls and branches are treated as side-effecting instructions. Because instruction scheduling interleaves the object instructions corresponding to source statements, if a break point is set, then it cannot be said whether all the instructions that have to execute before the break point have executed, or no instructions that have to execute after the break point have not executed.

9.14 Value Change Problem

While debugging unoptimized code, source level debuggers allow the value of a variable to be changed at debug time. When optimizations are performed, it is unsafe to change the value of a variable at some break points in the program. Changing the value of a variable at some locations may lead to unexpected and erroneous results. Many source level debuggers that support debugging optimized code do not allow the value of a variable to be changed when debugging optimized code.

When the compiler performs optimizations based on the value of a variable and relation between the values of variables, the debugger should not allow the user to change the value of a variable such that it violates the properties based on which the optimizations are performed. The debugger has to reevaluate some of the expressions when the value of a variable has to be changed.

The values of variables that are nonresident at a break point cannot be changed at debug time. Values of some endangered variables can be changed at debug time. A value change problem refers to the problem of finding whether it is safe to change the value of a variable at a particular break point and changing the value of the variable without affecting the semantics of the program. Allowing the user to change the value of a variable requires the compiler to pass additional information to the debugger. The debugger has to perform additional analysis to change the value of a variable. Allowing the debugger to change the value of a variable is useful in many situations. We enumerate the situations where it is beneficial to allow value change at debug time. Some optimizations such as constant folding that do not affect the data value problem and code location problem affect the value change problem.

9.14.1 Advantages of Allowing Value Change at Debug Time

1. At a particular break point if it is known that the value of a variable is causing an error, the value of the variable can be changed and then one can proceed to check whether the rest of the program is working correctly without going through the edit-compile-debug cycle. Thus, the effective debugging time can be reduced.
2. Allowing the debugger to change the value of a variable allows the user to traverse all paths of the program and check that each path is working correctly. This is very useful in cases where it is difficult or impossible to change the inputs.

9.14.2 Constant Propagation and Constant Folding

In constant propagation, for all statements of the form $x = c$ the compiler replaces the later uses of x with the value of c. The statement $x = c$ is a candidate for dead code elimination. In constant folding expressions all of whose operands are constants are evaluated at compile time and the expression is replaced with the result of the evaluation. Constant propagation along with dead code elimination causes endangered variables.

When applied in isolation, constant folding does not affect debugging. When constant folding is applied on a constant expression where one of the operands is a constant propagated variable, the value of that variable cannot be changed. If the value of a variable that is constant propagated is changed, the changed value of the variable is not reflected in the expression where the variable is constant propagated. When the user wants to change the value of a variable that is constant propagated and one of the instances is constant folded, the debugger has to reevaluate the expression that is folded

```
S1:  X = 38
S2:  IF(P)
     {
S3:    P = X * 4
     }
     ELSE
     {
S4:    M = X * Y
     }
```

Sample code

(a)

```
S1:  X = 38
S2:  IF(P)
     {
S3:    P = 38 * 4
     }
     ELSE
     {
S4:    M = 38 * Y
     }
```

Sample code after
Constant Propagation

(b)

```
S1:  X = 38
S2:  IF(P)
     {
S3:    P = 152
     }
     ELSE
     {
S4:    M = 38 * Y
     }
```

Sample code after constant
Propagation and Constant Folding

(c)

FIGURE 9.15 Effects of constant folding and propagation on value change problem.

using the new value of the variable and replace the uses of the constant folded expression with the changed value.

In Figure 9.15, the sample code in Figure 9.15(b) is the result of applying constant propagation to the code given in Figure 9.15(a), and Figure 9.15(c) is the result of applying constant propagation and constant folding to the code given in Figure 9.15(a). If the user sets a break point before statement $S2$ and wants to change the value of x at that break point, then the debugger has to reevaluate the expression in $S3$ with the changed value of x. The compiler must pass all the expressions that are constant folded to the debugger.

DEFINITION 9.15 (constant propagation definition descriptor). *Constant propagation definition descriptor is a tuple* $\langle N, V,$ *Value,const-prop-id*$\rangle$ *where N is the line number of the source instruction, V is the variable that is assigned, Value is the constant value that is assigned and const-prop-id is the unique identifier assigned for each constant propagation definition descriptor.*

DEFINITION 9.16 (constant propagation descriptor). *Constant propagation descriptor is a tuple* $\langle$ *N,const-prop-id*$\rangle$ *where N is the line number of the source instruction for which this is generated and const-prop-id indicates the identifier of the corresponding constant propagation definition descriptor.*

For every assignment expression that is constant propagated, a constant propagation definition descriptor is stored. For each expression where constant propagation is performed, a constant propagation descriptor is added.

For all constant folded expressions that contain a constant that is the result of constant propagation, the expression is stored and passed on to the debugger. If the user wants to change the value of a variable at a break point, then the debugger checks whether any constant propagation definition descriptors involving this variable reach the break point. If such constant definitions exist, then the debugger checks whether any of the instances where this is constant propagated are involved in constant folding. If so, then the debugger has to reevaluate the expression and replace the old value with the new value. Reaching definitions analysis is used to find all constant propagation definition descriptors that reach an instruction.

If the compiler performs constant propagation and constant folding recursively, then the amount of the information that must be stored grows exponentially.

```
S1:  X = Y              S1:  X = Y
     . . .                   . . .
S2:  use X              S2:  use Y
     . . .                   . . .
S3:  use Y              S3:  use Y
     . . .                   . . .
S4:  use X              S4:  use Y
     . . .                   . . .
S5:  def X              S5:  def X

Sample Code             Sample code after
                        Copy Propagation

    (a)                      (b)
```

FIGURE 9.16 Effects of copy propagation on value change problem.

9.14.3 Copy Propagation

Copy propagation affects the value change problem. The debugger cannot directly change the runtime value of a variable that is involved in copy propagation. If the debugger changes the runtime value of a variable that is copy propagated, then the changed value is not reflected in the statements where it is copy propagated and the uses of the variable in these statements are replaced with uses of another variable.

In the sample code shown in Figure 9.16 if the user sets a break point before $S2$ and wants to change the value of variable x, just changing the runtime value of the variable cannot suffice, because the changed value is not reflected in statements $S2$ to $S5$ as the uses of x in these statements are replaced by uses of y. If the user wants to change the value of y at a break point set after $S2$, then the debugger should not directly change the runtime value of y as the uses of x in statements $S2$ and $S4$ are replaced with y.

The compiler gathers more information about the optimizations and propagates it to the debugger. The additional information is used by the debugger to change the value of a variable. For all copy statements that are the cause of copy propagation, a copy propagation definition descriptor $\langle N, V1, V2\rangle$, where N is the line number of the source instruction, *V1* is the variable that is assigned and *V2* is the variable whose value is assigned, is stored. For all statements where copy propagation is done a copy propagation descriptor ⟨*N, Copy-def-id*⟩ is stored, where N is the line number of the source instruction for which this instruction is generated and *Copy-def-id* is the unique identifier for the copy instruction that is the cause of this copy propagation.

At a break point, if the user wants to change the value of a variable, then it is checked whether the variable is involved in copy propagation. Two cases arise. First, the variable that has to be changed is the value that is assigned; second, the variable that has to be changed is the variable whose value is assigned. In the first case, for all the statements where this variable is replaced with another variable, the statement is changed so that it uses the new and changed value of the variable and the runtime value of the variable is changed. The debugger uses code patching to do this. In the second case where the variable to be changed is on the right-hand side of the copy statement, the uses of the variable that are copy propagated instances of another variable are replaced with uses of the old value of the variable, and the runtime value of the variable is changed.

9.14.4 Common Subexpression Elimination

Common subexpression elimination affects the value change problem. When the user wants to change the value of a variable that is involved in a common subexpression, the debugger has to reevaluate the subexpression and change the runtime value of the temporary as well as the runtime value of the variable to be changed.

DEFINITION 9.17 (cse assignment descriptor). *The cse assignment descriptor is a tuple* $\langle N, Id, Temp, V_1, \ldots, V_n \rangle$ *where N is the source line number to which the subexpression corresponds, Id is the unique identifier value that is assigned to each subexpression, Temp is the variable that is used to store the result of the subexpression and* V_1 *to* V_n *are the variables involved in the subexpression.*

DEFINITION 9.18 (cse descriptor). *The cse descriptor is a tuple* $\langle N, Cse\text{-}id \rangle$ *where N is the source line number of the source statement corresponding to this instruction and Cse-id is the unique identifier of the corresponding cse assignment descriptor.*

For common subexpression elimination, as in copy propagation, the optimization information is replaced in the descriptors and information is used for changing the value of a variable at debug time. When subexpression elimination is performed, at all instructions where the evaluation is done and the result is stored into a temporary variable, a cse assignment descriptor is added. At all places where the subexpression is replaced with a temporary variable, the cse descriptor is added.

At a break point, if the user wants to change the value of a variable that is involved in a subexpression, the debugger finds the cse assignment descriptor corresponding to the subexpression and reevaluates the subexpression with the changed value of the variable and updates the runtime value of the temporary variable. Reaching definition analysis is used to find all cse assignment descriptors that reach a point and check whether the variable that is to be changed is present in any one of these descriptors. If the variable is involved in any such subexpression, then the subexpression is reevaluated with a changed value of the variable and the runtime value of the temporary is modified. This requires the debugger to reevaluate the subexpression at debug time.

9.14.5 Code Motion Transformations

Optimizations such as loop invariant code motion, global instruction scheduling and code hoisting change the order of execution of the source statements and source assignment statements. Changing the order of execution of the source assignment statements causes endangered variables. Changing the order of execution of the statements affects the value change problem. Code motion moves some statements to execute before the break point. In this case, when the debugger changes the value of a variable the changed value of the variable is not reflected in the statements moved earlier.

9.15 Conclusions

This chapter presents an overview of the techniques and issues in debugging programs. Most of the issues related to debugging of programs have been addressed. A good amount of literature is available on debuggers.

However, several important research issues remain open (e.g., debugging optimized code). Although debuggers form an important tool in any program development environment, not much work has been done yet on this topic. This chapter has surveyed and shown that optimized code can be debugged by building extra data structures at compile time. We have shown how effects of optimizations can be modeled in the context of debuggers and how values of the variable that have been moved around in the code can be computed. We have also shown that it is viable to allow the values of variable to be changed, in optimized code, at debug time. The compiler has to store additional information about optimizations and pass it to the debugger. The debugger has to perform additional computations at debug time. The time taken for these computations is negligible while performing interactive debugging.

Acknowledgment

M. Sarath Kumar contributed mainly to the section on debugging optimized code.

References

[1] J.D. Ullman, A.V. Aho and R. Sethi, *Compilers: Principles, Techniques and Tools*, Addison-Wesley, Reading, MA, 1999.

[2] Compaq Computer Corporation, Ladebug Debugger Manual, March 2001; available at *ftp.compaq.com/pub/products/software/developer/.*

[3] M. Copperman, Debugging optimized code without being misled, *ACM Trans. Programming Languages Syst.*, 16(3), May 1994.

[4] M. Ruscetta, D.S. Coutant and S. Meloy, Doc: A Practical Approach to Source-Level Debugging of Globally Optimized Code, in Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, June 1988.

[5] H. Liberman, Guest Ed., The debugging scandal and what to do about it, *Commun. ACM,* 40(4), 1997.

[6] M.S. Johnson, Ed., Software engineering symposium on high level debugging, *ACM SIGPLAN Notices,* 18(8), 1983.

[7] S. Simmons, G. Brooks and G.J. Hansen, A New Approach to Debugging Optimized Code, Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, May 1992.

[8] W.C. Gramlich, Debugging methodology, session summary, *ACM SIGPLAN Notices*, 18(8), 1, 1983.

[9] J. Hennessey, Symbolic debugging of optimized code, *ACM Trans. Programming Languages Syst.*, 4(3), July 1982.

[10] Hewlett Packard, *HP wdb documentation*, 9th ed., June 2000; available at *devresource.hp.com/devresource/Tools/wdb/doc/index.html.*

[11] M.S. Kumar, Debugging Optimized Code: Value Change Problem, M.Tech thesis, Department of Computer Science, IIT Kanpur, April 2001.

[12] H. Patil, B. Olsen, W.W. Hwu, L.C. Wu and R. Mirani, A New Framework for Debugging Globally Optimized Code, Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, May 1999.

[13] J.R. Levine, *Linkers and Loaders*, Morgan Kaufmann, San Francisco, 2000.

[14] Microsoft Corporation, Visual C user manual, available in MicroSoft's Visual Studio Documentation.

[15] S.S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann, San Francisco, 1997.

[16] V. Paxson, A survey of support for implementing debuggers, Class project, University of California, Berkeley, October 1990.

[17] A. Reza, Source-Level Debugging of Globally Optimized Code, Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA, 1996; available at *cs.cmu.edu/ali/thesis.ps*.

[18] A. Reza and T. Gross, Detection and Recovery of Endangered Variables Caused by Instruction Scheduling, Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, June 1993.

[19] A. Reza and T. Gross, Source Level Debugging of Scalar Optimized Code, Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation, May 1996.

[20] S. Shebs, R. Stallman and R. Pesch, Debugging with GDB, GNU, March 2000; available at *sunsite.ualberta.ca/Documentation/Gnu/gdb-5.0/*.

[21] J.B. Rosenberg, *How Debuggers Work: Algorithms, Data Structures and Architecture*, John Wiley & Sons, New York, 1996.

[22] Sun Microsystems, Forte C Datasheet, October 2000; available at *www.sun.com/forte/c/datasheet.html*.

[23] P. Zellweger, An Interactive High-Level Debugger for Control-Flow Optimized Programs, ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging, 1983.

10

Dependence Analysis and Parallelizing Transformations

Sanjay Rajopadhye
Colorado State University

10.1 Introduction

The first impression one gets when phrases such as *dependence analysis* and *automatic parallelization* are mentioned is that of loop programs and array variables. This is not surprising, because the loop is the classic repetitive structure in any programming language, and clearly this is where programs spend a significant amount of their time. Second, because of the early impetus on high-performance computing for large numerical applications (e.g., FORTRAN programs) on supercomputers, a long

0-8493-1240-7/03/$0.00+$1.50

research effort has been underway on parallelizing such programs. The area has been active for over a quarter century, and a number of well-known texts on this topic are readily available.

Any approach to automatic parallelization must be based on the following fundamental notions: (1) detection of which statements in which iterations of a (possibly multiply nested and possibly imperfect) loop truly depend on each other, in the precise sense that one of them produces a value that is consumed by the other; (2) hence determination of which operations can be executed in parallel; and (3) transformation of the program so that this choice of parallelization is rendered explicit. The first problem is called dependence analysis, the second constitutes the additional analysis necessary to choose the parallelization and the third is called program restructuring.

In all generality, these are extremely difficult problems. Nevertheless, for certain classes of programs elegant and powerful methods are available. Therefore, instead of giving yet another survey of a vast field, we present in this chapter, a somewhat less well-known approach, called the *polyhedral model*. The model is based on a mathematical model of both the program to be parallelized and the analysis and transformation techniques used. It draws from operational research, linear and nonlinear programming and functional and equational programming. Its applicability is, however, restricted to a well-defined and limited (but nevertheless important) subset of imperative programs.

Essentially, programs in the polyhedral model are those for which the preceding three questions can be resolved systematically and optimally. This class is called *affine control loops* (ACLs). This restriction is motivated by two principal reasons:

- ACLs constitute an important (sub) class of programs. Many programs spend the majority of their time in such loops. This occurs not only in numerically intensive computations (where such loops are more or less *siné qua non*) but also in other domains such as signal and image processing and multimedia applications.
- Elegant and powerful mathematical techniques exist for reasoning about and transforming such programs. Drawing from the methods of operational research and linear and nonlinear programming, they yield provably optimal implementations for a variety of cost criteria.

Independent of how the parallelism is to be specified (notions of a language for expressing the parallelized program, its semantics and its efficient implementation on a parallel machine), two primary hurdles exist in detecting parallelism in a sequential (imperative) program. The first is the reutilization of memory — the problem of false (anti- and output-) dependences. The fact that the same memory location is used to store two distinct values implies that all computations using the first value must be executed before it is (over) written by the second, regardless of whether they are independent of the latter (and hence potentially parallelizable). Indeed, the dependence analysis step that precedes any parallelization consists essentially of identifying the true dependences between the different operations of the program.

The second aspect of imperative sequential languages that hinder parallelism detection is what is called the *serialization* of *reductions*. Reductions are operations that combine many (say, n) data values using an associative (and often commutative) operator. In an imperative sequential program such computations are serialized (often in an arbitrarily chosen manner), merely because the programming language cannot express the independence of the result on the order of evaluation. The dependence graph obtained naively from such a program is a sequence of length n, and does not allow any parallelism. However, the associativity allows us to potentially execute the computation in logarithmic time (on $n/2$ processors with only binary operators), or in time $\frac{n}{p} + \lg p$ on p processors, which is of the order $\Theta(\frac{n}{p})$ if $n \gg p \lg p$ (such a parallel implementation is thus work optimal).

The polyhedral model enables the resolution of these two problems. First of all, it allows for an exact data flow analysis of ACLs that completely eliminates false dependences, thus enabling essentially perfect dependence analysis. The result of such an analysis yields an intermediate form of the program that may be formally described as a *system of affine recurrence equations* (SARE) with variables that are defined over polyhedral domains. The polyhedral model enables a second analysis of such SAREs to detect the presence of reductions in the program.

The formalism of SAREs is interesting in and of itself, instead of a mere intermediate form for the analysis of ACLs. It constitutes a class of equational programs that are usually closer to the mathematical notions underlying the algorithms embodied in most ACLs. The notation of SAREs can be extended to express reduction operations, and this yields an even more intuitive and mathematical formalism useful for algorithm designers. Because they are equational, program development in this formalism can benefit formal methods such as program verification, synthesis by correctness-preserving transformations and abstract interpretation. The ALPHA language developed at IRISA subscribes to this view, and the tools and techniques used by the MMALPHA system for parallelizing ALPHA programs are based fundamentally on the same techniques that we describe in this chapter for ACLs.

The remainder of this chapter is organized as follows. In Section 10.2 we first present an overview of the parallelization process. Section 10.3 describes the mathematical foundations that we need. In Section 10.4 we discuss the algorithms for exact data flow analysis of ACLs. We digress in Section 10.5 to show how the notation of SAREs together with a mechanism for describing reduction operations can constitute a full-fledged equational, data parallel language (this section may be skipped at first reading). Next, Sections 10.6 through 10.8 describe the methods of static analysis of SAREs. Thet treat three important problems, namely, that of scheduling an SARE, and that of allocating the computations to processors and to memory locations, respectively. Section 10.9 then describes the problem of generating code from a scheduled and appropriately transformed SARE. Finally, we describe limitations, open problems and conclude with some bibliographic notes.

10.2 Overview of the Parallelization Process

Deriving a parallel program in the polyhedral model consists of the following steps (note that the steps may not all be performed in the order stated, and it is not essential that all steps be performed):

1. We analyse the sequential program (i.e., the ACL) to determine the corresponding polyhedral SARE.
2. We perform a second analysis pass on the SARE to detect reductions and other collective associative operations. Due to space constraints we do not give details of this analysis in this chapter.
3. A polyhedral SARE explicitly names every value computed by (i.e., the result of every operation in) the program. Hence, our next step consists of determining three functions related to its final implementation. For each operation of the SARE, we determine:

   - Execution date (i.e., a schedule)
   - Processor allocation that specifies the processor where that operation is to be executed
   - Memory address where the result is to be stored

   The schedule may impose an order on the accumulations of the intermediate results in reductions and scans. Vis-à-vis the other two functions, one often uses the convention implied by the *owner-computes* rule, namely, that a value computed by any processor is stored in the (local) memory of that processor itself. In this case, a memory allocation function defined as a pair — specifying a processor and a (local) address on this processor — is sufficient to subsume the processor allocation function. Note that two issues are related to the choice of these functions. First, they must be valid, according to certain architectural and semantic constraints. In addition, it is desirable that they be chosen so as to optimize certain cost criteria.

We will see that the polyhedral model provides us with elegant techniques to address both these issues.

4. Once these functions are chosen, we construct an equivalent SARE that respects the following conventions: certain indices of all the variables of the SARE, called the *temporal* indices, represent time (i.e., the schedule that we have chosen, others represent the processor indices and still others represent memory addresses).
5. From this transformed SARE, we produce code that (1) respects the order implied by the temporal indices, (2) has the parallelism implied by the processor indices, and (3) allocates and accesses the memory as specified by the memory indices. This code is in the form of an ACL that is now parallelized and optimized for various cost criteria (that guided the choice of the three functions in step 3, above).

10.3 Notations and Mathematical Foundations

In an ACL program, the two distinct classes of variables are data variables, and index variables (the latter set also includes certain variables called *size parameters* that are assigned a single value for any instance of the program). Data variables are multidimensional arrays (scalars viewed as zero-dimensional arrays), and represent the primary values computed by the program. Index variables are integer typed, are never explicitly assigned (they get their values implicitly in loops) and are used to access data variables.

The only control construct (other than sequential statement composition) is the **for** loop (note that there is no conditional if-then-else construct[1]). The lower (cf. upper) bound of such a loop is an affine expression of the surrounding loop indices and the size parameters, or the maximum (cf. minimum) of such expressions. The body of the loop is either:

- Another loop
- An assignment statement
- A sequential composition of any of the above

In any assignment statement, the left-hand side (lhs) is a data variable, and the right-hand side (rhs) is an expression involving data variables. The access function of the data variables, whether on the lhs or the rhs, is an affine function of the surrounding loop indices and the size parameters. For our later analysis, we shall assume that the rhs expression is atomic.

In an ACL, an assignment statement S is executed many times for different values of the surrounding loop indices. The loop indices are said to be *valid* if they are within the appropriate bounds, and the set of valid indices surrounding S is called its *iteration domain*, D. Because no conditionals exist, each loop body must be executed exactly once for each valid value of the surrounding indices. Every operation in the program is therefore uniquely identified by $\langle S_i, z\rangle$, where S_i is a statement, and $z \in D$ is an integer vector, the iteration vector. For two operations O_1 and O_2, we denote by $O_1 \lhd O_2$, the fact that operation O_1 *precedes* O_2 *in the sequential execution order* of the ACL.

We now recall certain standard definitions regarding polyhedra.

[1]This constraint may be relaxed without any loss of generality, provided that the conditionals are (conjunctions of) affine inequalities involving only the index variables and size parameters.

Definition 10.1. *A* rational polyhedron *is a set of the form* $\{z \in \mathcal{Q}^n \mid Qz \geq q\}$, *where* Q (*cf.* q) *is an integer matrix* (*cf. vector*). *An* integral polyhedron *or a* polyhedral domain (*or simply a polyhedron, when it is clear from the context*) *is the set of integer points in a rational polyhedron:*

$$D \equiv \{z \in \mathcal{Z}^n \mid Qz \geq q\} \tag{10.1}$$

A parametric family *of polyhedra corresponds to the case where the* q *is a* (*vector-valued*) *affine function of* m *size parameters* p (*i.e.*, $\{z \in \mathcal{Z}^n \mid Qz \geq q - Pp\}$) *or equivalently:*

$$\left\{ \begin{pmatrix} z \\ p \end{pmatrix} \in \mathcal{Z}^{n+m} \mid \begin{bmatrix} Q & P \end{bmatrix} \begin{pmatrix} z \\ p \end{pmatrix} \geq q \right\}$$

We may therefore view such a parametric family of polyhedra as a single $n + m$ *dimensional polyhedron. Note, however, that the converse view, namely, a single* $n + m$ *dimensional polyhedron equivalent to an* n*-dimensional polyhedron, parameterized by* m *parameters, is valid for rational polyhedra, but not for integer polyhedra. For example, the projection of an integral polyhedron on one of its axes is not necessarily an integral polyhedron. Nevertheless, for reasons of clarity we often abuse the notation and view an integer polyhedron as a parameterized family of its constituent projections.*

Parametric polyhedra also admit an equivalent dual definition in terms of generators (their vertices, rays and lines) that are all piecewise affine functions of their parameters.

Example 10.1
Consider $\{i, j, n \mid 0 \leq j \leq i; j \leq n; 1 \leq n\}$ as a 3-dimensional polyhedron. It has two vertices: $[0, 0, 1]$ and $[0, 1, 1]$. Its rays are the vectors in the set $\{[1, 0, 0], [0, 0, 1], [0, 1, 1]\}$. The same polyhedron, viewed as a 2-dimensional polyhedron (parameterized by n), has two vertices, $\{[0, 0], [n, n]\}$, and a ray, $[1, 0]$.

Example 10.2
Consider $\mathcal{P}_1 = \{i, j, n, m \mid 0 \leq j \leq i \leq n; j \leq m; 1 \leq n, m\}$ as a 2-dimensional polyhedron (with parameters n and m). Depending on the relative values of its parameters, it is either a triangle (if $n \leq m$) or a trapezium (otherwise). It does not have rays, and we can see that the set of its vertices is a piecewise affine function of its parameters:

$$\begin{cases} \{1 \leq n \leq m\} \Rightarrow \{[0, 0], [n, 0], [n, n]\} \\ \{1 \leq m < n\} \Rightarrow \{[0, 0], [n, 0], [n, m], [m, m]\} \end{cases}$$

Finally, the polyhedron $\mathcal{P}_2 = \{i, j, n, m \mid 0 \leq i = j - 1 \leq n\}$ is a line segment (parameterized by n), but in a 2-dimensional space. Its vertices are $S_2 = \{[0, 1], [n, n + 1]\}$.

We denote by $z_1 \prec z_2$, the (strict) *lexicographic order* between two vectors, z_1 and z_2. Note that this is a total order, and that the vectors may even be of different dimensions (just like the words in a dictionary). We define $\prec_n$ as the order, $\prec$ applied only to the first n components of two vectors (i.e., if z'_1 and z'_2 are the first n components of z_1 and z_2, respectively, and then $z_1 \prec_n z_2$ if and only if $z'_1 \prec z'_2$. The inverse order $\succ$, and the related (partial) orders $\preceq$ and $\succeq$ are natural extensions and may be defined analogously.

We may easily see that the *execution order* of operations in a sequential ACL (denoted by $\lhd$) is closely related (but not identical) to the lexicographic order. For any two operations, $\langle S_1, z_1\rangle$ and $\langle S_2, z_2\rangle$, let n_{12} be the number of common loop indices surrounding S_1 and S_2. Then:

$$\langle S_1, z_1\rangle \lhd \langle S_2, z_2\rangle \equiv \begin{cases} z_1 \preceq_{n_{12}} z_2 & \text{if } S_1 \text{ appears } \textit{before } S_2 \text{ in the text of the ACL} \\ z_1 \prec_{n_{12}} z_2 & \text{otherwise} \end{cases}$$

Lmax (cf. Lmin) denotes the lexicographic maximum (cf. minimum) of two or more vectors. We remark that the Lmax (or the Lmin) of all the points in a polytope (i.e., a bounded polyhedron) must be one of its vertices, and that the Lmax (cf. Lmin) of two or more piece-wise affine functions is a piecewise affine function. Hence the Lmax (cf. Lmin) of all the points in a union of polyhedra is a piecewise affine function of its parameters.

DEFINITION 10.2. *A* recurrence equation *(RE) is an equation of the following form that defines a variable X at all points z, in a domain, D^X*

$$X(z) = D^X \ : \ g(\ldots X(f(z))\ldots) \tag{10.2}$$

where

- *z is an n-dimensional* index variable.
- *X is a* data variable*, denoting a function of n integer arguments; it is said to be an n-dimensional variable.*
- *$f(z)$ is a* dependence function*, $f : \mathcal{Z}^n \rightarrow \mathcal{Z}^n$; it signifies the fact that to compute the value of X at z, we need the value of X at the index point $f(z)$.*
- *g is an elementary computation (in our later analyses we assume that it is strict and executes in unit time).*
- *" ... " indicate that there may be other similar arguments to g, with possibly different dependences.*
- *D^X is a set of points in $\mathcal{Z}^n$ and is called the domain of the equation. Often, the domains are* parameterized *with one or more (say, l) size parameters. In this case, we represent the parameter as a vector, $p \in \mathcal{Z}^l$, and use p as an additional superscript on D.*

We may also have multiple equations that define X as follows:

$$X(z) = \begin{cases} & \vdots \\ D_i & : \ g_i(\ldots X(f(z))\ldots) \\ & \vdots \end{cases} \tag{10.3}$$

Here, each row of the definition is called a clause (or branch) of the equation, and the domain of X is the (disjoint) union of the domains of each of the clauses $D^X = \bigcup_i D_i$. One may also define an extension of the formalism of REs that allows one to specify computations that have reduction operations involving associative or commutative operators, but this is beyond the scope of this chapter.

An RE is called an *affine recurrence equation* (ARE) if each dependence function is of the form, $f(z) = Az + Bp + a$, where A (cf. B) is a $n \times n$ (cf. $n \times l$) matrix, and a is an n vector.

DEFINITION 10.3. *A* system *or recurrence equations (SRE) is a set of mutually recursive recurrence equations, each one defining one of m variables $X_1, \ldots, X_m$ over an appropriate domain (D^{X_i} of dimension n_i for X_i). Because the equations are mutually recursive, the dependence function associated with an occurrence of, say X_j, on the rhs of an equation defining X_i is a mapping from $\mathcal{Z}^{n_i}$ to $\mathcal{Z}^{n_j}$. In a system of AREs (i.e., an SARE) the dependence functions are all affine (the A matrices are not necessarily square).*

We often desire to manipulate (i.e., geometrically transform) the domains of variables in an SRE and construct an equivalent SRE. In particular, consider the following SRE:

$$X[z] = z \in D^X : g_X(X[f_{XX}(z)], Y[f_{XY}(z)], \dots) \tag{10.4}$$
$$Y[z] = z \in D^Y : g_Y(X[f_{YX}(z)], Y[f_{YY}(z)], \dots)$$

We would like to construct a semantically equivalent SRE where the domain of X is now transformed by some function $\mathcal{T}$. The required SRE is given next (a proof of a general form of this important result is given in Section 10.5):

$$X[z] = z \in \mathcal{T}(D^X) : g_X(X[\mathcal{T} \circ f_{XX} \circ \mathcal{T}'(z)], Y[f_{XY} \circ \mathcal{T}'(z)], \dots) \tag{10.5}$$
$$Y[z] = z \in D^Y : g_Y(X[\mathcal{T} \circ f_{YX}(z)], Y[f_{YY}(z)], \dots)$$

that $\mathcal{T}$ admits a left inverse in the context, D^X (i.e., that a function $\mathcal{T}'$ exits such that, $\forall z \in D^X, \mathcal{T}'(\mathcal{T}(z)) = z$, i.e., $\mathcal{T}' \circ \mathcal{T} = \text{Id}$). An important special (but not the only) case is when $\mathcal{T}(z) = Tz + t$ for some unimodular matrix, T, and a constant vector, t. Also note that SAREs are closed under affine change of basis (COBs) (i.e., if $\mathcal{T}(z) = Tz + t$ for some constant matrix T, and vector t, the resulting SRE is also an SARE.

Definition 10.4. *Finally, we define the* reduced dependence graph*, (RDG) of an SRE as the (multi) graph with a node corresponding to each variable in the SRE. For each occurrence of a variable say, Y, on the rhs of the equation defining say, X, there is a directed edge from X to Y. The edge is labeled with a pair, $\langle \mathcal{D}_i^X, f \rangle$, specifying the (sub) domain and the associated dependence function. The RDG is an important tool for our analysis of SREs.*

10.4 Exact Data Flow Analysis of Affine Control Loops

We now discuss how to to determine the true dependences of an ACL. Before we proceed, let us emphasize a golden rule that we respect in our entire approach. During the analysis of our ACL, the resulting SARE, and implicitly its data flow graph, we do not explicitly construct this graph. This rule is motivated by the following factors:

- The graph is usually too big, as compared with the size of the original code, to be easily manipulated by the compiler or parallelizer. For example, a matrix multiplication program has only a few lines of code but they specify about n^3 operations (and hence an n^3 node data flow graph). It is unreasonable to expect that compiler to explicitly construct this n^3 node graph for analysis purposes.
- More importantly, for parametric programs it is usually not (completely) known statically (i.e., at compile time). For our matrix multiplication example, the data flow graph for a 10×10 input matrix is distinct from that for a 100×100 matrix. The size of the matrix is a parameter of the program and is not known at compile time. Any compilation or parallelization method that requires explicitly constructing the data flow graph of the program can only be able to produce code after the size parameter is instantiated.
- Finally, even if we were to accept these limitations and construct the graph explicitly, it would not be very useful in producing efficient code. The code would need to enumerate each of the operations explicitly, and hence would correspond to a complete “unrolling” of the loops and would have an unacceptably large size.

The implication of our golden rule is that we cannot directly use the conventional and well-developed methods for scheduling and mapping computations specified as *task graphs* to parallel machines. We need to exploit the regularity of our programs and work with a reduced representation of the data flow graph.

We now seek to identify the true dependences of the computations of our ACL. For this, we render explicit the results computed by each operation of the program, and construct an SARE that has the same semantics as the original ACL. Consider an assignment statement:

$$S_i : \ \mathbf{X}[\mathbf{f_i^0}(\mathbf{z}, \mathbf{p})] = \mathbf{g_i}(\ldots \ \mathbf{Y}[\mathbf{f_i^k}(\mathbf{z}, \mathbf{p})] \ldots)$$

Because each operation of the program is uniquely identified by an assignment statement and the values of the surrounding loop indices, the variables of our SARE are simply (the unique labels of) the assignment statements in the ACL. Their domains are the corresponding iteration domains, D_i. This determines the lhs of our SARE. Hence, what we need to determine is the rhs of the equations of our SARE. Obviously, the function g_i (the function computed by the expression on the rhs of the assignment) simply carries over to our SARE. To determine the arguments to g_i, we need to resolve the following question:

Problem 10.1. *For each read,* $\mathbf{Y}[f_i^k(z, p)]$ *of a variable (array)* $\mathbf{Y}$*, with an access function* f_i^k *on the rhs of* S_i*, find the source of the value: which* operation *(i.e., which iteration of which statement) wrote the value to be read?*

In general, this is a function of z, and indeed it is this function that can be the dependence function of our SARE. Our solution is some instance, say z', of an assignment statement, S_j, which has the variable $\mathbf{Y}$ (accessed with some function f_j^0) on its lhs. We consider all such statements as candidates and address them one by one. Each of them is executed many times and possibly writing to many different memory addresses, and moreover the same memory address may be (over)written many times.

For each candidate statement, our solution is one of possibly many instances z' that satisfy the following conditions (because our solution is a function of z, we also include constraints that z must satisfy):

$$\left\| \begin{array}{lcl} z & \in & D_i \\ z' & \in & D_j \\ f_i^k(z, p) & = & f_j^0(z', p) \\ \langle S_j, z' \rangle & \lhd & \langle S_i, z \rangle \end{array} \right. \tag{10.6}$$

The first two constraints ensure that we are dealing with valid operations, the third one ensures that the two operations in question address the same memory location and the final one states that the operation $\langle S_j, z' \rangle$ precedes the operation $\langle S_i, z \rangle$ in the order of execution of the original ACL. Observe that these conditions (10.6) are not yet complete: they admit multiple valid points $z' \in D_j$ that precede $\langle S_i, z \rangle$ and that write into the same memory location that is read by $\langle S_i, z \rangle$. Of these, we have to find the most recent one.

To do so, we first observe that the final constraint in (10.6) is not a simple affine (in)equality. Nevertheless, because it involves the lexicographic precedence between index points, it may be expressed as the disjunction (union) of a finite number of such constraints. Hence, the set of possible solutions, z', that we need to consider is a finite union of polyhedra. Each one is of dimension $n_i + n_j + m$ (the index variables that are involved are those in the respective domains of S_i (i.e., z), of S_j (i.e., z'), and the m parameters of the ACL. We view this as an n_j-dimensional polyhedron, but parameterized with the $n_i + m$ other indices. It is therefore clear that its vertices are piecewise affine functions of z and p.

Now, let us return to the problem of determining the most recent point z' that satisfies the constraints (10.6). For each polyhedron (obtained by the decomposition of $\lhd$ into a disjunction of (in)equalities),

```
S1:   x[n] := b[n]/u[n,n];
      for i = n - 1 down to 1 do
S2:       s := 0;
          for j = i + 1 to n do
S3:           s := s + x[j] * u[i, n - j];
          enddo
S4:       x[i] := (b[i] - s) / u[i,i]
      endo
```

FIGURE 10.1 An affine control loop to solve an upper triangular system of equations.

this is nothing but the point that is the last one to be executed in the sequential execution order (i.e., its Lmax, which has to be one of its vertices — a piecewise affine function of z and p). Among the different polyhedra, the most recent one is therefore the Lmax of their vertices, which is also a piecewise affine function of z and p.

Finally, we return to the comparison of such solutions for different candidates S_j (recall that more than one statement may write into the array variable Y in the ACL). We simply take the Lmax of each of the candidate solutions, and hence we may conclude that the overall solution to Problem 10.1 as posed earlier is a piecewise affine function of z and p. It may be automatically computed by a tool capable of manipulating parameterized polyhedra, or a parametric integer linear programming solver.

Although the overall idea is fairly simple, the details are intricate and are best left to an automatic program analysis tool. We illustrate the analysis method by means of the following example:

Problem. In the program of Figure 10.1, determine the source of **s** on the rhs of **S3**.

Solution. We have 4 statements, of which **S1** is special (its domain is $\mathcal{D}_0 = \mathcal{Z}^0$, the empty polyhedron). The other relevant domains are $\mathcal{D}_2 = \mathcal{D}_4 = \{i \mid 1 \leq i \leq n-1\}$, and $\mathcal{D}_3 = \{i, j \mid 1 \leq i < j \leq n\}$. The following source analyses are needed: **s** and **x** in **S3**, and **s** in **S4**. The main point to note is that the i loop goes down from $n-1$ to 1, and hence we have to be careful about our precedence order, $\prec$, and that we not always look for the lexicographic maximum, Lmax (in the i dimension it is minimum).

Because two statements write into s, Src(**s**, **S3**) $=$ Last($\langle$**S2**, $f_1(i,j)\rangle$, $\langle$**S3**, $f_2(i,j)\rangle$), where $f_1(i,j)$ and $f_2(i,j)$ are, respectively:[2]

$$f_1(i,j) = \text{Last}\left\{ (i' \mid i, j) \;\middle\|\; \begin{array}{l} i' \in \mathcal{D}_2 \\ (i,j) \in \mathcal{D}_3 \\ \langle \mathbf{S2}, i' \rangle \prec \langle \mathbf{S3}, (i,j) \rangle \end{array} \right\} \tag{10.7}$$

$$f_2(i,j) = \text{Last}\left\{ (i', j' \mid i, j) \;\middle\|\; \begin{array}{l} (i', j') \in \mathcal{D}_3 \\ (i,j) \in \mathcal{D}_3 \\ \langle \mathbf{S3}, (i', j') \rangle \prec \langle \mathbf{S3}, (i,j) \rangle \end{array} \right\} \tag{10.8}$$

[2]For notational simplicity, vectors are written as rows, and the bar separates the parameters (i.e., we seek in Eq. (10.7) the last i' as a function of i and j such that the stated constraints are satisfied).

By using the definition of the $\prec$ relation,[3] and our knowledge of the program text, we see that $\langle \mathbf{S2}, i' \rangle \prec \langle \mathbf{S3}, (i, j) \rangle$ if and only if $i' \geq i$. Hence:

$$f_1(i, j) = \text{Last}\left\{ (i' \mid i, j) \,\middle\|\, \begin{array}{l} i' \in \mathcal{D}_2 \\ (i, j) \in \mathcal{D}_3 \\ i \leq i' \end{array} \right\} \tag{10.9}$$

$$= \text{Last}\left\{ (i' \mid i, j) \parallel i \leq i' \leq n;\ 1 \leq i < j \leq n \right\} \tag{10.10}$$

This set of points is viewed as a family 1-dimensional polyhedra (line segments indexed by i') parameterized by points in a 2-dimensional parameter polyhedron, $\mathcal{D}_3$. At all points in $\mathcal{D}_3$, the line segment is from i to n, and here the Last operation is equivalent to finding the lexicographic minimum. Hence, $f_1(i, j) = i$.

To find $f_2(i, j)$, we observe that $\langle \mathbf{S3}, (i', j') \rangle \prec \langle \mathbf{S3}, (i, j) \rangle$ is equivalent to $i' > i \vee (i' = i \wedge j' < j)$, a disjunction of two sets of inequality constraints, and hence:

$$f_2(i, j) = \text{Last}\left(\left\{ (i', j' \mid i, j) \,\middle\|\, \begin{array}{l} (i', j') \in \mathcal{D}_3 \\ (i, j) \in \mathcal{D}_3 \\ i < i' \end{array} \right\} \right.$$

$$\left. \bigcup \left\{ (i', j' \mid i, j) \,\middle\|\, \begin{array}{l} (i', j') \in \mathcal{D}_3 \\ (i, j) \in \mathcal{D}_3 \\ i' = i;\ j' < j \end{array} \right\} \right) \tag{10.11}$$

$$= \text{Last}\left(\text{Last}\left\{ (i', j' \mid i, j) \,\middle\|\, \begin{array}{l} (i', j') \in \mathcal{D}_3 \\ (i, j) \in \mathcal{D}_3 \\ i < i' \end{array} \right\}, \right.$$

$$\left. \text{Last}\left\{ (i', j' \mid i, j) \,\middle\|\, \begin{array}{l} (i', j') \in \mathcal{D}_3 \\ (i, j) \in \mathcal{D}_3 \\ i' = i;\ j' < j \end{array} \right\} \right) \tag{10.12}$$

$$= \text{Last}\left(\text{Last}\left\{ (i', j' \mid i, j) \parallel 1 \leq i < i' < j' \leq n;\ i < j \leq n \right\}, \right.$$

$$\left. \text{Last}\left\{ (i', j', \mid i, j) \parallel 1 \leq i' = i < j \leq n;\ j' < j \right\} \right) \tag{10.13}$$

This is Last $(f_2'(i, j), f_2''(i, j))$ where:

$$f_2'(i, j) = \left\{ \begin{array}{lll} \{i, j \mid i = n - 1;\ j = n\} & : & \perp \\ \{i, j \mid 1 \leq i < j \leq n;\ i \leq n - 2\} & : & (i + 1, n) \end{array} \right. \tag{10.14}$$

$$f_2''(i, j) = \left\{ \begin{array}{lll} \{i, j \mid 1 \leq i = j - 1 \leq n - 1\} & : & \perp \\ \{i, j \mid 1 \leq i < j - 1 \leq n - 1\} & : & (i, j - 1) \end{array} \right. \tag{10.15}$$

(10.16)

[3]Observe the sense of the inequality.

and hence:

$$f_2(i,j) = \begin{cases} \{i,j \mid i = n-1;\, j = n\} & : \ \text{Last}(\bot, \bot) \\ \{i,j \mid 1 \le i = j-1 \le n-2\} & : \ \text{Last}((i+1,n), \bot) \\ \{i,j \mid 1 \le i < j-1 \le n-1;\, i \le n-2\} & : \ \text{Last}((i+1,n), (i, j-1)) \end{cases} \tag{10.17}$$

$$= \begin{cases} \{i,j \mid i = n-1;\, j = n\} & : \ \bot \\ \{i,j \mid 1 \le i = j-1 \le n-2\} & : \ (i+1,n) \\ \{i,j \mid 1 \le i < j-1 \le n-1;\, i \le n-2\} & : \ (i, j-1) \end{cases} \tag{10.18}$$

Finally, remember that:

$$\begin{aligned} \text{Src}(\mathbf{s}, \mathbf{S3}) &= \text{Last}\,(\langle \mathbf{S2}, f_1(i,j)\rangle, \langle \mathbf{S3}, f_2(i,j)\rangle) \\ &= \text{Last}\left(\langle \mathbf{S2}, i\rangle, \left\langle \mathbf{S3}, \begin{cases} \{i,j \mid i = n-1;\, j = n\} & : \ \bot \\ \{i,j \mid 1 \le i = j-1 \le n-2\} & : \ (i+1,n) \\ \{i,j \mid 1 \le i < j-1 \le n-1;\, i \le n-2\} & : \ (i, j-1) \end{cases} \right\rangle \right) \\ &= \text{Last}\left(\langle \mathbf{S2}, i\rangle, \begin{cases} \{i,j \mid i = n-1;\, j = n\} & : \ \langle \mathbf{S3}, \bot\rangle \\ \{i,j \mid 1 \le i = j-1 \le n-2\} & : \ \langle \mathbf{S3}, (i+1,n)\rangle \\ \{i,j \mid 1 \le i < j-1 \le n-1;\, i \le n-2\} & : \ \langle \mathbf{S3}, (i, j-1)\rangle \end{cases} \right) \\ &= \begin{cases} \{i,j \mid i = n-1;\, j = n\} & : \ \text{Last}(\langle \mathbf{S2}, i\rangle, \langle \mathbf{S3}, \bot\rangle) \\ \{i,j \mid 1 \le i = j-1 \le n-2\} & : \ \text{Last}(\langle \mathbf{S2}, i\rangle, \langle \mathbf{S3}, (i+1,n)\rangle) \\ \{i,j \mid 1 \le i < j-1 \le n-1;\, i \le n-2\} & : \ \text{Last}(\langle \mathbf{S2}, i\rangle, \langle \mathbf{S3}, (i, j-1)\rangle) \end{cases} \\ &= \begin{cases} \{i,j \mid i = n-1;\, j = n\} & : \ \langle \mathbf{S2}, i\rangle \\ \{i,j \mid 1 \le i = j-1 \le n-2\} & : \ \langle \mathbf{S2}, i\rangle \\ \{i,j \mid 1 \le i < j-1 \le n-1;\, i \le n-2\} & : \ \langle \mathbf{S3}, (i, j-1)\rangle \end{cases} \\ &= \begin{cases} \{i,j \mid 1 \le i = j-1 \le n-1\} & : \ \langle \mathbf{S2}, i\rangle \\ \{i,j \mid 1 \le i < j-1 \le n-1;\, i \le n-2\} & : \ \langle \mathbf{S3}, (i, j-1)\rangle \end{cases} \end{aligned} \tag{10.19}$$

We have determined the precise source of **s** on the rhs of statement **S3** as a function of its loop iteration indices (and the system parameters). Essentially, it states that for the very first iteration (i.e., for $j = i + 1$) the producer is the ith iteration of statement **S2**; otherwise it is the "previous," that is, the $(i, j - 1)$-th iteration of **S3** itself, as can be easily verified. *end of example*

10.5 System of Affine Recurrence Equations: Merely an Intermediate Form?

In this section, we argue that the formalism of SAREs, augmented with reduction operations, is more than just an intermediate form for the parallelization of ACLs, but is interesting in its own right. We present an equational language ALPHA based on this formalism, describe its expressive power and briefly explain its denotational semantics. We also describe some of the analyses that are enabled by this formalism (some that would be extremely difficult for a imperative language).

ALPHA, designed by Mauras in 1989 in the context of systolic array synthesis at IRISA (France), is a strongly typed data-parallel functional language based on SAREs. MMALPHA is a prototype transformation system (also developed at IRISA, and available under the Gnu Public License at

```
system ForwardSubstitution : { N | N>1 }               – comments are like this
                      ( A : { i,j | 0<j<i<=N } of real;  – a 2D input variable
                        B : { i | 0<i<=N } of real)      – a 1D input variable
          returns     ( X : { i | 0<i<=N } of real );    – a 1D output variable
let
     X = case
          {i | i=1} : B;
          {i | i>1} : B - reduce(+, i, j → i), A * X.(i, j → j));
          esac;
tel:
```

FIGURE 10.2 ALPHA program for the forward substitution algorithm.

www.irisa.fr/cosi/ALPHA) for reading and manipulating ALPHA programs through correctness-preserving transformations and eventually generating very high speed integrated circuit hardware description language (VHDL) description of regular, systolic very large scale integrated (VLSI) circuits or (sequential or parallel) code for programmable (i.e., instruction-set) processors.

ALPHA variables are type declared at the beginning of a program, and represent polyhedra2l-shaped multidimensional arrays. The polyhedra specified in the declaration are called the *domains* of the variables. For example, a (strictly) lower triangular, real matrix is specified by the following declaration:[4]

```
A: {i,j| 0<j<i<=N } of real
```

To introduce the main features of ALPHA, consider the problem of solving, using forward substitution, a system of linear inequalities, $Ax = b$, where A is a lower triangular $n \times n$ matrix with unit diagonal. A high-level, mathematical description of the program would be:

$$\text{for } i = 1, \dots, n, \quad x_i = \begin{cases} \text{if } i = 1 & b_j \\ \text{if } i > 1 & b_i - \sum_{j=1}^{i-1} A_{i,j} x_j \end{cases} \tag{10.20}$$

The corresponding ALPHA program (Figure 10.2) is identical, except for syntactic sugar. The first line names the system and declares that it has a positive integer parameter, **N**, which can take any integer value greater than 1. The next three lines are declarations of the input and output variables of the system (respectively, before and after the **returns** keyword). The domain of **A** is triangular, while **B** and **X** are one-dimensional variables (vectors). A system may also have local variables (not present here), which are declared after the system header, using the keyword **var**. The body of the program is a number of equations delineated by the **let** and **tel** keywords. The rhs of an equation is an expression, following the syntax given in Table 10.1.

In our example, we have a single equation, almost identical to (10.20) above. The **case** construct has the usual meaning and allows us to define conditional expressions. The **restrict** construct has the syntax ⟨**domain**⟩ **:** ⟨**expr**⟩, and denotes the expression ⟨**expr**⟩ but restricted to the subset of index points in ⟨**domain**⟩. The **reduce** construct corresponds to the summation in (10.20) and has

[4]This syntax is to be read as "the set of i, j such that" the specified linear inequalities are satisfied. In specifying the inequalities, we may group expressions using parentheses. For example, $\mathbf{(i + j, j + 10)}$ $\mathbf{<= N}$ specifies a domain where both $\mathbf{i + j}$ and $\mathbf{j + 10}$ are no greater than **N**.

TABLE 10.1 Syntax of ALPHA Expressions (Summary)

$\langle$**Exp**$\rangle$	:= $\langle$**Const**$\rangle$ | $\langle$**Var**$\rangle$	Atomic Expressions
	| $\oplus$ $\langle$**Exp**$\rangle$ | $\langle$**Exp**$\rangle$ $\oplus$ $\langle$**Exp**$\rangle$	Unary/Binary Pointwise ops
	| **if** $\langle$**Exp**$\rangle$ **then** $\langle$**Exp**$\rangle$ **else** $\langle$**Exp**$\rangle$	A Ternary Pointwise op
	| $\langle$**Dom**$\rangle$:$\langle$**Exp**$\rangle$	Restrictions
	| **case** $\langle$**Exp**$\rangle$; . . . $\langle$**Exp**$\rangle$; **esac**	Case Expressions
	| $\langle$**Exp**$\rangle$.$\langle$**Dep**$\rangle$	Dependence Expressions
	| **reduce** ($\oplus$, $\langle$**Dep**$\rangle$, $\langle$**Exp**$\rangle$)	Reductions
$\langle$**Dom**$\rangle$	:= {$\langle$**Idx**$\rangle$ . . . |$\langle$**IdxExpr**$\rangle$ >= 0}	Domains
$\langle$**Dep**$\rangle$	:= ($\langle$**Idx**$\rangle$ . . . → $\langle$**IdxExpr**$\rangle$ . . .)	Dependences
$\langle$**IdxExpr**$\rangle$	:= Any affine expression of indices and system parameters	

three parts: an associative and commutative operator, a projection function and an expression. Here the operator is +. The projection is **(i, j → i)**, and denotes (intuitively) the fact that within the body of the **reduce**, there are two indices, **i** and **j**, but only **i** is visible outside the scope of the **reduce** (i.e., the two-dimensional expression of the body is projected by the mapping **(i, j → i)** to a one-dimensional one. The body of our **reduce** construct is the expression, **A** $*$ **X. (i, j → j)**.

Here, **(i, j → j)**, is a dependency function and denotes the fact that to compute the body at **[i, j]**, we need the value of **X** at index point **[j]** (the dependency on **A** is not explicitly written — it is the identity). Dependencies are important in ALPHA. They have the syntax **(Idx, . . . → IdxExpr, . . .**), where each **Idx** is an index name, and **IdxExpr** is an affine expression of the system parameters and the **idx**'s. ALPHA and MMALPHA use this syntax for specifying a multidimensional affine function in many different contexts (e.g., the projection function in a **reduce** expression). This syntax can be viewed as a special kind of lambda expression, restricted to affine mappings from $\mathcal{Z}^n$ to $\mathcal{Z}^m$. Such a function, f, may be equivalently represented by an $m \times n$ matrix A, and an m-vector a (i.e., $f(z) = Az + a$) and we later use this form for analysis purposes.

10.5.1 Semantics of Alpha Expressions

Because ALPHA is a data-parallel language, expressions denote collections of values. Indeed all expressions (not just variables) can be viewed as multidimensional arrays and denote a function from indices to values. ALPHA semantics consist of two parts: a semantic function and a domain. The semantic function is defined using classic methods and is fairly obvious (the only subtle points are dependencies and reductions, as described later), but the domains are a unique aspect of ALPHA. Every (sub) expression in a program has a domain, and it can be determined from the domains of its subexpressions, as summarized follows:

- *Identifiers and constants.* An identifier simply denotes the variable it identifies, and is defined over its declared domain. In general, this is a finite union of polyhedra. A constant expression denotes the constant itself, and its domain is $\mathcal{Z}^0$, the zero-dimensional polyhedron.
- *Pointwise operators.* The expression **E** $\oplus$ **F** denotes the pointwise application of the operator $\oplus$ to the corresponding elements of the expressions **E** and **F**, and hence its domain is Dom(**E**) $\cap$ Dom(**F**), the intersection of the domains of its subexpressions. The semantics of unary or ternary (the **if-then-else**) pointwise operators are similar.
- *Restriction.* The expression $\mathcal{D}$: **E** denotes the expression **E**, but restricted to the domain $\mathcal{D}$, and hence its domain is $\mathcal{D} \cap$ Dom(**E**).

- *Case.* The expression **case E; F esac** denotes an expression that has alternative definitions (there may be more than two alternatives). Its domain is Dom(**E**) ∪ Dom(**F**), the (disjoint) union of the domains of its subexpressions.
- *Dependency.* First, we note that the dependency ($\mathbf{z} \rightarrow f(\mathbf{z})$) by itself simply denotes the affine function, f. We therefore abuse the notation somewhat to use the same nomenclature for both the syntactic construct as well as for the corresponding semantic function.

  The expression **E.f** denotes an expression whose value at **z** is the value of **E** at **f**(**z**). Its domain must therefore be the set of points which are mapped by f to some point in the domain of **E**. This is nothing but the preimage, of Dom(**E**) by f, that is:

$$\text{Dom}(\mathbf{E}\,.\,\mathbf{f}) = \text{Pre}(\text{Dom}(\mathbf{E}), \mathbf{f}) = f^{-1}\text{Dom}(\mathbf{E})$$

  where f^{-1} is the relational inverse of f.
- *Reductions:* The expression **reduce** ($\oplus$, **f, E**) denotes an expression whose domain is the image of the domain of **E** by f. Its value at any point, z, in this domain is obtained by taking all points in the domain of **E** that are mapped to z by f, and applying the associative and commutative operator $\oplus$ to the values of **E** at these points.

Note that the preceding semantics imply that the domain of any ALPHA expression may be determined recursively in a top-down traversal of its syntax tree. Because the declared domains in ALPHA (i.e., the domains of the leaves of the tree) are finite unions of polyhedra, and thanks to the mathematical closure properties of polyhedra and affine functions, the domain of any ALPHA expression can be easily determined by using a library for manipulating polyhedra.

10.5.2 ALPHA Transformations

The denotational semantics given earlier describe what an ALPHA expression denotes or means, without necessarily showing how to compile or otherwise execute an ALPHA program. The denotational semantics enable us to formally reason about ALPHA programs, and to develop and prove the validity of program transformations. These program transformations are available in the MMALPHA system, and we shall describe two of the important ones here.

10.5.2.1 Normalization

A number of properties can be proved based on the denotational semantics of ALPHA expressions. For example, we can show that for any expression **E**, and dependences f_1 and f_2, the expression $\mathbf{E.f_1.f_2}$ is semantically equivalent to **E.f**, where $f = f_1 \circ f_2$ is the composition[5] of f_1 and f_2.

Although a large number of such transformation rules could be developed and proposed to the user, a certain set of rules is particularly useful to simplify any ALPHA expression into what is called a *normal form*, or the **case-restriction-dependency** form. It consists of an (optional) outer case, each of whose branches is a (possibly restricted) simple expression. A simple expression consists of (possibly a reduction of) either variables or constants composed with a single dependency function (which may be omitted, if it is the identity), or pointwise operators applied to such subexpressions. This simplification is obtained by a set of rewrite rules that combine dependencies together, eliminate empty domain expressions, introduce new local variables to define variables for certain subexpressions (to remove nested reductions), etc.

[5]Note that function composition is right associative, that is, $f_1 \circ f_2(z) = f_1(f_2(z))$.

Let $\mathbf{X} = \{\mathbf{i}, \mathbf{j} | \mathbf{0} < \mathbf{i}, \mathbf{j} <= \mathbf{N}\} : \mathbf{X}.(\mathbf{p}, \mathbf{q} \rightarrow \mathbf{q}, \mathbf{p}) + 0.(\mathbf{i}, \mathbf{j} \rightarrow)$ be an equation in an ALPHA program (observe that the domain of the constant `0` is $\mathcal{Z}^0$, whose preimage by the function $(i, j \rightarrow)$ is $\mathcal{Z}^2$, which is coherent with the rest of the expression). Because the rhs of this equation is normalized, we can rewrite the equation as follows:

- Rename the indices in the restrictions and in the dependencies to be identical (e.g., by replacing **p** and **q** by **i** and **j**, respectively).
- Move the index names in the dependencies (left of the $\rightarrow$) and in domains (left of the $|$) to the left of the entire equation.

This yields the following "sugared" syntax, called the *array notation*, which is often more readable (indeed, all subsequent examples in this chapter are presented in this notation).

$$\mathbf{X[i, j]} = \{|\mathbf{0} < \mathbf{i}, \mathbf{j} <= \mathbf{N}\} : \mathbf{X[i, j]} + 0[]$$

Also observe that SAREs as defined in Section 10.3 constitute a proper subset of ALPHA programs, namely, those that do not contain reductions, and those that are normalized.

10.5.2.2 Generalized Change of Basis

Perhaps the most important transformation in the ALPHA system is the change of basis (COB). The intuition behind it is as follows. Because an ALPHA variable can be viewed as a multidimensional array defined over a polyhedral domain, we should be able to "move" (or otherwise change the "shape" of) its domain and construct an equivalent program. We now develop such a transformation.

Let $\mathcal{T}$ and $\mathcal{T}'$ be functions such that for all points z in some set S of index points, $\mathcal{T}' \circ T(z) = z$. Note that $\mathcal{T}$ and $\mathcal{T}'$ may not even be affine, but even in the case when they are, we do not insist that they be square, or that $\mathcal{T}' \circ \mathcal{T}$ be the identity. We say that $\mathcal{T}'$ is the left inverse of $\mathcal{T}$ in the context of S. The following can easily be proved from the semantics of ALPHA expressions as defined earlier.

REMARK **10.1.** *For any* ALPHA *expression* **E***, let* $\mathcal{T}$ *and* $\mathcal{T}'$ *be such that* $\mathcal{T}'$ *is the left inverse of* $\mathcal{T}$ *in the context of* Dom(**E**). *Then* **E** *is semantically equivalent to* $\mathbf{E}.\mathcal{T}'.\mathcal{T}$*, and any occurrence of the former anywhere in the program may be replaced by the latter.*

This implies in particular, that if we choose **E** as the subexpression consisting of just the variable **X**, then every occurrence of **X** can be replaced by $\mathbf{X}.\mathcal{T}'.\mathcal{T}$, without affecting the program semantics. Moreover, if **Expr** is the entire rhs of the equation defining **X**, then:

$$\mathbf{X}.\mathcal{T}' = \mathbf{Expr}.\mathcal{T}' = \mathbf{X}'(\text{say})$$

We may therefore introduce a new local variable $\mathbf{X}'$ and define its domain to be $\text{Pre}(D_X, \mathcal{T}')$, and the rhs of its defining equation to be $\mathbf{Expr}.\mathcal{T}'$. Next, we replace every occurrence of the subexpression $\mathbf{X}.\mathcal{T}'$ in the program by $\mathbf{X}'$ (because $\mathbf{X}.\mathcal{T}' = \mathbf{X}'$); and finally because **X** is no longer used in the program, drop it, and then rename the $\mathbf{X}'$ to be **X**. This argument is embodied in the following theorem:

THEOREM **10.1.** *In an* ALPHA *program with a local variable* **X** *declared over a domain* **D***, let* $\mathcal{T}$ *and* $\mathcal{T}'$ *be such that* $\mathcal{T}'$ *is the left inverse of some* $\mathcal{T}$ *in the context of* **D**. *The following transformations yield a semantically equivalent program:*

- *Replace the domain of declaration of* **X** *by* $Pre(\mathbf{D}, \mathcal{T}')$.
- *Replace all occurrences of* **X** *on the rhs of any equation by* $\mathbf{X}.\mathcal{T}$.
- *Compose the entire rhs expression of the equation for* **X** *with* $\mathcal{T}'$.

Many program transformations such as alignment, scheduling and processor allocation can be implemented as an appropriate COB. Moreover, we can see that the rules for COB for SREs in Section 10.3 are merely a special case where the resulting program is normalized after applying the preceding rules.

10.5.3 Reasoning about ALPHA Programs

The functional and equational nature of ALPHA provides us important advantages not available in a conventional imperative language. These include the ability to formally reason about programs, the possibility to systematically derive programs from mathematical specifications, the use of advanced program analysis techniques such as abstract interpretation, etc. We illustrate one of them by showing how formal equivalence properties of ALPHA programs can be proved systematically. Consider the following two recurrences, both defined over the same domain, $D = \{i, j, k \mid 1 \le i, j \le n; 0 \le k \le n\}$:

$$T[i,j,k] = \begin{cases} \{\mid k = 0\} & : \ 0 \\ \{\mid i = j = k\} & : \ 1 + T[i,j,k-1] \\ \{\mid i = k \neq j\} & : \ 1 + \max(T[k,k,k], T[i,j,k-1]) \\ \{\mid j = k \neq i\} & : \ 1 + \max(T[i,j,k-1], T[k,k,k]) \\ \{\mid i, j \neq k; k > 0\} & : \ 1 + \max(T[i,j,k-1], \\ & \quad\ T[i,k,k], T[k,j,k-1]) \end{cases} \tag{10.21}$$

$$T'[i,j,k] = \begin{cases} \{\mid k = 0\} & : \ 0 \\ \{\mid i = j = k\} & : \ 3k-2 \\ \{\mid i = k \neq j\} & : \ 3k-1 \\ \{\mid j = k \neq i\} & : \ 3k-1 \\ \{\mid i, jk; k > 0\} & : \ 3k \end{cases} \tag{10.22}$$

The definition of T is recursive, whereas that of T' is in closed form. However, it is probably not immediately obvious that the two functions compute the same result. We would like to formally prove this. Specifically, we would like to show that for any point z in D, $T[z] = T'[z]$. This can be done manually by an inductive argument (essentially a structural induction on the recursive structure of T).

To do this mechanically with a theorem prover and MMALPHA, we first write an ALPHA program that has three variables, all defined over the same domain, D. The first two are integer-typed variables, T and T' as defined above. The third is a Boolean variable, Th, the theorem that we seek to prove (i.e., the rhs of its equation is simply the expression $\mathbf{T} = \mathbf{T}'$). We would like to show that **Th** has the value true everywhere in its domain. Our proof proceeds as follows:

- We first substitute the definitions of $\mathbf{T}$ and $\mathbf{T}'$ in the rhs of **Th** (this is a provably correct transformation, because ALPHA is functional, and is trivially simple to implement in MMALPHA), yielding the following equation for **Th** (a normalization has been done to render the program more readable).

$$\mathbf{Th}[i,j,k] = \begin{cases} \{\mid k = 0\} & : \ 0 = 0 \\ \{\mid i = j = k\} & : \ 3k-2 = 1 + T[i,j,k-1] \\ \{\mid i = k \neq j\} & : \ 3k-1 = 1 + \max(T[k,k,k], T[i,j,k-1]) \\ \{\mid j = k \neq i\} & : \ 3k-1 = 1 + \max(T[i,j,k-1], T[k,k,k]) \\ \{\mid i, j; k > 0\} & : \ 3k = 1 + \max(T[i,j,k-1], T[i,k,k], T[k,j,k-1]) \end{cases} \tag{10.23}$$

- Next we make the inductive hypothesis, namely, that the theorem is true for the recursive calls in the definition of **T**, so that **T** may be replaced by $\mathbf{T}'$ on the rhs of (10.23). In general, this is not a correctness-preserving transformation in MMALPHA. It is simply used here in the context of proving a certain property, and corresponds to making the induction hypothesis.
- We next substitute the newly introduced instances of $\mathbf{T}'$ by the closed form definition from Equation (10.22) and simply normalize the program. We obtain an equation (not shown) with a number of case branches, each of which is an equality of some arithmetic expressions. By using a standard theorem prover with some knowledge about arithmetic operations, it is fairly easy to show that these are all tautologies.

Actually, for our example, we can even do a little better, because the closed form of the expressions are all affine functions of the indices. For such programs, the entire proof can be completely performed in MMALPHA, and indeed, if we simply normalize the program after making the inductive hypothesis, we simply obtain the following equation for **Th**:

$$\mathbf{Th}[i, j, k] = \mathbf{True} \tag{10.24}$$

which is exactly what we wanted to prove (and why we did not show it earlier).

10.6 Scheduling in the Polyhedral Model

We next describe how to resolve one of the fundamental analysis questions, namely, assigning an execution date to each instance of each variable in the original SARE. Remember that our golden rule implies that we work on the compact representation of the program, instead of the computation graph that it induces (called the *extended* dependence graph [EDG]). This means that the schedule cannot be specified by enumerating the time instances at which each operation is executed, but instead as a closed form function. We first give a classic technique (the wavefront method) to determine schedules for a single uniform recurrence equation (URE). Then, we describe how this can be extended to deal with system of uniform recurrence equations (SUREs), present some of the limitations of these extensions, and develop the algorithms used to determine more general schedules for SUREs. Next we show how these scheduling algorithms can be carried over to AREs and SAREs, by exploiting the fact that the domains of the variables are polyhedra.

Because our schedule is to be expressed in closed form, the time instant at which an operation $O = \langle S_i, z\rangle$ (or equivalently, a variable X at point z) is executed is given by the function $\tau(O)$ or $\tau_X(z)$. In the polyhedral model we restrict ourselves to affine schedules, defined as follows (recall that p is the l-dimensional size parameter):

$$\tau(\langle S_X, z\rangle) = \Lambda_X z + \alpha_X + \Lambda'_X p \tag{10.25}$$

where Λ_X (resp. Λ'_X) is a constant $k \times n_X$ (resp. $k \times l$) integer matrix, and α_X is an integral k vector. Here, k is called the *dimension* of the schedule, and n_X is the number of dimensions in the iteration domain of S_X). Such a schedule maps every operation of the ACL to a k-dimensional integer vector. Because a natural total order relation — the lexicographic order — exists over such vectors we can interpret these vectors as a time instant.

There are a number of special cases that we shall consider. If $k = 1$ we have what are called 1-dimensional schedules — these are the simplest to understand and visualize, and we shall initially focus on this class. Another common case is when the schedule function is the same for all the variables of the SARE (regardless of the dimension of the schedule, but usually for

1-dimensional schedules). We call them *variable-independent* schedules,[6] and drop the subscript i in Equation (10.25). Note that variable-independent schedules can only be defined for SREs where all variables have the same number of dimensions. A slightly more general case is when the linear part of the affine function is the same for all variables, but the constant α_X may be different. Such schedules are called *shifted linear schedules*.

One-dimensional (variable-independent) schedules have a nice, intuitive geometric interpretation. The set of points computed at a given instant t are precisely those that satisfy $\Lambda z + \Lambda' p + \alpha = t$. Because, for a given problem instance $\Lambda' p$ is fixed, these points are characterized by the equation $\Lambda z =$ constant, which defines a family of hyperplanes whose normal vector is Λ. Such schedules are therefore visualized as wavefronts or iso-temporal hyperplanes through the iteration space. We defer the geometric visualization of multidimensional schedules (i.e., schedules with $k > 1$) to later.

Because the schedule maps operations to k-dimensional integer vectors, it is obvious that any schedule is valid if and only if the total order induced by the schedule respects the causality constraints of the computation, as described later.

REMARK 10.2. *A schedule as defined in Equation* (10.25) *is valid if and only if for every edge* $\langle D, f\rangle$ *from node* X *to* Y *in the* RDG:

$$\begin{gathered}\forall z \in D \\ \Lambda_X z + \alpha_X + \Lambda'_X p \succ \Lambda_Y f(z) + \alpha_Y + \Lambda'_Y p\end{gathered} \tag{10.26}$$

In formulating this constraint we assume a machine architecture that can execute any instance of the rhs of any statement in the original ACL in one time step. The main goal of the scheduling algorithms is to express the potentially unbounded instances of the preceding constraints (10.26) in a compact manner by exploiting properties of the polyhedral model.

10.6.1 Scheduling a Single Uniform Recurrence Equation: One-Dimensional Schedules

Consider a single URE with appropriate boundary conditions (not shown):

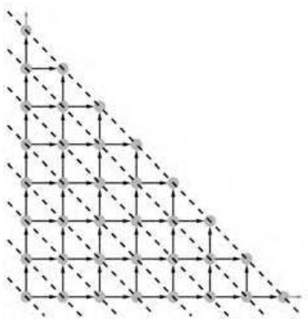
$$\forall z \in D \quad X[z] = g(X[z + d_1] \ldots X[z + d_s]) \tag{10.27}$$

The scheduling constraints of Equation (10.26) reduce to: for $j = 1 \ldots s$

$$\begin{gathered}\forall z \in D \\ \Lambda z + \alpha + \Lambda' p > \Lambda(z + d_j) + \alpha + \Lambda' p \\ \text{i.e., } \forall z \in D \\ \Lambda d_j < 0\end{gathered} \tag{10.28}$$

We observe that the z "cancels out" from the constraints, thus giving us a finite number of constraints. The feasible space of valid schedules is thus a polyhedron (indeed, a cone). There are (potentially unbounded) many valid schedules, and by introducing an appropriate linear cost function (e.g., the

[6]Sometimes we use the term *variable-dependent* schedules when we are *not* restricting ourselves to variable-independent schedules.



$$\begin{aligned} X[i,j] &= g(X[i-1,j], X[i,j-1] \\ t(i,j) &\equiv ai + bj + \alpha \end{aligned}$$

Schedule validity conditions

$$\begin{aligned} [a,b][0,-1]^T &< 0 \\ [a,b][-1,0]^T &< 0 \\ \alpha &\geq 0 \end{aligned}$$

i.e., $\{\mathsf{a}, \mathsf{b}, \alpha \mid \mathsf{a}, \mathsf{b} > 0, \alpha \geq 0\}$

The constraint on α is because $t(i,j)$ must be positive at all points in the domain. It is easy to see that the *optimal*, i.e., the fastest schedule is $t(i,j) = i + j$.

FIGURE 10.3 Scheduling a single URE with a one-dimensional schedule.

total execution time) we can easily formulate scheduling as an integer linear programming problem, and draw from well-established techniques to find optimal schedules. An example of this technique is given in Figure 10.3.

10.6.2 Scheduling Systems of Uniform Recurrence Equations: Variable-Dependent Schedules

With SUREs, we may use the same technique as earlier and seek a single, variable-independent schedule. The formulation of the schedule constraints then remains identical to that in Equation (10.28) above. However, variable-independent schedules are restrictive because some SUREs, that is, the SURE of Equations (10.29) to (10.32) given below, may not admit such a schedule.

$$y_i = Y[i, n-1] \tag{10.29}$$

$$Y[i,j] = \begin{cases} j = 0 \ : \ W[i,j] * X[i,j] \\ j > 0 \ : \ Y[i,j-1] + W[i,j] * X[i,j] \end{cases} \tag{10.30}$$

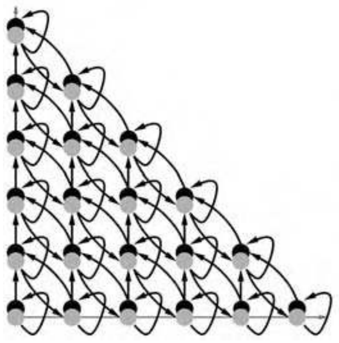
$$X[i,j] = \begin{cases} j = 0 \ : \ x_i \\ j > 0 \ : \ X[i-1,j-1] \end{cases} \tag{10.31}$$

$$W[i,j] = \begin{cases} i = 0 \ : \ w_j \\ i > 0 \ : \ W[i-1,j] \end{cases} \tag{10.32}$$

Here, the computation at a point $[i, j]$ needs another result at the same point, and hence there is a dependence vector $\vec{0}$. It does not matter that the two variables involved are distinct, as far as the scheduling constraints are concerned:

A simple way out of this situation is to use shifted linear schedules, for which the constraints now include the α values of each variable. For the SURE of Equations (10.29) to (10.31), the optimal schedule is obtained to be $t_X(i,j) = t_W(i,j) = i + j$ and $t_Y(i,j) = i + j + 1$. However, although shifted linear schedules resolve the problem for the preceding SURE, they are still not general enough, as illustrated in Figure 10.5. The problem of determining a variable-dependent 1-dimensional schedule for an SURE can also be formulated as a linear programming problem, although the arguments are a little more intricate than those leading to Equation (10.28).

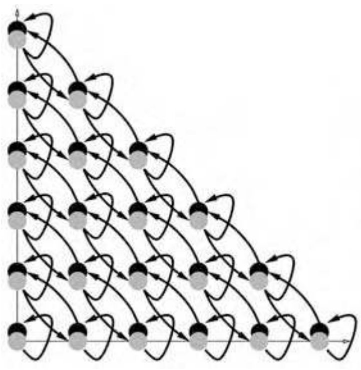
Essentially, we define linear constraints that ensure that each data value "comes from a strictly preceding" hyperplane. Note that the simple geometric interpretation of a single family of isotemporal wavefronts sweeping out the iteration domain seems to break down with variable-dependent schedules. This (slightly) complicates the final code generation step, as we shall see later.



$$
\begin{aligned}
X[i,j] &= f(X[i-1,j+1], Y[i,j-1]) \\
Y[i,j] &= g(Y[i+1,j-1], X[i,j])
\end{aligned}
$$

For this SURE, it can be verified from the EDG that (i) the longest path reaching (either of the two) nodes at index point $[i,j]$ must pass through *all* the points in the triangular region "below" the $i+j$ "diagonal." Since the number of such points is a quadratic function of the indices, there can be no linear schedule—there will always be a point (far enough from the origin) such that a path reaching it will exceed any proposed linear schedule.

FIGURE 10.4 Limitations of variable-independent and shifted linear schedules.



$$
\begin{aligned}
X[i,j] &= f(X[i-1,j+1]) \\
Y[i,j] &= g(Y[i+1,j-1], X[i,j])
\end{aligned}
$$

It can be easily verified from the EDG (shown left) of the above SURE that (i) the computations on each diagonal line are independent; (ii) the length of the longest path reaching a green or light gray node (resp. a red or dark gray node) at index point $[i,j]$ is i (resp. $i+2j+1$). The SURE thus does not admit a variable-independent (or even shifted linear) schedule. But clearly, a valid variable-dependent schedule is $t_X(i,j) = i$; $t_Y(i,j) = i+2j+1$.

FIGURE 10.5 Limitations of one-dimensional schedules.

10.6.3 Scheduling Systems of Uniform Recurrence Equations: Multidimensional Schedules

Although variable-dependent, one-dimensional affine schedules work well on SUREs such as the preceding example, they are still too restrictive. It is not always possible to find such a schedule for many SUREs (see Figure 10.4). It is therefore necessary to use the full generality of multidimensional schedules as defined earlier. A nice geometric interpretation still exists (easiest to visualize with variable-independent schedules). We interpret each row of the Λ matrix as defining a family of hyperplanes or wavefronts. The first row defines, say the hours, the next one the minutes, and so on. However, caveat emptor: the hours and minutes analogy is at best approximate. In lexicographic order, $\begin{pmatrix} 0 \\ x \end{pmatrix}$ can *never* precede $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$, however large we make x; but 61 min exceeds 1 h.

Clearly in a k-dimensional schedule, the notion of optimality is simply to reduce k as much as possible (a quadratic schedule is faster than a cubic one, and so on), and this gives a simple greedy strategy for finding a multidimensional schedule for SURE, which has been proved to be optimal.

- Try to determine a 1-dimensional schedule, using the linear programming formulation implied by Equation (10.28) or its generalization to variable-dependent, 1-dimensional schedules (not shown here). If the algorithm succeeds, we are done.
- If not, we seek to resolve a modified linear programming problem, where the inequalities are now nonstrict (i.e., we seek scheduling hyperplanes, such that all dependences come from either

a strictly preceding, or even the same hyperplane). If even the weak problem does not admit a feasible solution, the SURE does not admit a schedule, and we are done.

- Otherwise, we seek a solution where as many dependences as possible are satisfied in the strict sense (this is the greedy strategy).
- Next, we delete from the RDG, the edges corresponding to dependences that are strictly satisfied by the preceding solution.
- We recursively seek a one-dimensional schedule on each connected component of the resulting RDG (unless we have already recursed to a depth equal to the number of dimensions, in which case too, the SURE does not admit a schedule, and we are done).

10.6.4 Scheduling Affine Recurrence Equations and Systems of Affine Recurrence Equations

Let us now consider a single ARE:

$$\forall z \in D \quad X[z] = g(X[A_1 z + a_1] \ldots X[A_s z + a_s]) \tag{10.33}$$

Following the same arguments as for the case of a single URE, the scheduling constraints of Equation (10.26) reduce to: for $j = 1 \ldots s$

$$\begin{gathered} \forall z \in D \\ \Lambda z + \alpha + \Lambda' p > \Lambda(A_j z + a_j) + \alpha + \Lambda' p \\ \text{i.e., } \forall z \in D \\ \Lambda((A_j - I)z + a_j) < 0 \end{gathered} \tag{10.34}$$

Here, the z does not "cancel out" from the constraints, and thus we still have a potentially unbounded number of constraints to satisfy. However, we know that the domain D is a polyhedron and admits a compact representation. In particular, we exploit the fact that an affine inequality constraint is satisfied at all points in a polyhedron iff it is satisfied at its extremal points. This leads to the following result.

REMARK 10.3. $\langle \Lambda, \alpha \rangle$ *is a valid schedule for the* ARE (10.33) *if and only if for* $j = 1 \ldots s$, *for each vertex,* σ *and each ray,* ρ *of* $\mathcal{D}$*:*

$$\begin{gathered} \Lambda\sigma + \alpha > 0 \\ \Lambda\rho \geq 0 \\ \Lambda\sigma > \Lambda(A_j\sigma + a_j) \\ \Lambda\rho \geq \Lambda A_j \rho \end{gathered}$$

Again, we have been able to exploit the compact representation (this time of the domain of the equation) to reduce the problem to the resolution of a finite number of linear constraints. As with UREs and SUREs and using arguments analogous to those we have seen earlier, this result can be extended to shifted linear and variable-dependent (one- and multidimensional) schedules.

10.6.5 Undecidability Results

We conclude this section with some important observations. The general problem of determining a schedule for an SARE is undecidable. Indeed, there are many subtle related results.

- For a single URE defined on an arbitrary polyhedral domain the scheduling problem is decidable.
- For an SURE defined over any bounded set of domains, the problem is decidable.
- For an SURE defined over an arbitrary set of domains, the problem is undecidable.

- For an SURE where all variables are defined over the entire positive orthant (a special case of unbounded domains), the problem is decidable.
- Because SAREs are more general than SUREs, the preceding results also hold for them. However, for an unbounded family of parameterized SAREs, each of which is defined over a bounded set of domains, the scheduling problem is also undecidable.

The algorithms that we have described in this section have nevertheless given necessary and sufficient conditions for determining schedules. This is consistent with the preceding results, because the scheduling algorithms restricted themselves to a specific class of schedules: affine schedules. This has an important subtle consequence on our parallelization problem. When we seek to compile an arbitrary ALPHA program, we must account for the fact that the compiler may not be able to find such a schedule.

However, if the ALPHA program (or equivalently the SARE) was the result of an exact data flow analysis of an ACL, we have an independent guarantee of the existence of a multidimensional affine schedule — the original (completely sequential) execution order of the ACL itself.

For ALPHA, there is a simple "fall-back" strategy to compile programs that do not admit a multidimensional affine schedule, but an automatic parallelizer for ACLs does not need such a fall-back strategy.

10.7 Processor Allocation in the Polyhedral Model

Analogous to the schedule that assigns a date to every operation (i.e., each instance of each variable in the SARE), a second key aspect of the parallelization is to assign a processor to each operation. This is done by means of a processor allocation function. For all the reasons mentioned earlier, we insist that this function be determined only by analyzing the RDG, and that it be specified as a closed form function. As with schedules, we confine ourselves to affine allocation functions, defined as follows:

$$\text{alloc}(\langle S_{\text{X}}, z\rangle) = \mathcal{A}_{\text{X}}(z) = \Phi_{\text{X}} z + \phi_{\text{X}} + \Phi'_{\text{X}} p \tag{10.35}$$

where Φ_{Y} (resp. ϕ_{Y} and Φ'_{Y}) is an $a_{\text{Y}} \times n_{\text{Y}}$ (resp. $a_{\text{Y}} \times 1$ and $a_{\text{Y}} \times l$) integral matrix. Thus, the resulting image of D_{Y} by $\mathcal{M}_{\text{Y}}$ is (contained in) an a_{Y} dimensional polyhedron.

We mention that there is a difficulty with such allocation functions. In general, the number of processors that such functions imply is (usually approximated by) a polynomial function of the size parameters of the program (e.g., one would need $\mathcal{O}(n^2)$ processors for matrix multiplication). This is usually too large to be used on a pragmatic machine, and we often treat the allocation function as the composition of two functions: an affine function gives us the "virtual processors" and the second one is a mapping that specifies the emulation of these virtual processors by a "physical machine" (usually through directives that one finds in languages like HPF, such as the block or cyclic allocation of computations or data to processors). For the purpose of this section, we focus on just the first part of the complete allocation function, namely, the affine function. Obviously, this introduces certain limitations, which we discuss in Section 10.10.

For an affine allocation function, observe that two points z and z' in D_{X} are mapped to the same processor if and only if $\Phi z = \Phi z'$; that is, $(z - z')$ belongs to the null space or kernel[7] of Φ. Similarly, recall that two points z and z' in D_{X} are scheduled at the same time instant if and only if $z - z'$ is in the kernel of Λ_{X}.

[7]For any matrix M, its null space or kernel is $\text{Ker}(M) \equiv \{z | Mz = 0\}$.

Unlike the schedule, the allocation function does not have to satisfy any causality constraints, and hence there is considerable freedom in choosing it. Indeed, the only constraint that we must ensure is that it does not conflict with the schedule, in the sense that multiple operations must not be scheduled simultaneously on the same processor. This can be formalized as follows.

REMARK 10.4. *An allocation function as defined in Equation* (10.35) *is compatible with a schedule, as defined in Equation* 10.25 *if and only if:*

$$\forall z, z' \in D_X$$

$$\Phi_X z + \phi_X + \Phi'_X p = \Phi_X z' + \phi_X + \Phi'_X p \Leftrightarrow \Lambda_X z + \alpha_X + \Lambda'_X p \neq \Lambda_X z' + \alpha_X + \Lambda'_X p \quad (10.36)$$

i.e.,

$$(z - z') \in \text{Ker}(\Phi) \Leftrightarrow (z - z') \notin \text{Ker}(\Lambda) \quad (10.37)$$

Recall that the lineality space of a polyhedron P is given by A if the equality constraints in the definition of P are of the form $Az = a$ for some constant vector a. Then the preceding constraint is equivalent to the following.

REMARK 10.5. *For a variable whose domain, D_X, has a lineality space given by A_X, an allocation function as defined in Equation* (10.35) *is compatible with a schedule as defined in Equation* (10.25) *if and only if the matrix* $\begin{bmatrix} \Lambda_X \\ \Phi_X \\ A_X \end{bmatrix}$ *is of full column rank.*

Once again, we have expressed the required constraints in a compact form independent of the size of the domains. Also note that we have a separate constraint for each variable, X (i.e., we assume that there is no possibility of conflict between the instances of different variables). This is because only finitely many variables there are in the SARE, and any potential conflict can be easily resolved by "serializing" among the (finitely many) conflicting operations.

To choose among the large number of potential allocation functions in the feasible space defined by the preceding constraints we may use two notions of cost. The first and natural one is the number of processors. This may be formalized as simply the number of integer points in the union of the images of each of the variable domains, D_X, by the respective allocation functions, $\mathcal{A}_X$. Note that because this is in general a polynomial of the size parameters, we cannot use linear programming methods but have to take recourse of nonlinear optimization. A second criterion is the communication engendered by the allocation (indeed, one often seeks to optimize this instead of the number of processors, because the latter can be later changed by the virtual-to-physical mapping). The polyhedral model provides us with a very clean way of reasoning about communication. Consider a (self) dependence $(z \to Az + a)$ in an SARE. Now, if the allocation function is given by 35, the computation $X[z]$ engenders a communication to processor $\mathcal{A}_X(Az + a)$ to $\mathcal{A}_X(z)$, that is, a distance of $\Phi_X(z - Az - a)$. Note that for SUREs, this is a constant (since A is the identity). This provides us with a quantitative measure of the communication, and can be used to choose the allocation function optimally.

10.8 Memory Allocation in the Polyhedral Model

The third key aspect of the static analysis of SAREs is the allocation of operations to memory locations. In this section we first introduce some basic machinery and then formulate the constraints that a memory allocation function must satisfy. We use the forward substitution program (Figure 10.6)

```
system ForwardSubstitution : { N | N>1 }
               ( A : { i,j | 0<j<i<=N } of real;
                 B : { i   | 0<i<=N } of real)
          returns ( X : { i   | 0<i<=N } of real );
var
  f : {i,j | (2,j+1)<=i<=N; 0<=j} of real;
let
  f[i,j] = case
        {| j=0} : 0[];
        {| 1<=j} : f[i,j-1] + A * X[j];
          esac;
  X[i] = case
        {| 2<=i} : B[i] - f[i,i-1];
        {| i=1} : B[i];
         esac;
tel;
```

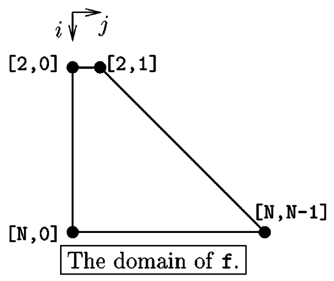


FIGURE 10.6 Forward substitution program with reduction replaced by a series of binary additions.

as a running example. As with the schedule and the processor allocation function, the memory allocation is also an affine function, defined as follows:

$$\text{Mem}(\langle S_Y, z\rangle) = \mathcal{M}_Y(z) = \Pi_Y z + \pi_Y + \Pi'_Y p \tag{10.38}$$

Here, $\Pi_Y (\pi_Y$ and Π'_Y, respectively) is an $(n_Y - m_Y) \times n_Y$ $((n_Y - m_Y) \times 1$ and $(n_Y - m_Y) \times l$, respectively) integral matrix. Thus, the resulting image of D_Y by $\mathcal{M}_Y$ is (contained in) an $(n_Y - m_Y)$ dimensional polyhedron (i.e., m_Y dimensions are "projected out"). For simplicity in the analysis presented here, we assume henceforth that π_Y and Π'_Y are both 0.

As with the processor allocation, a memory allocation function is characterized by null space of Π_Y, and can be completely specified by means of m_Y constant vectors, ρ_i for $i = 1, \ldots, m_Y$, that form a basis for Ker(M), and which can be unimodularly completed (i.e., an $n_Y \times n_Y$ unimodular[8] matrix exists whose first m_Y columns are $\rho_1, \ldots, \rho_{m_Y}$). One can visualize that an index point z is mapped to the same memory location as $z + \sum_i \mu_i \rho_i$, for any integer linear combination of the ρ_i's.

Example 10.3

For the forward substitution program (Figure 10.6) we may propose that the variable **f** be allocated to a 1-dimensional memory by the projection $M_{\texttt{f}} = [1, -1]$, which allocates **f [i, j]** to memory location **i** − **j**. This allocation function is specified by the projection vector $\rho_{\texttt{f}} = [1, 1]$. The image of the domain of **f** by the projection is {**m**|**1** <= **m** <= **N**} (i.e., a vector of length **N**).

Alternatively, we may propose that all the points in the domain of **f** be allocated to a single scalar (i.e., a projection of its domain to $\mathcal{Z}^0$). Here the projection vectors are the two unit vectors [0, 1] and [1, 0].

Observe that the choice of the projection vectors is not unique. In the first case, [−1, 1] is also a valid projection, and in the second case the columns of any 2 × 2 unimodular matrix are valid projections.

We now study the validity of memory allocation functions. Because the memory allocation is, in general, a many-to-one mapping, certain (in fact, most) values can be overwritten as the computation proceeds. We need to ensure that no value is overwritten before all the computations that depend on it are themselves executed, formalized as follows:

[8]A square integer matrix M is said to be unimodular if and only if $\det(M) = \pm 1$. Hence, its inverse exists and is integral.

Definition 10.5. *A memory function* Mem_Y *is valid if and only if for all* $z \in D_Y$, *the* next write *after* $t_Y(z)$ *into the memory location* $\mathrm{Mem}_Y(z)$ *occurs after* all uses *of* $Y[z]$.

Note that the preceding definition uses the term after, which clearly depends on the schedule. Thus, the memory allocation function, unlike the processor allocation, has an intricate interaction with the schedule. Nevertheless, we have a condition analogous to the one for the processor allocation, but which gives us only necessary conditions.

Remark 10.6. *For a variable whose domain* D_Y *has a lineality space given by* A_Y, *a memory allocation function as defined in Equation* (10.38) *is valid if the matrix* $\begin{bmatrix} \Lambda_Y \\ \Pi_Y \\ A_Y \end{bmatrix}$ *is of full column rank.*

We also assume that for $i = 1, \dots, m_Y$, the vectors ρ_i's are such that $\Lambda_Y \rho_i$ is lexicographically positive. Note that this does not impose any loss of generality: we may always choose the sign of the ρ_i's appropriately.

In the remainder of this section, we first formalize three notions: the next-write function, the usage set, and the lifetime; by using these we formulate the validity constraints that the memory allocation function must satisfy.

10.8.1 Next-Write Function

Now consider, for any $z \in D_Y$, the next point z' that overwrites the memory location of $Y[z]$. We want to express z' as a function of z.

Definition 10.6. *For any point* $z \in D_Y$, *the* next write *is the earliest scheduled point that satisfies the following constraints:*

$$\left\| \begin{array}{lcl} \Pi_Y z & = & \Pi_Y z' \\ z & \in & D_Y \\ z' & \in & D_Y \\ \Lambda_Y z & \prec & \Lambda_Y z' \end{array} \right. \tag{10.39}$$

We write it as $z + \sigma_Y(z)$. *We define* $\Lambda_Y \sigma_Y(z)$ *which is the time interval between the computation of* $Y[z]$ *and its destruction as the* overwrite window *of* $Y[z]$.

Note that the constraints (10.39) define a set of k disjoint polyhedra, parameterized by z, and hence the next write can be obtained by resolving k parametric integer programming problems [15]. Hence, $\sigma_Y(z)$ is a piecewise affine function of z.

Now, observe that if we drop the constraints that z and z' must belong to D_Y from (10.39), the solution is simply $z + \sigma_Y$ for some constant vector, σ_Y, which we call the *next-write vector* for Y. It is clear that $\Lambda_Y \sigma_Y$ is a lower bound on the overwrite window of $Y[z]$; and for points "far" from the boundaries, this approximation is exact. Furthermore, because our domains are large enough, such points exist (indeed, are the rule instead of the exception). The preceding ideas are formulated in the following proposition and illustrated through the subsequent examples.

Proposition 10.1. $\sigma_Y(z)$ *is a piecewise affine function, such that at points sufficiently far from the boundaries of* D_Y *its value is a constant,* σ_Y.

Example 10.4

For the forward substitution program (Figure 10.6), let the memory allocation function for $\mathbf{f}$ be specified by $\Pi_{\mathbf{f}} = [1, -1]$, and the schedule be given by $t_{\mathbf{f}}(i, j) = \begin{pmatrix} i + j \\ j \end{pmatrix}$, i.e.,

$\Lambda_{\mathbf{f}} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$, $\alpha_{\mathbf{f}} = 0$. Recall that the schedule may be visualized as follows: the first row represents the family of lines $i + j = h$ corresponding to the first time dimension (i.e., the hth hour), and the second row specifies the minutes $(j = m)$ within each hour (i.e., the second time dimension). Then, for a point $z \in D_{\mathbf{f}}$ the next write into the same memory location as z is $\mathbf{f}(z')$, where:

$$z' = \begin{cases} \{i, j \mid 1 < i < N; 0 \le j < i\} & : \quad z + \begin{pmatrix} 1 \\ 1 \end{pmatrix} \\ \{i, j \mid i = N; 0 \le j < i\} & : \quad \text{undefined} \end{cases}$$

On the other hand, if (for the same schedule) the memory allocation is specified by the two unit vectors (i.e., all the points in the domain of $\mathbf{f}$ are allocated to a single scalar), then:

$$z' = \begin{cases} \{i, j \mid 2 < i \le N; 0 \le j < i\} & : \quad z + \begin{pmatrix} -1 \\ 1 \end{pmatrix} \\ \{i, j \mid i = 2; j = 0\} & : \quad z + \begin{pmatrix} 1 \\ 0 \end{pmatrix} \\ \{i, j \mid 2 < i; j = i - 1; 2i - 1 \le N\} & : \quad z + \begin{pmatrix} i \\ -i + 1 \end{pmatrix} \\ \{i, j \mid 2 < i; j = i - 2; i + j \le N - 1\} & : \quad z + \begin{pmatrix} j + 1 \\ 0 \end{pmatrix} \\ \{i, j \mid 2 < i; j = i - 2; i + j \ge N\} & : \quad z + \begin{pmatrix} 0 \\ i - N + 1 \end{pmatrix} \\ \{i, j \mid j = i - 1 = N - 1\} & : \quad \text{undefined} \end{cases}$$

Observe how $z' - z = \sigma_{\mathbf{f}}(z)$ is a piecewise affine function of i and j, and note how the first clause corresponds to interior points whereas all the other clauses are boundary cases. They occur on subdomains with equalities, and it can be verified that, for each $z \in D_{\mathbf{f}}$, there does not exist $z'' \in D_{\mathbf{f}}$ such that:

$$\Lambda_{\mathbf{f}} \left(z + \begin{pmatrix} -1 \\ 1 \end{pmatrix} \right) \prec \Lambda_{\mathbf{f}} z'' \prec \Lambda_{\mathbf{f}} z'$$

In other words, the value of $\sigma_{\mathbf{f}}$ due to the first clause is a lower bound on those predicted by the other clauses. Thus, for the two allocation functions, the *next-write vector* $\sigma_{\mathbf{f}}$ is $\begin{pmatrix} 1 \\ 1 \end{pmatrix}$ and $\begin{pmatrix} -1 \\ 1 \end{pmatrix}$, respectively. *end of example*

10.8.2 Usage Set

Assume that the (sub) expression $Y.(z \to Mz + m)$ occurs on the rhs of the equation defining X. The context where it appears in the ALPHA program (e.g., within a case or a restriction) defines a domain $D'_{\mathrm{X}} \subseteq D_{\mathrm{X}}$ where the dependency is said to hold. This information can be readily computed from the program text and is written as follows (F serves to name the dependency):

$$F : \forall z \in D'_{\mathrm{X}}, \quad \mathrm{X}[z] \to \mathrm{Y}[M_z + m] \tag{10.40}$$

Let us also use $\mathcal{F}_Y$ to denote the set of dependencies on Y (i.e., the set of dependencies where the variable on the rhs is Y). For any $z \in D_{\mathrm{Y}}$, we are interested in determining the users of $Y[z]$, with respect to a dependency, F. Note that because the affine function $Mz + m$ may not be invertible, there may be multiple users. They constitute some subset of D_X (or more precisely, of D'_X, because the dependency holds only there). Let us call this set D_X^F. The following properties must be satisfied:

$$\left\| \begin{array}{l} z \ = Mz' + m \\ z \ \in D_{\mathrm{Y}} \\ z' \in D'_{\mathrm{X}} \end{array} \right. \tag{10.41}$$

This can be viewed as the set of points $\begin{pmatrix} z \\ z' \end{pmatrix}$ belonging to a polyhedron. Alternatively, and equivalently, it can also be viewed as a set of points z' belonging to a polyhedron parameterized by z (and of course the system parameters), denoted by $D_{\mathrm{X}}^F(z)$. It is this interpretation that we shall use.

Example 10.5
For the forward substitution program (Figure 10.6), the dependencies on variable **f** are:

$$\begin{array}{lll} F_1 : & \{i, j, N \mid 1 \le j < i \le N\} & : \ \mathbf{f}[i, j] \to \mathbf{f}[i, j-1] \\ F_2 : & \{i, N | 2 \le i \le N\} & : \ \mathbf{X}[i] \to \mathbf{f}[i, i-1] \end{array}$$

Consider the first dependency, F_1. For any point $z = \begin{pmatrix} i \\ j \end{pmatrix} \in D_{\mathtt{f}}$, the set of its users $\mathbf{z}' = \begin{pmatrix} i' \\ j' \end{pmatrix}$ with respect to F_1 is:

$$\begin{aligned} D_{\mathbf{f}}^{F_1}(z) &= \{i', j', i, j, N \mid i = i', j' = j + 1\} \\ &\quad \cap \{i', j', i, j, N \mid 2 \le i \le N; 0 \le j < i\} \\ &\quad \cap \{i', j', i, j, N \mid 1 \le j' < i' \le N\} \\ &= \{i', j', i, j, N \mid 2 \le i' = i \le N; j' = j + 1; 0 \le j < i - 1\} \end{aligned}$$

This is viewed as a two-dimensional polyhedron over i' and j', parameterized by i, j (and N). Observe that the very last constraint cited earlier, namely, $j < i - 1$, correctly implies that points on the $i = j + 1$ are not used by any computation (with respect to the dependency F_1).

For the dependency, F_2, we proceed similarly (recall that we are considering usage by **X**, so z is 2-dimensional, and z' is 1-dimensional):

$$\begin{aligned} D_{\mathbf{f}}^{F_2}(z) &= \{i', i, j, N \mid i = i', j = i' - 1\} \\ &\quad \cap \{i', i, j, N \mid 2 \le i \le N; 0 \le j < i\} \\ &\quad \cap \{i', i, j, N \mid 2 \le i' \le N\} \\ &= \{i', i, j, N \mid i = i' = j + 1; 2 \le i \le N\} \end{aligned}$$

Observe again, that only values of **f** computed on the $i = j + 1$ boundary are used (with respect to this dependency), and this is correctly predicted by our computation of $D_{\mathbf{f}}^{F_2}(z)$. Furthermore, the set of points that use **f[i, i−1]** is a singleton in the domain of **X**, namely, **X[i′]** where $\mathbf{i} = \mathbf{i}'$.

To illustrate the case when the dependency function is not invertible, consider the following dependency on **X**:

$$F_3 : \quad \{i, j, N \mid 1 \le j < i \le N\} \quad : \quad \mathbf{f}[i, j] \to \mathbf{X}[j]$$

Here, $z = (i)$ is one-dimensional, and $z' = \begin{pmatrix} i' \\ j' \end{pmatrix}$ is two-dimensional, and we have:

$$\begin{aligned} D_{\mathbf{X}}^{F_3}(z) &= \{i', j', i, N \mid i = j'\} \\ &\quad \cap \{i', j', i, N \mid 1 \leq i \leq N\} \\ &\quad \cap \{i', j', i, N \mid 1 \leq j' < i' \leq N\} \\ &= \{i', j', i, N \mid j' = i;\ 1 \leq i < N;\ i < i' \leq N\} \end{aligned}$$

Observe again, that this correctly predicts that the first $N - 1$ values of **X** are used. Viewing this as a family of two-dimensional polyhedra parameterized by i (and N) we see that a given **X[i]** is used by multiple points **f [i′, j′]** such that $j' = i$ and $i < i' \leq N$. *end of example*

10.8.3 Lifetime

Because we are given a schedule for the program, we know that for any $z \in D_{\mathrm{Y}}$, $Y[z]$ is computed at (the k-dimensional) time instant $\Lambda_{\mathrm{Y}} z + \alpha_{\mathrm{Y}}$, and for any $z' \in D_{\mathrm{X}}^F(z)$, $X[z']$ is computed at time $\Lambda_{\mathrm{X}} z' + \alpha_{\mathrm{X}}$. Hence, the time between the production of $Y[z]$ and its use by $X[z']$ is simply $\Lambda_{\mathrm{X}} z' + \alpha_{\mathrm{X}} - \Lambda_{\mathrm{Y}} z - \alpha_{\mathrm{Y}}$. We have the following definition, where $\mathrm{Lmax}_{x \in S} f(x)$ denotes the lexicographic maximum of $f(x)$ over the set S.

DEFINITION 10.7. *The partial lifetime, $d_{\mathrm{Y}}^F(z)$, of $Y[z]$ with respect to the dependency F is:*

$$d_{\mathrm{Y}}^F(z) = \underset{z' \in D_{\mathrm{X}}^F(z)}{\mathrm{Lmax}} (\Lambda_{\mathrm{X}} z' + \alpha_{\mathrm{X}} - \Lambda_{\mathrm{Y}} z - \alpha_{\mathrm{Y}}) \tag{10.42}$$

The (total) lifetime of $Y[z]$ is:

$$d_{\mathrm{Y}}(z) = \underset{F \in \mathcal{F}_Y}{\mathrm{Lmax}}\, d_Y^F(z) \tag{10.43}$$

The lifetime of the entire variable Y is:

$$d_{\mathrm{Y}} = \underset{z \in D_{\mathrm{Y}}}{\mathrm{Lmax}}\, d_{\mathrm{Y}}(z) \tag{10.44}$$

We observe that $d_{\mathrm{Y}}^F(z)$, the lexicographic maximum over a polyhedron parameterized by z, is a piecewise affine function of z (and the system parameters). Furthermore, $d_{\mathrm{Y}}(z)$, the lexicographic maximum of a finite number of such functions (note that $\mathcal{F}_{\mathrm{Y}}$ is a finite set of dependencies), is also a piecewise affine function of z. Hence, d_{Y} is the lexicographic maximum, not of an affine, but a piecewise affine, cost function over the domain D_{Y}. This can be expressed as the resolution of a finite number of parametric integer programming problems (one for each of the "pieces" of the cost function), and hence is the lexicographic maximum of a finite number of piecewise affine functions. As a result, we have the following:

PROPOSITION 10.2. *The $d_{\mathrm{Y}}(z)$ is a piecewise affine function of z and the system parameters.*

Example 10.6

Continuing with the forward substitution program, let the respective schedules for **f** and **X** be given by $t_{\mathbf{f}}(i, j) = i + j$ and $t_{\mathbf{X}}(i) = 2i$. The lifetime of any $\mathbf{f}(i, j)$ is especially easy to determine because the usage sets are singletons. Because $\mathbf{f}(i, j)$ is computed at date $i + j$, its partial lifetime with respect to F_1 is the lexicographic maximum of $t_{\mathbf{f}}(i', j') - t_{\mathbf{f}}(i, j)$ over $D_{\mathbf{f}}^{F_1}(z)$, and because $i = i'$ and $j' = j + 1$:

$$d_f^{F_1}(z) = 1$$

Similarly, for F_2 the usage sets are again singletons, and it is easy to see that **f[i, j]** for $2 \leq i = j + 1 \leq N$ is precisely used by **X[i′]**, where $i' = i$. These points are computed at $i + j$ and $2i'$, respectively, whose difference is always 1. Hence:

$$d_f^{F_2}(z) = 1$$

If we choose the alternative (sequential) schedule given by $t_f(i, j) = \begin{pmatrix} i + j \\ j \end{pmatrix}$ and $t_X(i) = \begin{pmatrix} 2i \\ i \end{pmatrix}$, the lifetime is now a 2-dimensional function, and can be easily shown to be:

$$\begin{aligned}
d_f^{F_1}(z) &= \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}\left(z + \begin{pmatrix} 0 \\ 1 \end{pmatrix}\right) - \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} z \\
&= \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \\
d_f^{F_2}(z) &= \begin{pmatrix} 2 \\ 1 \end{pmatrix} z' - \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} z \\
&= \begin{pmatrix} 2 \\ 1 \end{pmatrix} (i) - \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} i \\ j \end{pmatrix} \\
&= \begin{pmatrix} i - j \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}
\end{aligned}$$

Note that the final simplification is possible because $D_f^{F_2}(z)$ is nonempty only when $i = j + 1$.

end of example

10.8.4 Validity Conditions for the Memory Allocation Function

We now develop necessary and sufficient conditions on the vectors ρ_i's so that the basic constraint on memory allocation function (Definition 10.5) holds. Recall that:

- The computation that overwrites $Y[z]$ is $Y[z + \sigma_Y(z)]$.
- The set of users of $Y[z]$ with respect to a dependency, F, is $D_X^F(z)$.
- Y[z] is computed at time $\Lambda_Y z + \alpha_Y$.
- $Y[z + \sigma_Y(z)]$ is computed at time $\Lambda_Y(z + \sigma_Y(z)) + \alpha_Y$.
- For any $z'' \in D_X^F(z)$, $X[z'']$ is computed at time $\Lambda_X z'' + \alpha_X$.

Hence, the condition imposed by Definition 10.5 holds in the context of F if and only if:

$$\forall z \in D_Y \text{ and } \forall z'' \in D_X^F(z)$$

$$\Lambda_X z'' + \alpha_X - \Lambda_Y z - \alpha_Y \preceq \Lambda_Y \sigma_Y(z) \tag{10.45}$$

Note that this inequality is nonstrict — we allow $Y[z]$ to be read at the same instant that its memory location is overwritten. This assumes that the machine architecture provides such synchronization. If this is not the case, we could insist on strict inequality, without changing the nature of the analysis.

Because the size of the sets D_Y and $D_X^F(z)$ may be arbitrary, an unbounded number of constraints may need to be satisfied. However, the fact that $D_X^F(z)$ is a polyhedron allows us to exploit the power of the polyhedral model, namely, that (10.45) holds at all $z'' \in D_X^F(z)$ if and only if it is satisfied by the points of $D_X^F(z)$ that maximize, in the lexicographic order, $\Lambda_X z''$. By using Definition 10.7, this reduces to:

$$\forall z \in D_Y, d_Y^F(z) \preceq \Lambda_Y \sigma_Y(z) \tag{10.46}$$

This defines the constraints on the overwriting of $Y[z]$ with respect to a single dependency. To determine the necessary and sufficient conditions on the memory allocation function, we need to extend the preceding analysis to $\mathcal{F}_Y$. Hence, a memory allocation function is valid if and only if:

$$\forall z \in D_{\mathrm{Y}}$$

$$d_{\mathrm{Y}}(z) \preceq \Lambda_{\mathrm{Y}}\sigma_{\mathrm{Y}}(z) \tag{10.47}$$

Now, observe that $d_{\mathrm{Y}}(z)$ is a piecewise affine function of z, and hence we can separate (10.47) into a finite number of similar constraints over disjoint subsets, D^i_{Y}, of D_{Y}, each of which has a single affine function $d^i_{\mathrm{Y}}(z)$, on the left. As a result, we can seek a separate memory allocation function for each subdomain, D^i_{Y}. Our experience has shown that piecewise linear allocations are often very useful (only certain subdomains may need larger memory, whereas for others, additional projection dimensions may exist). Alternatively, we could also use a single homogeneous allocation function over the entire domain, D_{Y}, by ensuring that $d_{\mathrm{Y}} \preceq \Lambda_{\mathrm{Y}}\sigma_{\mathrm{Y}}$, which is a safe approximation. Thus, the improvement due to piecewise linear allocations comes at a price of some overhead, and increased compilation time. Our current implementation uses a homogeneous allocation by default and performs the refined analysis as a user-specified option.

Like the lifetime function, $\sigma_{\mathrm{Y}}(z)$ is also a piecewise affine function of z. Moreover, $\sigma_{\mathrm{Y}}(z)$ is a constant (see Proposition 10.1.) almost everywhere except near the boundaries of D_{Y}, and hence of the corresponding D^i_{Y}'s. This constant (namely, the next-write vector, σ_{Y}) is a lower bound on $\sigma_{\mathrm{y}}(z)$. We therefore approximate $\sigma_{\mathrm{Y}}(z)$ by σ_{Y} and henceforth use the following sufficient condition:

$$\forall z \in D^i_{\mathrm{Y}}, d_{\mathrm{Y}}(z) \preceq \Lambda_{\mathrm{Y}}\sigma_{\mathrm{Y}} \tag{10.48}$$

Finally, we again use the key idea of the polyhedral model, namely, that (10.48) holds for all points $z \in D^i_{\mathrm{Y}}$ if and only if it holds for the points in D^i_{Y} that maximize the lhs (i.e., if and only if):

$$d^i_{\mathrm{Y}} \preceq \Lambda_{\mathrm{Y}}\sigma_{\mathrm{Y}} \tag{10.49}$$

where d^i_{Y} is the lifetime of Y over the subdomain D^i_{Y}.

Hence, we have reduced the validity conditions for a memory allocation function to the satisfaction of a finite number of linear constraints. The following proposition relates this constraint on σ_{Y} to the ρ_i's.

PROPOSITION 10.3. *A set of projection vectors* (ρ_i) *defines a valid memory projection for a variable* Y *if and only if for all integral linear combinations* η, *of* ρ_i*'s:*

$$0 \prec \Lambda_{\mathrm{Y}}\eta \Rightarrow d_{\mathrm{Y}} \preceq \Lambda_{\mathrm{Y}}\eta$$

PROOF. *If part.* If the ρ_i's are such that (10.49) holds $\forall z \in \mathbb{Z}^{n_{\mathrm{Y}}}$, let if possible $\eta = \sum_{i=1}^{m_{\mathrm{Y}}} \mu_i\rho_i$, $\mu_i \in \mathbb{Z}$ such that $0 \prec \Lambda_{\mathrm{Y}}\eta$ and $d_{\mathrm{Y}} \preceq \Lambda_{\mathrm{Y}}\eta$. Then, for some $z \in \mathbb{Z}^{n_{\mathrm{Y}}}$, $0 \prec \Lambda_{\mathrm{Y}}\eta \prec d_{\mathrm{Y}} \preceq \Lambda_{\mathrm{Y}}\sigma_{\mathrm{Y}}$. This implies that η and σ_{Y} are distinct, and hence $Y[z+\eta]$ and not $Y[z+\sigma_{\mathrm{Y}}]$ is the next write into the same memory cell as $Y[z]$, a contradiction.

Only if part. Obviously, σ_{Y} is a linear combination of the ρ_i's such that $0 \prec \Lambda_{\mathrm{Y}}\sigma_{\mathrm{Y}}$.

Finally, as with the memory and the processor allocation functions, a well-defined notion of optimality exists for the memory allocation, namely, the volume of memory that is used for a given schedule. Indeed, because this is a polynomial function of the size parameters, the most important criterion to minimize is the number of linearly independent projection vectors, ρ_i. It can be shown (constructively) that this is equal to one plus the number of leading zeroes in d_{Y}, the lifetime vector for the variable.

10.9 Code Generation

In the previous three sections we have addressed some essential issues of analyzing SAREs. Specifically, we have given methods to determine three types of functions: schedules to assign a date to each operation, processor allocation to assign operations to processors and memory allocations to store intermediate results. These functions may be chosen so as to optimize certain criteria under certain assumptions, but note that the optimization problems are far from resolved, and indeed many interesting problems are open. Our focus in this section is to consider a different problem orthogonal to the choice of these functions. This is the problem of code generation — given the preceding three functions, how do we produce parallel code that implements these choices?

By insisting that code generation is independent of the choice of the schedule and allocation functions, we achieve a separation of concerns; for example, the methods described here may be used to produce either sequential or parallel code for programmable (i.e., instruction-set) processors, or even VHDL descriptions of application-specific or nonprogrammable hardware that implements the computation specified by the SARE.

We first show how to implement or compile an SARE (ALPHA program) in the complete absence of any static analysis. In other words, we present a simple operational semantics for ALPHA and describe how to produce naive code. This highlights the separation of concerns mentioned earlier, and also provides us with a baseline, fallback implementation, which we are able to progressively improve, as and when static analysis is available. Next, we shall see how the schedule (plus the processor allocation if available) can be used to enable us to generate efficient imperative but memory inefficient code; finally, we describe how the memory allocation function can be used to perform very simple and minor modifications of this code to produce memory-efficient code.

10.9.1 Polyhedron Scanning

Before we enter into the details, we develop an important tool that we need, namely, a resolution of the polyhedron scanning problem, defined as follows.

Problem 10.2. *Given a (possibly parameterized) polyhedron, $\mathcal{P}$ construct a loop nest to visit the integral points in $\mathcal{P}$ in lexicographic order of the indices.*

This problem has been well studied by parallelizing compiler researchers, and the solution is based on the well-known technique of Fourier–Motzkin elimination. This can be done by what is called *separation of polyhedra* and the key step here involves projecting a $k+1$-dimensional polyhedron $P(z_1, \dots, z_k, z_{k+1}) \equiv \{z \mid Az \geq b\}$ onto its first k indices. To do this, we partition each row of A into one of three categories: those that (1) are independent of, (2) provide a lower bound on, or (3) provide an upper bound on z_{k+1}. From every pair of upper and lower bound inequalities, we eliminate z_{k+1} to get a new inequality that does not include z_{k+1}, and to this list we add the ones from the first category. This yields an alternative and equivalent representation of the polyhedron P. By successively eliminating $z_k, z_{k-1}, \dots$ we may rewrite the constraints of the polyhedron in a sequence where the indices are separated. As an example, consider the polyhedron, $\{\mathbf{i,j,k} \mid \mathbf{i} >= \mathbf{0};\ -\mathbf{1}+\mathbf{M} >= \mathbf{0};\ -\mathbf{j}+\mathbf{N} >= \mathbf{0};\ \mathbf{k} >= \mathbf{0};\ \mathbf{i}+\mathbf{j}-\mathbf{k} >= \mathbf{0}\}$, in the context of parameters constrained by, $\{\mathbf{N, M} \mid \mathbf{N, M} > \mathbf{0}\}$, and let us seek to visit all its integer points in the scanning order, $\{\mathbf{j,k,i}\}$. Polyhedron separation in this order is achieved by first eliminating **i**, then **k** and finally **j**. This yields the format shown on the left that follows, from which we can construct the loop nest shown alongside, by simply "pretty-printing" it — each line introduces a new index, and its lower and upper bounds are evident from the inequalities on the line:

```
{j| 0<=j<=N} ::                    for (j=0; j++; j<=N)
  {k,j| 0<=k<=j+M} ::                for (k=0; k++; k<=j+M)
    {i,k,j| 0<=i<=M; i>=k-j}           for (i=max(0,k-j); i++; i<=M)
```

If the dual (generator) representation of the polyhedron is available, many optimizations can be performed in this basic step (instead of pairwise combination of all the lower and upper bound constraints during each projection step, we can simply add a pair of rays to the generator representation).

It is important to note here that the Fourier–Motzkin elimination method is exact for real and rational polyhedra, but nonexact for integral polyhedra. This is because, in general, the projection of an integral polyhedron is not necessarily also in integral polyhedron. However, well-known methods exist to make the method exact for integral polyhedra. Polyhedron scanning is an important component of code generation.

10.9.2 ALPHA Operational Semantics: Naive Code Generation

In contrast with the denotational semantics given in Section 10.5, we now provide a simple operational semantics, one that allow us to develop our fallback strategy for executing ALPHA programs in the absence of any static analysis.

10.9.2.1 Semantics of Expressions

Recall that because ALPHA expressions denote mappings from indices to values, operational semantics may be developed if we first define this mapping. We do so as a simple interpreter, in terms of a function Eval : $\langle \mathbf{Exp} \rangle \times \mathcal{Z}^n \rightarrow$ Type. We first introduce a mechanism for computing and propagating errors (although this is not strictly necessary):

$$\text{Eval}(\langle \mathbf{exp} \rangle, z) = \begin{cases} \text{Eval}'(z) & \text{if } z \in \text{Dom}(\mathbf{exp}) \\ \bot & \text{otherwise} \end{cases}$$

Eval$'$ is defined recursively, with six cases corresponding to the six syntax rules (see Table 10.1) for ALPHA expressions:

$$\begin{aligned}
\text{Eval}'(\langle \mathbf{Const} \rangle, z) &= C \\
\text{Eval}'(\langle \mathbf{E1} \rangle\ op\ \langle \mathbf{E2} \rangle, z) &= \text{Eval}'(\langle \mathbf{E1} \rangle, z) \oplus \text{Eval}'(\langle \mathbf{E2} \rangle, z) \\
\text{Eval}'(\mathbf{case} \ldots \langle \mathbf{Ei} \rangle \ldots \mathbf{esac}, z) &= \begin{cases} \vdots & \\ \text{Eval}'(\langle \mathbf{Ei} \rangle, z) & \text{if } z \in \text{Dom}(\langle \mathbf{Ei} \rangle) \\ \vdots & \end{cases} \\
\text{Eval}'(\texttt{D} : \langle \mathbf{E} \rangle, z) &= \text{Eval}'(\langle \mathbf{E} \rangle, z) \\
\text{Eval}'(\langle \mathbf{E} \rangle.f, z) &= \text{Eval}'(\langle \mathbf{E} \rangle, f(z)) \\
\text{Eval}'(\langle \mathbf{Var} \rangle, z) &= \text{EvalVar}(z)
\end{aligned}$$

10.9.2.2 Semantics of Equations

The denotational semantics of ALPHA equations essentially specify that equations denote additions to a store of definitions (such semantics are fairly standard and hence were not described in Section 10.5). The corresponding operational semantics are also straightforward, namely, that every equation causes a function to be defined and added to the store, and the function body is the operational semantics of the expression on the rhs of the equation:

$$\text{Eval}(\mathbf{Var}{=}\langle \mathbf{Exp} \rangle) = (\text{defun EvalVar}(\text{Eval}(\langle \mathbf{Exp} \rangle z)$$

10.9.2.3 Semantics of Programs

ALPHA programs are systems, and they denote mappings from input variables to output variables. The corresponding operational semantics are also straightforward. They require a mechanism to specify the following steps:

1. Read input variables.
2. Compute (local) and output variables.
3. Write output variables.

This can be achieved by a simple set of loops to scan the domains of each output variable, **Var**, and at each point, call the function **EvalVar**.

This essentially produces an interpreter, and suffers from the standard drawbacks of interpretive implementations of functional languages, namely, the recomputation of previously evaluated values (e.g., a program to compute the nth Fibonacci number takes time exponential in n).

However, we may modify the operational semantics to exploit a well-known strategy used in functional languages, namely, "caching" or tabulation. Instead of recomputing previously evaluated values, we cache them by storing them in a table (the Eval$'$ function is modified so that instead of making a recursive call, it first checks whether the value has been previously evaluated and stored). The recursive call is made only if this is not so, and when it returns the value is cached into the table. In ALPHA this requires a very simple modification, namely, the allocation of memory for the tables. This memory corresponds to the size of the domains and can be allocated as an array. The resulting code has the following structure:

- Declarations of multidimensional array variables correspond to each local and output variable in the program.
- The definition of the functions EvalVar is given for all variables. For the local and output variables this follows the semantics as defined earlier. For input variables, such a function simply reads the input array at the specified address.
- A main program consists of one set of loops (produced by polyhedron scanning) that visits the domain of each of the output variables, and calls the corresponding EvalVar function at each point (the order of each of the output variables, as well the order in which the domains are scanned is immaterial).

Finally, we point out that very simple and implicit parallelism exists, and the code may be very easily modified to run in a demand-driven manner on a multithreaded machine — all the pointwise operators are strict, and this is where a synchronization is necessary to ensure that arguments are available before they are combined to produce results. Everything else, including the function calls, can be done in parallel.

10.9.3 Exploiting Static Analysis: Imperative Code Generation

The code produced in the absence of static analysis suffers from two main drawbacks. Because a function call exists for each evaluation, a considerable overhead of context switching is present. Furthermore, the memory allocated to each domain corresponds to the entire domain, and this code is memory inefficient. We next examine how the results of static analysis, namely, the schedule and the processor and memory allocation functions can be used to improve the code. The main idea is summarized as follows:

1. We rewrite the SARE, by means of an appropriate COB, such that the first k indices of the domains of all variables are the k time dimensions; the remaining indices are to be interpreted as virtual processors.

2. We produce declarations of multidimensional arrays to store the variables of the program. For each variable, we allocate memory corresponding to the smallest rectangular box enclosing its domain.
3. We next generate code that visits the union of the domains of all the variable in the lexicographic order imposed by the first k indices (loops scanning other indices, if any, are annotated as **forall** loops). At each point visited, we simply evaluate the body of the SARE without recursive calls, and without testing whether the arguments have been previously evaluated (the schedule guarantees that this is unnecessary).
4. Note that the code that we produce here is single assignment form: each computed value (representing an operation in the original program) is stored in a distinct memory location. Suppose that in addition to the schedule and the processor allocation we also have for each variable of the SARE, a memory allocation function $\mathcal{M}_i$ that gives a valid address that the result of $S_i[z]$ is stored at memory address $\mathcal{M}_i(z)$. Then we may easily generate multiple assignment code by modifying the preceding code as follows:

   - Modify the preamble so that it allocates memory arrays for only (the bounding box of) $\mathcal{M}_i(D_i)$, the image of D_i by the memory allocation function.
   - Systematically replace every access — on the rhs or the lhs of any statement — to the variable S_i (i.e., an expression of the form $S_i[f(z)]$ for some affine function f) by $S_i[\mathcal{M}_i(f(z))]$.

10.9.4 Scanning a Union of Polyhedra

The key unresolved problem in the code generation algorithm outlined earlier is that of generating code to visit the points of a union of polyhedra, as required in step 3 above. We now address this problem, building on the solution to the (single) polyhedron-scanning problem seen earlier (Figure 10.7).

We note that the problem is not as easy as it seems at first glance. We cannot simply scan the individual polyhedra separately and somehow combine the solution. For example, consider the domains $\mathcal{D}_1 = \{\mathbf{i} \mid \mathbf{1} <= \mathbf{i} <= \mathbf{10}\}$ and $\mathcal{D}_2 = \{\mathbf{i} \mid \mathbf{4} <= \mathbf{i} <= \mathbf{12}\}$, with respective schedules $\Lambda_1(i) = i + 1$ and $\Lambda_2(i) = 2i$. The statements associated with points 1 and 10 of $\mathcal{D}_1$ have to be executed at logical time 2 and 11, whereas the statements associated with points 4 and 8 of $\mathcal{D}_2$ have to be executed at times 8 and 16. Because $1 < 8 < 10 < 16$, we cannot first completely scan $\mathcal{D}_1$, and then scan $\mathcal{D}_2$, and we also cannot scan $\mathcal{D}_2$ followed by $\mathcal{D}_1$. Hence, it is not possible to generate separate loops to scan both $\mathcal{D}_1$ and $\mathcal{D}_2$. These loops must be partially merged, so that the statements are executed in the order given by the schedule.

The simplest solution to scanning a union of polyhedra uses a perfectly nested loop to scan a convex superset of this union. This superset may be either the bounding box of the domain — Figure 10.8(a) — or the convex closure of the domain — Figure 10.8(b). However, because the domain scanned by this loop is also a superset of each statement domain, we cannot unconditionally execute the statements within the loop body. Instead, each statement S_i must be guarded by conditions testing that the current loop index vector belongs to $\mathcal{D}_i$.

However, this solution yields an inefficient (albeit compact) code because of the following limitations:

- *Empty iterations* A perfectly nested loop scans a convex polyhedron. Because a domain that is a union of polyhedra may not be convex, some iterations of the loop may not execute any statements. It is often especially inefficient when a domain has an extreme aspect ratio. For example in Figure 10.8(a), only about **M** out of every **N** points are usefully visited. Depending on the parameter values (say if $\mathbf{M} \ll \mathbf{N}$) and on the complexity of the loop body, this could have a significant overhead.

(a) Statements qualified by domains

```
{i,j | 1<=i<=N; 1<=j<=M} :: S1;
{i,j | i=j; 3<=j<=N} :: S2;
```

(b) View of the domains for the case where $4 \leq M \leq N$

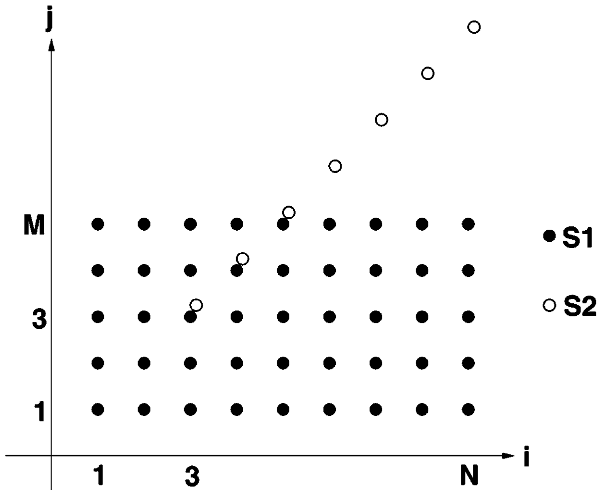


FIGURE 10.7 Systems of statements qualified by domains.

(a) bounding box	(b) convex closure

```
for (i=1; i<=N; ++i){
    for (j=1; j<=N; ++j){
        if (j<=M)
            S1;
        if (i==j && i>=3)
            S2;
    }
}
```
	```
	for (i=1; i<=N; ++i){
	  for (j=1; j<=min ((i+3N+M-4)/4, i+M-1, N); ++j){
	      if (j<=M)
	          S1;
	      if (i==j && i>=3)
	          S2;
	  }
	}
	```

FIGURE 10.8 Scanning the union of domains of Figure 10.7 by visiting points in a convex superset.

- *Control overhead* Each loop iteration must test the guards of all guarded statements, causing a significant control overhead.
- Finally, finite unions of finite parameterized polyhedra do not always admit finite convex supersets. Consider, for example, the following parameterized domains: $\{\mathbf{i} \mid 1 <= \mathbf{i} <= \mathbf{N}\}$ and $\{\mathbf{i} \mid 1 <= \mathbf{i} <= \mathbf{M}\}$; the smallest convex superset of these domains is the infinite polyhedron $\{\mathbf{i} \mid 1 <= \mathbf{i}\}$ (note that $\{\mathbf{i} \mid 1 <= \mathbf{i} <= \mathbf{max(M, N)}\}$ is not a convex polyhedron).

```
for (i = 1; i <= 2; ++i) {
    for (j = 1; j <= M; ++j) {
        S1;
    }
}
for (i = 3; i <= N; ++i) {
    for (j = 1; j <= min((-1 + i),M); ++j) {
        S1;
    }
    if (i<=M) {
        j = i;
        S1;
        S2;
    }
    if (i>=M+1) {
        j = i;
        S2;
    }
    for (j = (1 + i); j <= M; ++j) {
        S1;
    }
}
```

FIGURE 10.9 Imperfectly nested loops scanning the union of domains of Figure 10.7.

To avoid this overhead, we can separate a union of polyhedra into several distinct regions that can each be scanned using imperfectly nested loops. The resulting code is more efficient because:

- Imperfectly nested loops can scan nonconvex regions, avoiding empty iterations.
- It may be possible to choose these loops such that some statement guards become always true or always false. When a guard is always true, the guard may be removed; when a guard is always false, the entire statement can be removed. This optimization increases the ratio of the number of executed statements to the number of tested guards, and thus reduces the control overhead. This can be carried through to its logical limit to yield a code with no guards.

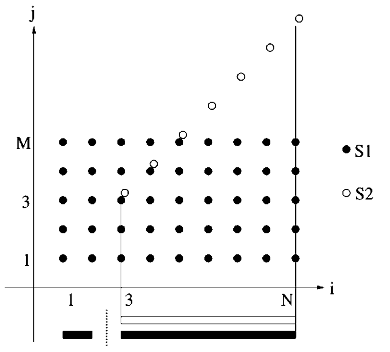
However, the efficiency is obtained at the expense of code size for two reasons. First, a perfect loop is replaced by an imperfectly nested loop, which scans a disjoint union of polyhedra. Second, a statement domain may be divided into several disjoint polyhedra, where each polyhedron is scanned by a different loop. In this case, the statement code has to be duplicated in each of these loops. Figure 10.9 illustrates this for the example of Figure 10.7.

The main idea is to recursively decompose the union of polyhedra into imperfectly nested loops, starting from outermost loops to innermost loops. At each step, we seek to solve the following problem: given a context (i.e., a polyhedron in $\mathbb{Z}^{(d-1)}$ containing the outer loops and system parameters), and a union of polyhedra in $\mathbb{Z}^n, n \geq d$, generate a loop that scans this union in lexicographic order.

We generate each additional level of loops by:

1. Projecting the polyhedra onto the outermost d dimensions
2. Separating these projections into disjoint polyhedra
3. Recursively generating loop nests that scan each of these
4. Sorting these loops so that their textual order respects the lexicographic order

(a) Projection on i and separation into disjoint polyhedra



(b) Sorted i loops

```
{i | 1<=i<=2} ::
  {i,j | 1<=i<=2; 1<=j<=M} ::
    S1;
{i | 3<=i<=N} ::
  {i,j | 3<=i<=N; 1<=j<=M} ::
    S1;
  {i,j | 3<=i<=N;  i=j} ::
    S2;
```

FIGURE 10.10 Projections of domains on first dimension and separation into disjoint loops.

Two subtle details arise. First, the idea of "sorting" polyhedra (in step 4 above), necessitates an order relation and an associated algorithm. Second, we need to ensure that the "separation" of polyhedra (in step 2 above) must be such that the resulting polyhedra can be so sorted.

We illustrate the algorithm with an example. Let us start with the program of Figure 10.7, and generate the first level of loops. Each domain is projected onto the first dimension and the parameters (i.e., {**i, N, M**}). These projections are then separated into three disjoint regions: one region containing only **S1**, one region containing both **S1** and **S2**, and one region containing only **S2**. For our example, the last region is empty, as shown in Figure 10.10(a). Furthermore, note that some regions (those that involve the difference of polyhedra) could be unions of polyhedra.

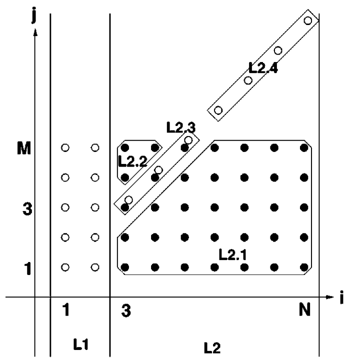
The two remaining regions represent two loops scanning different pieces of dimension **i**, parameterized by **N** and **M**. At this point, only the first level polyhedra ({**i** | 1 <= **i** <= 2} and {**i** | 3 <= **i** <= **N**}) can be interpreted as loops. The other (two-dimensional) polyhedra act as guards on the statements. Note that in Figure 10.10, these guards are partially redundant with their context. The redundant constraints are eliminated later.

We recursively generate separate pieces of code to scan, respectively, regions 1 (containing **S1** alone) and 2 (containing both **S1** and **S2**). Then we textually place the former before the latter. This order respects the lexicographic schedule. Finding such a textual order is trivial for the first dimension, but becomes more complicated for subsequent levels. In the rest of this example, whenever such an order is needed, we give a valid order without discussing how to obtain it.

Now, we generate the next level of loops in the context of the first-level loops. The first **i**-loop — labeled **L1** in Figure 10.11(a) — contains only one statement, and thus the generation of its inner loop is a perfect loop generation problem. Next, consider the second *i*-loop, **L2**. It contains two guarded statements. In the next level of transformation, these domains are first separated into four disjoint polyhedra: two of them (namely, **L2.1** and **L2.2**) containing **S1** alone, one (**L2.3**) containing both **S1** and **S2**, and one (**L2.1**) containing **S2** alone. Then, we sort these polyhedra, such that the following constraints are respected:

- Loop (**L2.1**) precedes loops (**L2.2**), (**L2.3**) and (**L2.4**).
- Loop (**L2.3**) precedes loop (**L2.2**).

(a) Disjoint polyhedra



(b) Sorted loops

```
L1     {i | 1<=i<=2} ::
L1.1     {i,j | 1<=j<=M} ::
           S1;
L2     {i | 3<=i<=N} ::
L2.1     {i,j | 1<=j<=i-1; j<=M} ::
           S1;
L2.3     {i,j | i=j; j<=M} ::
           S1;
           S2;
L2.4     {i,j | i=j; M+1<=j} ::
           S2;
L2.2     {i,j | i+1<=j<=M} ::
           S1;
```

FIGURE 10.11 Second level: projection and separation of the second loop nest (**L2**).

Multiple valid textual orderings of these four loops exist, and we propose one of them in Figure 10.11(b). Also note that no constraint exists on the relative position of loops (**L2.2**) and (**L2.4**), because their respective contexts ($\{\mathbf{i} \mid \mathbf{i} <= \mathbf{M} - \mathbf{1}\}$ and $\{\mathbf{i} \mid 1 >= \mathbf{M} + 1\}$) are distinct. In other words, for any value of the parameter **M** and the outer loop index **i**, at least one of these loops is empty; it follows that any textual order of these loops can execute correctly.

Finally, we generate code by pretty printing the sorted nested loops, yielding the code we saw previously in Figure 10.9.

10.10 Limitations of the Polyhedral Model

The polyhedral model suffer from two important limitations. The first is inherent in the model, namely, the restrictive class of programs covered. This is indeed a fundamental limitation, although one may perform an approximate analysis by investigating cases where the dependences in the program as well as the domains of iterations are approximated by affine functions and polyhedra, respectively. Nevertheless, the model precludes computations that have a more dynamic behavior, in the sense that the control flow is conditioned by results computed during the program. In spite of this limitation, the number of programs that come within its scope is considerable. One might view the limitation as the price to pay for the powerful analysis techniques that the model offers.

The second, and somewhat more serious limitation is that neither the analyses nor the program transformations that the model offers can satisfactorily deal with resource constraints. We have already had an inkling of this when we considered the processor allocation function in Section 10.7, where we were able to formalize and reason about the potential conflicts and the communication behavior only for a set of virtual processors. One might question whether choosing an optimal mapping to these virtual processors can remain optimal after the virtual-to-physical mapping that is expected to follow. Moreover, because this virtual-to-physical mapping assigns the computations of many virtual processors to a single physical processor, it is also necessary to modify the schedule. Hence, our choice of the so-called *optimal* schedules that we made in Section 10.6 is also open to question — the machine model there assumed an unbounded number of processors.

The most well-known method for dealing with resource constraints is through a technique called *tiling* or *blocking* or *supernode partitioning*. Essentially, the idea is to cluster a certain number of nodes in the EDG as a single node and try to develop compact analysis methods. The common method of specifying tiles is as groups of computation nodes delineated by a family of hyperplanes (i.e., integer points within a hyperparallelepiped-shaped subset of the iteration space). Unfortunately, such transformations do not satisfy the closure properties of the polyhedral model. This is especially true if we wish to retain both the size and the number of tiles in each dimension as a parameter for the analysis and hence develop compact methods for analysis and code generation.

10.11 Bibliographic Notes

A number of references are available on the more conventional aspects of dependence analysis and restructuring compilers, notably the texts by Allen and Kennedy [3], Banerjee [6, 7] and Zima and Chapman [49].

The roots of the polyhedral model can be traced to the seminal work of Karp, Miller and Winograd [22], who defined SUREs, and for SUREs where all variables are defined over the entire positive orthant, resolved the scheduling problem. They developed the multidimensional scheduling[9] algorithm explained in Section 10.6.

Another field that contributed heavily to the polyhedral model was the work on automatic synthesis of systolic arrays, which initially built on the simpler ideas (one-dimensional affine schedule for a single SURE, and shifted-linear schedules for SAREs) of Karp et al. Though many authors in the early 1980s worked on the systolic synthesis by space time transformations, the first use of recurrence equations was in 1983 by Quinton [33], who studied a single URE[10] defined over a polyhedral index space. He expressed the space of one-dimensional affine schedules as the feasible space of an integer linear programming problem, thus establishing the link to the work of Karp et al. Rao [39, 40] and Roychowdhury [42] investigated multidimensional schedules for SUREs, and Rao also improved the alorithm given by Karp et al.

Delosme and Ipsen first studied the scheduling problems for affine recurrences (they defined the term *ARE*), but without considering the domains over which they were defined [14]. They showed that all (one-dimensional) affine schedules belong to a cone. Rajopadhye et al. addressed the problem of scheduling a single ARE defined over a polyhedral index domain and showed that the space of valid one-dimensional affine schedules is described in terms of linear inequalities involving the extremal points (vertices and rays) of the domain [38]. Quinton and Van Dongen also obtained a somewhat tighter result [34]. Yaacoby and Cappello [48] investigated a special class of AREs. Mauras et al. extended this result to SAREs, but with one-dimensional variable dependent schedules [30]. Rajopadhye et al. further extended this and proposed piecewise affine schedules for SAREs [37]. Feautrier [18] gave an alternative formulation using the Farkas lemma, for determining (one-dimensional, variable dependent) affine schedules for an SARE. He further extended the method to multidimensional schedules [19]. Some similar ideas were also developed in the loop parallelization community, notably the work of Allen and Kennedy [1, 2], Lamport [24] and Wolf and Lam [45]. An excellent recent book by Darte, Robert and Vivien [12] provides a detailed description of scheduling SUREs, and also addresses the problem of SAREs by formalizing affine and more general dependences as dependence cones engendered by a finite set of uniform dependence vectors.

Feautrier [16] paved the way to using these results for loop parallelization by developing an algorithm for exact dataflow analysis of ACLs (which he called *static control loops*, somewhat of a misnomer) using parametric integer programming, and showed that this yields an SARE whose

[9]Today we know them to be multidimensional schedules, but this understanding was slow in coming.

[10]Actually Quinton worked on a a particular restricted form of SUREs, but this can always be viewed as a single URE for analysis purposes.

variables are defined over (a generalization of) polyhedral domains. The ideas in Section 10.4 are based on his work, though presented differently.

In terms of the decidability of the scheduling problem, Gachet and Joinnault [21] showed the undecidability of SURE scheduling when the variables are defined over arbitrary domains. Saouter and Quinton [43] showed that scheduling even parameterized families of SAREs (each of whose domains is bounded) is also undecidable.

The processor allocation function as defined in Section 10.7 is directly drawn from the systolic synthesis literature. Modeling the communication as the image of the dependences by the allocation function also draws from work on the localization of affine dependences, notably by Choffrut and Culik [10], Fortes and Moldovan [20], Li and Chen [27, 28], Quinton and Van Dongen [34], Rajopadhye et al. [35, 36, 38], Roychowdhury et al. [41, 42], Wong and Delosme [46], and Yaacoby and Cappello [47].

The problem of memory allocation in the polyhedral model has been addressed by Chamski [8], De Greef, Catthoor and De Man [13], Lefebvre and Feautrier [26], Quilleré and Rajopadhye [31] (from which the results presented in Section 10.8 are drawn) and Rajopadhye and Wilde [44].

The development of a programming language and transformation system based on the polyhedral model also draws considerably from the area of systolic synthesis. Some early results on the closure of a single ARE were developed by Rajopadhye, Purushothaman and Fujimoto [38]. Chen defined the language Crystal based on recurrences defined over data fields [9], and Choo and Chen developed a theory of domain transformations for Crystal. ALPHA was defined by Mauras [29] and developed an initial transformation framework for ALPHA programs.

The key problem of code generation, namely, polyhedron scanning was posed and resolved by Ancourt and Irigoin [5] using the Fourier–Motzkin elimination. It is used in many parallelizing tools such as the SUIF system [4] developed at Stanford University, PIPS developed at the École des Mines de Paris and LOOPO developed at the University of Passau. The efficient formulation using the dual representation was developed by LeVerge et al. [25]. Code generation from a union of polyhedra was addressed by Kelly, Pugh and Rosser [23] for the Omega project at the University of Maryland, and by Quilleré, Rajopadhye and Wilde [32] (the results presented in Section 10.9 are drawn from the latter).

Additional information about the polyhedral model may be obtained from a survey article by Feautrier [17], and many of the mathematical foundations are described by Darte [11].

References

[1] R. Allen and K. Kennedy, PFC: A Program To Convert FORTRAN to Parallel Form, in IBM Conference on Parallel Processing and Scientific Computing, 1982.

[2] R. Allen and K. Kennedy, Automatic translation of FORTRAN programs to parallel form, *ACM Trans. Programming Languages Syst.*, 9(4), 491–542, October 1987.

[3] R. Allen and K. Kennedy, *Optimizing Compilers for Modern Architectures*, Morgan Kaufmann, San Francisco, 2002.

[4] S.P. Amarasinghe, Parallelizing Compiler Techniques Based on Linear Inequalities, Ph.D. thesis, Computer Science Department, Stanford University, Stanford, CA, 1997.

[5] C. Ancourt and F. Irigoin, Scanning polyhedra with DO loops, in *Third Symposium on Principles and Practice of Parallel Programming (PPoPP)*, ACM SIGPLAN, ACM Press, New York, 1991, pp. 39–50.

[6] U. Banerjee, *Dependence Analysis for Supercomputing*, Kluwer Academic, Dordrecht, 1988.

[7] U. Banerjee, *Loop Transformations for Restructuring Compilers*, Kluwer Academic, Dordrecht, 1993.

[8] Z. Chamski, Generating Memory Efficient Imperative Data Structures from Systolic Programs, Technical report PI-621, IRISA, Rennes, France, December 1991.

[9] M.C. Chen, A parallel language and its compilation to multiprocessor machines for VLSI, in *Principles of Programming Languages*, ACM Press, New York, 1986.

[10] C. Choffrut and K. Culik, II, Folding of the plane and the design of systolic arrays, *Inf. Process. Lett.*, 17, 149–153, 1983.

[11] A. Darte, Mathematical tools for loop transformations: from systems of uniform recurrence equations to the polytope model, in *Algorithms for Parallel Processing*, Vol. 105, IMA Volumes in Mathematics and Its Applications, M.H. Heath, A. Ranade and R. Schreiber, Eds., 1999, pp. 147–183.

[12] A. Darte, Y. Robert, and F. Vivien, *Scheduling and Automatic Parallelization*, Birkhäuser, Basel, 2000.

[13] E. De Greef, F. Catthoor, and H. De Man, Memory Size Reduction through Storage Order Optimization for Embedded Parallel Multimedia Applications, in Parallel Processing and Multimedia, Geneva, Switzerland, July 1997.

[14] J.-M. Delosme and I.C.F. Ipsen, Systolic array synthesis: computability and time cones, in M. Cosnard, P. Quinton, Y. Robert and M. Tchuente, Eds., *International Workshop on Parallel Algorithms and Architectures*, Elsevier Science/North Holland, Amsterdam, April 1986, pp. 295–312.

[15] P. Feautrier, Parametric integer programming, *RAIRO Rech. Opérationelle*, 22(3), 243–268, September 1988.

[16] P. Feautrier, Dataflow analysis of array and scalar references, *Int. J. Parallel Programming*, 20(1), 23–53, February 1991.

[17] P. Feautrier, Automatic parallelization in the polytope model, in *The Data Parallel Programming Model: CNRS Spring School*, G.-R. Perin and A. Darte, Eds., Springer-Verlag, New York, 1996, pp. 79–103.

[18] P. Feautrier, Some Efficient Solutions to the Affine Scheduling Problem, Part I, One-Dimensional Time, Technical report 28, Labaratoire MASI, Institut Blaise Pascal, April 1992.

[19] P. Feautrier, Some Efficient Solutions to the Affine Scheduling Problem, Part II, Multidimensional Time, Technical report 78, Labaratoire MASI, Institut Blaise Pascal, October 1992.

[20] J.A.B. Fortes and D. Moldovan, Data Broadcasting in Linearly Scheduled Array Processors, in Proceedings, 11th Annual Symposium on Computer Architecture, 1984, pp. 224–231.

[21] P. Gachet and B. Joinnault, Conception d'Algorithmes et d'Architectures Systoliques, Ph.D. thesis, Université de Rennes I, September 1987 (in French).

[22] R.M. Karp, R.E. Miller and S.V. Winograd, The organization of computations for uniform recurrence equations, *JACM*, 14(3), 563–590, July 1967.

[23] W. Kelly, W. Pugh and E. Rosser, Code Generation for Multiple Mappings, in Frontiers 95: 5th Symposium on the Frontiers of Massively Parallel Computation, McLean, VA, 1995.

[24] L. Lamport, The parallel execution of DO loops, *Commun. ACM*, 83–93, February 1974.

[25] H. Le Verge, V. Van Dongen and D. Wilde, Loop nest synthesis using the polyhedral library, Technical report PI 830, IRISA, Rennes, France, May 1994; also published as INRIA Research report 2288.

[26] V. Lefebvre and P. Feautrier, Optimizing storage size for static control programs in automatic parallelizers, in *Euro-Par '97*, Lengauer, Griebl, and Gorlatch, Eds., Vol. 1300, Springer-Verlag, 1997.

[27] J. Li and M. Chen, The data alignment phase in compiling programs for distributed memory machines, *J. Parallel Distributed Comput.*, 13, 213–221, 1991.

[28] J. Li and M.C. Chen, Compiling communication efficient programs for massively parallel machines, *IEEE Trans. Parallel Distributed Syst.*, 2(3), 361–376, July 1991.

[29] C. Mauras, ALPHA: un Langage Équationnel pour la Conception et la Programmation d'Architectures Parallèles Synchrones, Ph.D. thesis, L'Université de Rennes I, IRISA, Campus de Beaulieu, Rennes, France, December 1989.

[30] C. Mauras, P. Quinton, S.V. Rajopadhye, and Y. Saouter, Scheduling affine parameterized recurrences by means of variable dependent timing functions, in *International Conference on Application Specific Array Processing*, S.Y. Kung and E. Swartzlander, Eds., IEEE Computer Society, Princeton, NJ, September 1990, pp. 100–110.

[31] F. Quilleré and S. Rajopadhye, Optimizing memory usage in the polyhedral model, *ACM Trans. Programming Languages Syst.*, 22(5), 773–815, September 2000.

[32] F. Quilleré, S. Rajopadhye and D. Wilde, Generation of efficient nested loops from polyhedra, *Int. J. Parallel Programming*, 28(5), 469–498, October 2000.

[33] P. Quinton, The systematic design of systolic arrays, in *Automata Networks in Computer Science*, F. Fogelman Soulie, Y. Robert and M. Tchuente, Eds., Princeton University Press, Princeton, NJ, 1987, pp. 229–260; preliminary versions appear as IRISA Technical reports 193 and 216, 1983, and in the proceedings of the IEEE Symposium on Computer Architecture, 1984.

[34] P. Quinton and V. Van Dongen, The mapping of linear recurrence equations on regular arrays, *J. VLSI Signal Process.*, 1(2), 95–113, 1989.

[35] S.V. Rajopadhye, Synthesizing systolic arrays with control signals from recurrence equations, *Distributed Comput.*, 3, 88–105, May 1989.

[36] S.V. Rajopadhye, LACS: A Language for Affine Communication Structures, Technical report 712, IRISA, 35042, Rennes Cedex, April 1993.

[37] S.V. Rajopadhye, L. Mui and S. Kiaei, Piecewise linear schedules for recurrence equations, in *VLSI Signal Processing, V*, K. Yao, R. Jain, W. Przytula and J. Rabbaey, Eds., IEEE Signal Processing Society, 1992, pp. 375–384.

[38] S.V. Rajopadhye, S. Purushothaman and R.M. Fujimoto, On synthesizing systolic arrays from recurrence equations with linear dependencies, in *Proceedings, 6th Conference on Foundations of Software Technology and Theoretical Computer Science, Lecture Notes in Computer Science*, Springer-Verlag, New York, 1986, pp. 488–503; later appeared in *Parallel Comput.*, June 1990.

[39] S. Rao, Regular Iterative Algorithms and Their Implementations on Processor Arrays, Ph.D. thesis, Information Systems Laboratory, Stanford University, Stanford, CA, October 1985.

[40] S. Rao and T. Kailath, What is a systolic algorithm?, in Proceedings, Highly Parallel Signal Processing Architectures, SPIE, Los Angeles, CA, January 1986, pp. 34–48.

[41] V. Roychowdhury, L. Thiele, S.K. Rao and T. Kailath, On the localization of algorithms for VLSI processor arrays, in *VLSI Signal Processing, III*, R.W. Brodersen and H.S. Moscovitz, Eds., IEEE Acoustics, Speech and Signal Processing Society, IEEE Press, 1988, pp. 459–470; a detailed version is submitted to *IEEE Trans. Comput.*

[42] V.P. Roychowdhury, Derivation, Extensions and Parallel Implementation of Regular Iterative Algorithms, Ph.D. thesis, Department of Electrical Engineering, Stanford University, Stanford, CA, December 1988.

[43] Y. Saouter and P. Quinton, Computability of recurrence equations, *Theor. Comput. Sci.*, 114, 1993.

[44] D. Wilde and S. Rajopadhye, Memory reuse analysis in the polyhedral model, *Parallel Process. Lett.*, 7(2), 203–215, June 1997.

[45] M. Wolf and M. Lam, Loop transformation theory and an algorithm to maximize parallelism, *IEEE Trans. Parallel Distributed Syst.*, 2(4), 452–471, October 1991.

[46] F.C. Wong and J.-M. Delosme, Broadcast Removal in Systolic Algorithms, in International Conference on Systolic Arrays, San Diego, CA, May 1988, pp. 403–412.

[47] Y. Yaacoby and P.R. Cappello, Converting affine recurrence equations to quasi-uniform recurrence equations, in *AWOC 1988: 3rd International Workshop on Parallel Computation and VLSI Theory*, Springer-Verlag, New York, June 1988; see also, UCSB Technical report TRCS87-18, February 1988.

[48] Y. Yaacoby and P.R. Cappello, Scheduling a system of nonsingular affine recurrence equations onto a processor array, *J. VLSI Signal Process.*, 1(2), 115–125, 1989.

[49] H. Zima and B. Chapman, *Supercompilers for Parallel and Vector Computers*, Frontier Series, ACM Press, New York, 1990.

11

Compilation for Distributed Memory Architectures

Alok Choudhary
Northwestern University

Mahmut Kandemir
Pennsylvania State University

11.1 Introduction

Distributed memory machines provide the required computational power to solve large-scale, data-intensive applications. These machines achieve high performance and scalability; however, they are very difficult to program. This is because taking advantage of parallel processors and distributed memory (see Figure 11.1) requires that both data and computation should be distributed between processors. In addition, because each processor can directly access only its local memory, nonlocal (remote) accesses demand a coordination (in the form of explicit communication or synchronization) across processors. Because the cost of interprocessor synchronization and communication might be very high, a well-written parallel code for distributed memory machines should minimize the number of synchronization and communication operations as much as possible. These issues make it very difficult to program these architectures and necessitates optimizing compiler help for generating efficient parallel code. Nevertheless, most of the current compiler techniques for distributed memory architectures require some form of user help for successful compilation.

This chapter presents an overview of existing techniques for distributed memory compilation and points to promising future directions. Although there are many ways of generating code for a distributed memory architecture (depending on the type of underlying abstraction, the communication primitives and the programming interface), most of this chapter focuses on message-passing machines with two-way communication primitives and on data parallelism. Issues investigated are automatic

0-8493-1240-7/03/$0.00+$1.50

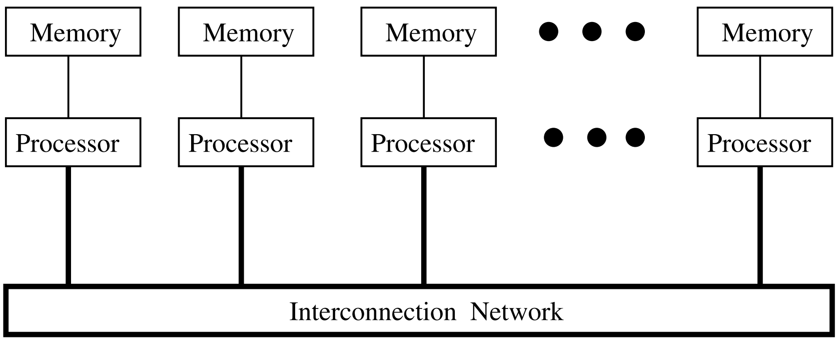


FIGURE 11.1 Distributed memory architecture.

data decomposition, computation partitioning, communication detection and optimization, locality related problems, and handling input and output (I/O) and irregular computations.

At the highest level, the two ways of generating code for distributed memory architectures are explicit (programmer-directed) parallelization and automatic parallelization. Most of the current languages for distributed memory architectures, such as Vienna FORTRAN [21], FORTRAN-D [51] and high-performance FORTRAN (HPF) [61], provide data alignment and distribution directives to the users. By using these directives, the users can specify the data mappings; that is, they give suggestions to the compiler as to how data (e.g., arrays) should be decomposed across parallel processors. Then, by using this information, an optimizing compiler can derive computation partitions and optimize communication and synchronization. In the automatic parallelization option, on the other hand, the compiler itself determines how data should be decomposed across processors [36, 63]. After data decomposition, the compiler proceeds with computation partitioning and communication optimization as in the first option.

Whether fully automatic or user assisted, the compilation strategies for distributed memory architectures start with a parallelism detection step. Exploiting large (coarse) granular parallelism in loop nest based codes enables processors to perform data communication more efficiently and improves overall execution time. To ensure coarse granular parallelism, the compiler typically employs a suite of code transformations. For example, the compiler can interchange two loops if doing so places data dependences into inner loop positions so that outer loops can safely be parallelized. Other transformations such as loop fission and iteration space tiling can also be used for maximizing parallelism. Because maximizing parallelism through loop-level transformations is itself a complex matter, in this chapter we assume that all parallelism-related optimizations have already been performed. We refer the reader to [83] for loop-based techniques to enhance parallelism and improve data locality in array-intensive codes.

The remainder of this chapter is organized as follows. Section 11.2 discusses the problem of automatic data distribution (layout detection) for distributed memory architectures. Section 11.3 presents computation partitioning techniques and discusses the differences between fully automatic and user-assisted approaches. Section 11.4 examines how interprocessor communication can be detected and optimized. In this section, we give details of several loop (loop-level) and global communication optimization strategies. Section 11.5 covers one-way communication. Section 11.6 addresses the locality problem from processor as well as memory hierarchy perspectives. Section 11.7 discusses the problem of compiling applications with irregular data accesses for message-passing architectures. Section 11.8 describes I/O compilation. Section 11.9 points to future trends in distributed memory compilation and Section 11.10 concludes the chapter with a summary.

11.2 Automatic Data Decomposition

Data distribution is one of the key aspects that a parallelizing compiler for a distributed memory architecture should consider to get efficiency from the system. The cost of accessing local and remote data can be one or several orders of magnitude different, and this can dramatically affect performance. Several groups studied techniques for automatic data decomposition. We can classify these techniques as integer linear programming (ILP)-based techniques and heuristic approaches. The objective of the techniques in the first group is to formulate the data decomposition problem formally using ILP and to come up with data decompositions (and accompanying computation partitionings) such that the parallelism is maximized and interprocessor data communication is minimized.

In [63], an ILP-based technique is presented. The technique starts by partitioning the program into code segments called phases. Each phase roughly corresponds to an innermost loop. In the second step, for each phase, a search space of candidate layouts (array decompositions) is constructed. The technique generates the promising candidate layouts for a phase based on their expected performance as part of an efficient data layout for the entire program. The phase structure of the program is represented in a phase control flow graph (PCFG), where each phase is represented by a single node, and edges (which represent the control flow between phases) are annotated using branch probabilities and loop control information. After that, a performance estimation step is applied and performance numbers (estimated execution times) are determined for all candidate data layouts and possible (layout) remappings between layouts (as we move from one phase to another). Then, armed with this information (the potential layouts for each array in each phase, cost of each layout, and cost of each potential remapping), the problem is formulated as a linear 0-1 integer problem and solved optimally. Reference [63] also presents different 0-1 ILP formulations for the same problem. It should be noted that the optimality of this approach is with respect to search spaces under consideration. That is, an inaccurate selection of search spaces may lead to a poor solution, let alone optimality.

An approach in a similar direction is taken by Garcia, Ayguade and Labarta [32, 33]. The difference is that instead of solving the layout problem in two subproblems, namely, alignment and decomposition, their strategy solves both the subproblems simultaneously.

Gupta and Banerjee [36] implemented the PARADIGM compiler on top of the Parafrase-2 system. They developed a methodology (heuristic) for automatic data partitioning given a sequential or shared memory parallel program, generating single program multiple data (SPMD) programs with explicit communication. More specifically, their approach divides the automatic data decomposition process in four phases. The first phase determines whether a blockwise data decomposition or a cyclic data distribution should be used for each dimension of each array. In the second phase, for each cyclicly distributed dimension, a cyclic factor is decided. A cyclic factor is the block size for a dimension distributed in a cyclic manner. The next phase determines the number of processors assigned in each of the processor-mesh dimensions, assuming that the maximal number of distributed dimensions is two. The overall computational cost is estimated from the computational cost of a single instance of each statement and the count of the number of times that statement is executed. Both communication and synchronization costs are also taken into account. Finally, they employ an alignment phase in which the array dimensions that are used the same way are aligned together. For each aligned array dimension, their approach decides whether the decomposition style has to be block or cyclic. This is performed by estimating the penalty incurred in execution time if the decomposition is selected as blockwise or cyclic.

A number of articles (e.g., [73]) also consider the use of more costly techniques such as genetic algorithms for solving the automatic data decomposition problem. Dierstein et al. [25] describe the ADDAP system, a parallelizing compiler for distributed memory machines. The compiler computes a data distribution for the arrays of the source program automatically by a branch-and-bound algorithm and parallelizes the inner loops of the program by inserting the necessary communication statements to access nonlocal array sections. Redistributions of data during program execution is arranged, if this is expected to result in a better program performance. The data distribution phase uses a communication analysis tool that computes the communication costs of the different data distributions by determining the number and size of the messages that each processor has to receive during program execution. The communication analysis tool also takes sequentializations into account that are caused by data dependences.

In addition to compiler-based techniques, a number of off-line tools also suggest suitable data decompositions to the compiler or user. Automatic data distribution and placement tool (ADAPT) [28] is a device that is designed to simplify and accelerate the task of developing HPF versions of existing constant folding (CFD) applications. The tool starts with an analysis of array index expressions of the loop nests. For each pair of arrays referenced in a nest statement it generates an arc in the alignment graph annotated with an affinity relation. The template, alignment and distribution directives for a particular loop nest then are derived from a transitive closure of the affinity relation. A compromise of data decompositions in different nests and subroutines is performed by merging annotated alignment graphs for adjacent nests or subroutine calls in the nest or call graph of the application. ADAPT has been implemented as a C++ program running in conjunction with a parallelization tool called CAPTools [52]. It takes advantage of the parse tree, interprocedural analysis and application database generated by CAPTools. It also relies on the use of directed graph class initially implemented in parallel debugger of distributed programs (p2d2). The tool has been tested with benchmark codes, such as ARC3D, BT and LU, and has been found to generate effective data decompositions.

Some studies (e.g., [10]) also considered automatic dynamic data redistribution. Using data redistribution (at runtime) is beneficial when a poor initial decomposition of the data is responsible for limited performance or different sections of the code demand different data decompositions for the same array. It should be noted, though, that the program should run for at least a few seconds because the communication system typically needs some time to move the data to the best layout and they need to spend some time there to amortize the cost of dynamic redistribution.

11.3 Computation Distribution

When a compilation strategy based on automatic data layout detection is employed, the computation distribution is performed along with data decomposition. One of the methods to achieve this is to employ a linear algebraic strategy in which loop nests, arrays and array accesses are all represented within a matrix framework.

Anderson and Lam [8] present an optimization strategy that determines data decompositions and computation distributions automatically and optimizes parallelism and locality simultaneously. In their mathematical model, data and computation decompositions are represented as affine transformations. A loop iteration is represented by $\vec{I} = (i_1, i_2, \ldots, i_n)$ and each array element is represented using $\vec{a} = (a_1, a_2, \ldots, a_m)$. An affine array index function is written as $\vec{f}(\vec{I}) = F\vec{I} + \vec{o}$, where F is an $m \times n$ matrix (called access or reference matrix) and $\vec{o}$ is an m-dimensional vector. Then, we can express a data decomposition as $D\vec{a} + \vec{\alpha}$ and a computation distribution as $C\vec{I} + \vec{\beta}$. In this formulation, D and C are $p \times m$ and $p \times n$ linear transformation matrices, respectively, where p is the dimension of processor topology. Having these definitions, Anderson and Lam state the problem as follows: find a computation distribution (C and $\vec{\beta}$) for each nest and a data decomposition (D and $\vec{\alpha}$) for each array in each loop nest such that parallelism is maximized and communication is minimized. In mathematical terms, by assuming that C_i and $\vec{\beta}_i$ specify computation distribution for nest i, D_j and $\vec{\alpha}_j$ specify data decomposition for array j; and $\vec{f_{ji}}(.)$ denotes the data reference to array j in nest i, to eliminate communication:

$$D_j(\vec{f_{ji}}(\vec{I})) + \vec{\alpha}_j = C_i(\vec{I}) + \vec{\beta}$$

should be satisfied for all loop iterations $\vec{I}$, nests i, and arrays j. Obviously, in many array-intensive codes, it may not be possible to satisfy this expression (constraint) for all arrays and loops. The approach proposed in [8] adopts an iterative strategy using as many constraints as possible.

It is also possible to adopt more sophisticated strategies. Amarasinghe and Lam [6] present a compilation strategy that is based on data flow analysis on individual instances of array accesses. By using a representation called last-write-tree (LWT) the authors capture perfect data flow information, perform data and computation decompositions and detect and optimize interprocessor communication. Their code generation strategy is based on scanning a polyhedron.

With another group of techniques programmers try to determine computation distribution based on user-defined data decompositions. The languages such as HPF allow programmers to describe how data are to be decomposed among the processors in a distributed memory machine. The programmer can describe the data-to-processor mapping in two stages: the DISTRIBUTE and ALIGN operations. DISTRIBUTE is an HPF directive that indicates how an array is to be decomposed into portions. For example, given an array declaration such as INTEGER X(200, 200), the directive !HPF$ DISTRIBUTE X(BLOCK,*) decomposes array X in such a fashion that each processor gets a number of consecutive rows of the array. It is also possible to distribute rows of the array in a cyclic manner using, for example, the directive !HPF$ DISTRIBUTE X(CYCLIC,*). The ALIGN directive, on the other hand, indicates how two (or more) arrays are aligned with respect to each other. As an example, !HPF$ ALIGN X(I,J) with Y(I,J) implies that the corresponding elements of arrays X and Y are to be decomposed in the same fashion. Typically, the arrays are first aligned with respect to each other (to capture locality) and then distributed across processors.

The data decomposition expressed using ALIGN and DISTRIBUTE directives is then used to guide the compiler to generate an SPMD style of execution. In this paradigm, each processor executes the same program, but operates on different data. To achieve this, a node program is generated and loaded into each processor. This program can typically update the local data only (owner-computes rule). Obviously, to create such a parallel code, the compiler has to translate global array references into local and nonlocal references. All nonlocal references should then be accessed using explicit communication. The early FORTRAN D compiler also partitioned computation across processors using the owner-computes rule, where each processor only computes values of data it owns.

In the owner-computes rule, the processor that owns the left-hand side (lhs) element performs the calculation. For example, in:

```
DO i = 1,n
 X(i-1) = Y(i*6)/Z(i+j)-X(i**i)
END DO
```

the processor that owns X(i−1) performs the assignment. The components of the right-hand side (rhs) expression may have to be communicated to this processor before the assignment is made. It is easy to see that for the owner-computes rule to be successful, the data decomposition across processors should be performed with care. It should be noted, however, that as this is a rule of thumb, it is not always followed; for example, if all the rhs objects are codistributed, then instead of all the rhs elements being sent to the owner of the lhs for computation, the computation of the result may take place on the home processor of the rhs elements and next be sent to the owner of the lhs for assignment. This would reduce the number of communications required. In fact, it is up to the compiler to decide when to follow the owner-computes rule and when not to. The next step in compilation is to minimize the communication due to rhs references as much as possible. The next section discusses this problem in detail.

Automatic data parallelism translator (Adaptor) [15] is a compilation system that transforms data parallel programs written in FORTRAN with array extensions, parallel loops and layout directives to parallel programs with explicit message passing. Adaptor only takes advantage of the parallelism in the array operations and of the parallel loops. It has no features for automatic parallelization.

11.4 Communication Detection and Optimization

On distributed memory machines, the time (cost) to access nonlocal (remote) data is usually orders of magnitude higher than accessing local data. For example, on the large-scale parallel machines, accessing remote data can consume thousands of cycles, depending on the distance between communicating processors [47]. Therefore, it is imperative that the frequency and volume of remote accesses are reduced as much as possible. In particular, in message-passing programs, the start-up cost for the messages can easily dominate the execution time. For example, on the IBM SP-2, the message start-up time is approximately 360 times the transfer time per word, indicating that optimizing communication is very important. The problem is much more severe on network of workstations because interprocessor latencies are much higher. Several software efforts have been aimed at reducing the communication overhead. The main goal of these optimizations is to increase the performance of programs by combining messages in various ways to reduce the overall communication overhead. These techniques can be classified into two categories depending on the scope of optimization: optimizations that target a single nest at a time (which we call local or loop-level optimizations in this chapter) and optimizations that target multiple nests simultaneously.

11.4.1 Local Optimizations

The most common optimization technique used by previous researchers is message vectorization [9, 10, 14, 31, 51]. In message vectorization, instead of naively inserting Send and Recv operations just before references to nonlocal data, communication is hoisted to outer loops. Essentially this optimization replaces many small messages with one large message, thereby reducing the number of messages. For example, consider the program fragment shown in Figure 11.2(a) and assume that all arrays are distributed across processors blockwise in the second dimension. Figure 11.2(b) and (c) shows naively inserted messages and message vectorization, respectively, for a processor `p` before loop bounds reduction (a technique to allow processors to execute only those iterations that have assignments that write to local memory [51]) and guard insertion (a technique that guarantees correct execution of statements within loop nests). The notation `send{B,q,n}` means that `n` elements of array `B` should be sent to processor `q`; `recv{B,q,n}` is defined similarly. For this discussion, we are not concerned with exactly which elements are sent and received. Notice that the version in Figure 11.2(c) reduces the message start-up cost as well as the message latency.

Some of the researchers [2, 51] also considered message coalescing, a technique that combines messages due to different references to the same array; and message aggregation that combines messages due to references to different arrays to the same destination processor into a single message. Once nonlocal accesses are vectorized at outer loops, the compiler considers the sets of elements that need to be communicated and combines two communication sets if they contain elements that need to be communicated between the same pair of processors and the elements in both the sets belong to the same array. This is called message coalescing. As discussed in [51], if two overlapping communication sets cannot be coalesced without loss of precision, they can be split into smaller sections in a manner that allows the overlapping sections to be merged precisely.

Message aggregation is employed to ensure that only one communication message (per nest) is sent to each processor. This requires that the communication sets due to different array variables are to be merged into a single set, which is then communicated. FORTRAN D compiler [51] applies message aggregation after message vectorization and coalescing by combining all communication sets representing data sent to the same processor.

```
DO j = 2, 255
 DO i = 1, 255
  A(i,j)=B(i,j)+B(i,j-1)
 END DO
END DO


DO j = 2, 255
 DO i = 2, 256
  C(i,j)=B(i,j-1)+C(i,j)
 END DO
END DO
```

(a)

```
DO j = 2, 255
 DO i = 1, 255
  send {B,p+1,1}, recv {B,p-1,1}
  A(i,j)=B(i,j)+B(i,j-1)
 END DO
END DO

DO j = 2, 255
 DO i = 2, 256
  send {B,p+1,1}, recv {B,p-1,1}
  C(i,j)=B(i,j-1)+C(i,j)
 END DO
END DO
```

(b)

```
send {B,p+1,255}, recv {B,p-1,255}
DO j = 2, 255
 DO i = 1, 255
  A(i,j)=B(i,j)+B(i,j-1)
 END DO
END DO

send {B,p+1,255}, recv {B,p-1,255}
DO j = 2, 255
 DO i = 2, 256
  C(i,j)=B(i,j-1)+C(i,j)
 END DO
END DO
```

(c)

```
send {B,p+1,256}, recv {B,p-1,256}
DO j = 2, 255
 DO i = 1, 255
  A(i,j)=B(i,j)+B(i,j-1)
 END DO
END DO


DO j = 2, 255
 DO i = 2, 256
  C(i,j)=B(i,j-1)+C(i,j)
 END DO
END DO
```

(d)

FIGURE 11.2 (a) A code fragment; (b) naive communication placement; (c) message vectorization; (d) global communication optimization.

If the compiler catches opportunities for exploiting regularity in communication patterns between different processor groups, it can use collective communication. In collective communication, instead of inserting individual Send/Recv primitives for individual communications, the fast collective communication primitives are supported by the underlying architecture. Important examples of communication types that can take advantage of collective communication are the reduction routines and scan operations.

11.4.2 Global Optimizations

The main problem with the local optimizations discussed earlier is that they optimize communication for a single nest at a time. This restriction prevents the compiler from performing interloop optimizations such as global elimination of redundant communication. To see this, consider Figure 11.2(d), which shows the global optimization of the same program fragment via elimination of redundant communication. Notice that, compared with the message-vectorized program in Figure 11.2(c), this version reduces both the number of messages and the communication volume.

A number of authors have proposed techniques based on data flow analysis to optimize communication across multiple loop nests [19, 34, 38, 59, 84, 85]. Most of these approaches use a variant of regular section descriptors (RSD) introduced by Callahan and Kennedy [17]. Two most notable representations are available section descriptor (ASD) [38] and section communication descriptor (SCD) [84, 85]. Associated with each array that is referenced in the program is an RSD that describes the portion of the array referenced. Although this representation is convenient for simple array sections such as those found in pure block or cyclic distributions, it is hard to embed alignment and general distribution information into it. Apart from inadequate support for block–cyclic distributions, working with section descriptors may sometimes result in overestimation of the communication sets, because regular sections are not closed under union and difference operators. The resulting inaccuracy may be linear with the number of data flow formulations to be evaluated, thus defeating the purpose of global communication optimization.

In [53], Kandemir et al. proposed a communication optimization method based on polyhedral techniques. More specifically, they represent the communication between processors using Presburger sets; this allows the compiler to combine and intersect arbitrary sets of communicated elements. In other words, their approach gives the compiler the ability to represent communication sets globally as equalities and inequalities, and to use polyhedron scanning techniques to perform optimizations such as redundant communication elimination and global message coalescing, which were not possible under the loop nest based (i.e., local) communication optimization schemes. This approach first performs a local analysis and determines for each nest the data elements that need to be communicated between processors. Then, an interval-based data flow analysis propagates communication sets between nests and eliminates unnecessary communication. Although this scheme is very general and can cope with a great variety of communication patterns, it can also increase the compilation time; therefore, it needs to be applied with care.

To maximize parallelism and optimize communication, an optimizing compiler can use a suite of loop and data transformations. As mentioned earlier, exploiting a larger granular parallelism improves the overall program behavior. However, procedure calls within nests may in most cases prevent effective loop parallelization. Hall et al. [40, 41] present compiler techniques for optimizing procedure calls within nested loops. One of these techniques is loop embedding which can be very useful when a procedure call occurs within a loop. This transformation moves a loop that contains a call to a procedure to the said procedure and changes the loop header and procedure parameter list accordingly. In doing so, it also needs to (1) eliminate actual parameters that vary in the loop body, (2) ensure that global variables accessed in the loop are within the scope of the called procedure, and (3) create local variables in the called procedure corresponding to locals of the caller that are accessed within the loop but not visible outside the loop [40, 41].

Compilers for distributed memory machines can also employ other, less frequently used optimizations such as pipelining computations, dynamic data decompositions, vector message pipelining, unbuffered messages, owner- or store-based optimizations and iteration reordering. We refer the interested reader to [51] and the references therein.

11.5 One-Way Communication

The compilers for many languages used to program distributed memory machines have traditionally relied on send and receive primitives to implement message-passing communication. The impact of this approach is twofold. First, it combines synchronization with communication in the sense that data messages also carry implicit synchronization information. Although this relieves the compiler of the job of inserting explicit synchronization messages to maintain data integrity and correct execution, separating synchronization messages from data messages may actually improve the performance of programs by giving the compiler the option of optimizing data and synchronization messages separately. In fact, O'Boyle and Bodin [65] and Tseng [78] have presented techniques to optimize synchronization messages on shared memory and distributed shared memory parallel architectures. These techniques, for example, can be applied to the synchronization messages of the programs compiled using separate data and synchronization messages. Second, the compiler has the task of matching send and receive operations to guarantee correct execution. As discussed earlier in this chapter, this is a difficult job and (in practice) limits the number of programs that can be compiled effectively for message-passing architectures.

Alternative communication and synchronization mechanisms, which can be called one-way communication, have been offered. Stricker et al. [74] and Hayashi et al. [43] suggest that the separation of synchronization from data transfer is extremely useful for realizing good performance. In the context of distributed operating systems, a similar separation of data and control transfer has been suggested by Thekkath Levy and Lazowska [77]. Split C [23] offers one-way memory operations and active messages [81] provide a software implementation of one-way communication. One-way communication is also part of the proposed message passing interface standard [64]. The main characteristic of these techniques is that they separate interprocessor data transfers from producer–consumer synchronization. A number of (physically) distributed memory machines such as the Fujitsu AP1000+ [43], the Cray T3D [22], the Cray T3E [72] and the Meiko CS-2 [11] already offer efficient low-level remote memory access (RMA) primitives that provide a processor with the capability of accessing the memory of another processor without direct involvement of the latter. To preserve the original semantics, however, a synchronization protocol should be observed.

In this section, we focus on the compilation of programs augmented with HPF-style data mapping directives using one-way communication operations `Put` (remote memory write) and `Synch` (synchronization). Although one-way communication strategies were originally meant for message-passing machines, they are readily applicable to uniform shared memory architectures as well. Gupta and Schonberg [37] show that the compilers that generate code for one-way communication can exploit shared-memory architectures with flexible cache–coherence protocols (e.g., Stanford FLASH [46] and Wisconsin Typhoon [70]).

The Put primitive — executed by the producer of a data — transfers data from producer memory to consumer memory. This operation is very similar to the execution of a Send primitive by the producer and execution of a matching Recv primitive by the consumer (in a two-way communication strategy). There is an important difference, however; the consumer processor is not involved in the transfer directly and all the communication parameters are supplied by the producer [64]. As stated earlier, to ensure correctness, synchronization operations might be necessary. A large number of synchronization operations can be used to preserve the semantics of the program. These include barriers, point-to-point (or producer–consumer) synchronizations and locks. The synchronization primitive used here, `Synch` (executed by the producer of a data), is a point-to-point communication primitive; however, the discussion here applies to other types of synchronizations as well. Both Stricker et al. [74] and Hayashi et al. [43] use barriers to implement synchronization whereas our effort is aimed at reducing the total amount of synchronization using data flow analysis, and using finer granularity point-to-point primitives where possible.

Clearly, in a compilation framework based on the Put operation, the correct ordering of memory accesses has to be imposed by the compiler using the synchronization primitives. That is, one of the main problems for compiling using Put primitives is to determine the points where explicit synchronization is necessary (to preserve correctness). A straightforward approach inserts a Synch operation just before each Put operation. The next question to be addressed then is whether every Synch operation inserted that way is always necessary. The answer is no, and [55] presents an algorithm to eliminate redundant synchronization messages. We refer to a Synch operation as redundant if its functionality can be fulfilled by other data communications or other Synch operations occurring in the program. The basic idea is to use another message in the reverse direction between the same pair of processors in place of the Synch call.

It should also be mentioned that in machine environments that support both one-way (Put/Get) and two-way (Send/Recv) communication mechanisms, it might be the case that two-way communication mechanisms are implemented using the one-way communication mechanisms. Because of this reason, one-way communication calls might have significantly lower start-up latencies and higher bandwidths. Therefore, when used in a communication-intensive part of a program, they can result in better scaling of that portion of the computation. This observation suggests that in some architectures we might be able to obtain better results using one-way communication. These advantages of one-way communication do not come for free: in cases where the communication activity between processors needs synchronization, explicit synchronization messages should be inserted in the code.

Finally, as discussed in [37, 55], although both Put and Get can be implemented in terms of each other, we prefer to use Put because of these two reasons: first, in general, the handshaking protocol for the Get primitive involves more messages than that of Put [37]; second, the synchronizations originating from Get primitive is due to flow dependences and are in general difficult to eliminate. The synchronization messages in case of the Put primitive, on the other hand, are needed for satisfying antidependences (pseudo dependences), and therefore easier to eliminate.

11.6 Data Locality Issues

On distributed memory machines, cache locality improvements (through data layout transformations) and memory locality (also called processor locality) improvements (through data distribution) are complementary. Although good cache locality optimization combined with a good interprocessor data decomposition strategy can effectively ensure low memory access costs, merely distributing the data across the memories of the processors in a best possible way may not necessarily ensure good cache locality. This is particularly true for array-intensive applications with poor cache behavior. We would like to delve a bit more into this interplay between cache locality optimization and data distribution on distributed memory machines. A detailed study of such interactions along with how a compiler can optimize codes for good overall performance, however, is beyond the scope of this chapter. Although it seems a reasonable idea to first use the best data decomposition strategy and then optimize each node program for the best locality (taking into account the cache locality), it might be possible to obtain better results by considering the interaction between processor locality and cache locality.

Specifically, a compilation strategy can work as follows. First, the compiler applies aggressive loop and data transformations to improve data reuse. After this step, most of intrinsic data reuse in the code is carried by the innermost loops. Note that this means the outer loops are dependence free and can safely be parallelized. It should also be noted that parallelizing these loops can imply a data decomposition strategy. We can expect that in a distributed memory architecture where both processor locality and cache locality are important such an integrated approach could perform better than a straightforward strategy that optimizes each type of locality in isolation. In conclusion, optimizing compilers for distributed memory architectures might attempt to improve both cache and memory locality in a unified framework.

11.7 Compiling Irregular Applications

In addition to standard distributions such as BLOCK and CYCLIC, languages such as FORTRAN D and Vienna FORTRAN also support irregular data distributions. FORTRAN D allows a user to explicitly specify an irregular distribution using an array, to specify a mapping of array elements to processors. Vienna FORTRAN allows user-defined functions to describe irregular distributions. The current version of HPF does not directly support irregular distributions. Language extensions have been proposed by Hanxeleden et al. [45] and Ponnusamy et al. [68] to support irregular distributions in languages such as FORTRAN D.

In irregular problems, data access patterns and workload are usually known only at runtime; hence decisions concerning data and work distributions are made at runtime. Obviously, these on-the-fly decisions require special runtime support. A set of procedures have been developed, called CHAOS

```
C Outer Loop L1
do n = 1, n step
...
C Inner Loop L2
   do i = 1, nedge
     y(edge1(i)) = y(edge1(i)) + f(x(edge1(i)), x(edge2(i)))
     y(edge2(i)) = y(edge2(i)) + g(x(edge1(i)), x(edge2(i)))
   end do
...
end do
```

FIGURE 11.3 An example code with an irregular loop.

[24], that can be used by an HPF-style compiler. CHAOS is a successor of PARTI and provides support for managing user-defined distributions, partitioning loop iterations, remapping data and index arrays and generating optimized communication schedules.

To motivate discussion, we focus on two irregular applications: an unstructured Euler solver and a molecular dynamics code. The nest structures of these two application codes consist of a sequence of loops with indirectly accessed arrays. The unstructured Euler solver application is used to study the flow of air over an airfoil. Complex aerodynamic shapes require high-resolution meshes and consequently large numbers of mesh points. Physical values such as velocity, pressure are associated with each mesh vertex. These values are called *flow variables* and are stored in arrays called *data arrays*. Calculations are carried out using loops over the list of edges that define the connectivity of the vertices (see Figure 11.3). To parallelize the unstructured Euler solver, mesh vertices must be partitioned. Because meshes are typically associated with some sort of physical objects, a spatial location can often be associated with each mesh point. The spatial locations of the mesh points and the connectivity of the vertices are determined by the mesh generation strategy. The way in which the vertices of such irregular computational meshes are numbered frequently does not have a useful correspondence to the connectivity pattern (edges) of the mesh. During mesh generation, vertices are added progressively to refine the mesh. While new vertices are added, new edges are created or older ones are moved around to fulfill certain mesh generation criteria. This causes the apparent lack of correspondence between the vertex numbering and edge numbering. One way to solve this problem is to renumber the mesh completely after the mesh has been generated. Mesh points are partitioned to minimize communication.

Promising heuristics have been developed that can use one or several of the following types of information: (1) spatial locations of mesh vertices, (2) connectivity of the vertices and (3) estimate of the computational load associated with each mesh point. For instance, a user might choose a partitioner that is based on coordinates. A coordinate bisection partitioner decomposes data using the spatial locations of vertices in the mesh. If the user chooses a graph-based partitioner, the connectivity of the mesh could be used to decompose the mesh. The next step in parallelizing this application involves assigning equal amounts of work to processors. A Euler solver consists of a sequence of loops that sweep over a mesh. The computational work associated with each loop must be partitioned among processors to balance the load. Consider a loop that sweeps over mesh edges, closely resembling the loop depicted in Figure 11.3. Mesh edges are partitioned so that (1) load balance is maintained and (2) computations mostly employ locally stored data.

Other unstructured problems have similar indirectly accessed arrays. For instance, consider the nonbonded force calculation in the molecular dynamics code, CHARMM, shown in Figure 11.4. Force components associated with each atom are stored as FORTRAN arrays. The loop L1 sweeps over all atoms. In this discussion, it is assumed that L1 is a parallel loop whereas L2 is a sequential one.

```
L1: do i = 1, NATOM
L2:   do index = 1, INB(i)
         j = Partners(i, index)
         Calculate dF (x, y and z components)
         Subtract dF from Fj
         Add dF to Fi
      end do
    end do
```

FIGURE 11.4 The nonbonded force calculation loop from CHARMM.

The loop iterations of L1 are distributed over processors. All computation pertaining to the iteration i of L1 is carried out on a single processor, so that loop L2 does not need to be parallelized. It is assumed that all atoms within a given cutoff radius interact with each other. The array Partners (i, *) lists all the atoms that interact with atom i. The inner loop calculates the three force components (x, y, z) between atom *i* and atom j (van der Waal's and electrostatic forces). They are then added to the forces associated with atom i and subtracted from the forces associated with the atom j. The force array elements are partitioned in a way as to reduce interprocessor communication in the nonbonded force calculation loop.

The CHAOS runtime library [24] has been developed to efficiently handle problems that consist of a sequence of clearly demarcated concurrent computational phases. Solving such concurrent irregular problems on distributed memory machines using CHAOS runtime support involves six major steps. The first four steps concern mapping data and computations onto processors. The next two steps concern analyzing data access patterns in a loop and generating optimized communication calls. A brief description of these phases is given next. Initially, arrays are decomposed into either regular or irregular distributions.

- *A. Data distribution.* Phase A calculates how data arrays are to be partitioned by making use of partitioners provided by CHAOS or by the user. The CHAOS library supports a number of parallel partitioners that use heuristics based on spatial positions, computational load, connectivity, etc. The partitioners return an irregular assignment of array elements to processors; this is stored as a CHAOS construct called the translation table. A translation table is a globally accessible data structure that lists the home processor and offset address of each data array element. The translation table may be replicated, distributed regularly or stored in a paged fashion, depending on storage requirements.
- *B. Data remapping.* Phase B remaps data arrays from the current distribution to the newly calculated irregular distribution. A CHAOS procedure remap is used to generate an optimized communication schedule for moving data array elements from their original distribution to the new distribution.
- *C. Loop iteration partitioning.* This phase determines how loop iterations should be partitioned across processors. A large number of possible alternative schemes exist for assigning loop iterations to processors based on optimizing load balance and communication volume. The CHAOS library uses the almost-owner-computes rule to assign loop iterations to processors. Each iteration is assigned to the processor that owns a majority of data array elements accessed in that iteration. This heuristic is biased toward reducing communication costs.
- *D. Remapping loop iterations.* This phase is similar to phase B. The indirection array elements are remapped to conform with the loop iteration partitioning. For example, in Figure 11.3, once loop L2 is partitioned, indirection array elements edge1(i) and edge2(i) used in iteration i are moved to the processor that executes that iteration.

- *E. Inspector.* Phase E carries out the preprocessing needed for communication optimizations and index translation.
- *F. Executor.* Phase F uses information from the earlier phases to carry out computation and communication. Communication is carried out by data transportation primitives of the CHAOS library, which use communication schedules constructed in Phase E. It should be noted that the Phase F is typically executed many times in real application codes; however, phases A through E are executed only once if the data access patterns do not change. When programs change data access patterns but maintain good load balance, phases E and F are repeated. If programs require remapping of data arrays from the current distribution to a new distribution, all phases are executed again.

A wide range of languages such as Vienna FORTRAN, pC++, FORTRAN D and HPF provide a rich set of directives that allow users to specify desired data decompositions. With these directives, compilers can partition loop iterations and generate the communication required to parallelize the code. The discussion that follows is presented in the FORTRAN D context. However, the same could be extended for other languages. The following discussion involves existing FORTRAN D language support and compiler performance for irregular problems.

FORTRAN D provides users with explicit control over data partitioning using DECOMPOSITION, ALIGN and DISTRIBUTE directives. In FORTRAN D a template, called a distribution, is declared and used to characterize the significant attributes of a distributed array. The distribution fixes the size, dimension and way in which the array is to be partitioned between processors. A distribution is produced using two declarations. The first declaration is DECOMPOSITION. Decomposition fixes the name, dimensionality and size of the distributed array template. The second declaration is DISTRIBUTE. Note that DISTRIBUTE is an executable statement and specifies how a template is to be mapped onto the processors. FORTRAN D provides the user with a choice of several regular distributions. In addition, a user can explicitly specify how a distribution is to be mapped onto the processors. A specific array is associated with a distribution using the FORTRAN D statement ALIGN. In the following example, D is declared to be a two-dimensional decomposition of size N × N. Array A is then aligned with the decomposition D. Distributing decomposition D by (*,BLOCK) results in a column partition of arrays aligned with D. A detailed description of the language can be found in Fox et al. [27].

```
S1 REALA(N, N)
S2 C$ DECOMPOSITION D(N, N)
S3 C$ ALIGN A(I, J) with D(I, J)
S4 C$ DISTRIBUTE D(*, BLOCK)
```

The data distribution specifications are then treated as comment statements in a sequential machine FORTRAN compiler. Hence, a program written with distribution specifications can be compiled and executed on a sequential machine; this ensures portability.

FORTRAN D supports irregular data distributions and dynamic data decomposition (i.e., changing the alignment or distribution of a decomposition at any point in the program during the course of execution). In FORTRAN D, an irregular partition of distributed array elements can be explicitly specified. Figure 11.5 depicts an example of such a FORTRAN D declaration. In statement S3 of this figure, two 1D decompositions, each of size N, are defined. In statement S4, decomposition reg is partitioned into equal sized blocks, with one block assigned to each processor. In statement S5, array map is aligned with distribution reg. An array map is used to specify (in statement S7) how distribution irregularity is to be partitioned between processors. An irregular distribution is specified using an integer array; when map(i) is set equal to p, element i of the distribution irregularity is

```
S1 REAL*8 x(N),y(N)
S2 INTEGER map(N)
S3 C\$ DECOMPOSITION reg(N),irreg(N)
S4 C\$ DISTRIBUTE reg(block)
S5 C\$ ALIGN map with reg
S6 ... set values of map array using some mapping method ..
S7 C\$ DISTRIBUTE irreg(map)
S8 C\$ ALIGN x,y with irreg
```

FIGURE 11.5 Fortran D irregular distribution.

assigned to processor p. A data partitioner can be invoked to set the values of the permutation array. Support for irregular distributions has been provided by Vienna FORTRAN [21] as well.

The following shows an irregular FORTRAN 90D FORALL loop:

```
L2 :
FORALL (i = 1 : nedge)
S1 REDUCE (SUM, y(edge1 (i)), f(x(edge1 (i)), x(edge2 (i))))
S2 REDUCE (SUM, y(edge2 (i)), g(x(edge1 (i)), x(edge2 (i))))
END FORALL
```

In this code fragment, loop L2 represents a sweep over the edges of an unstructured mesh. Because the mesh is unstructured, an indirection array must be used to access the vertices during a loop over the edges. In loop L2, a sweep is carried out over the edges of the mesh and the reference pattern is specified by integer arrays edge1 and edge2. Loop L2 carries out reduction operations that are the only types of dependency between different iterations of the loop in which they may produce a value to be accumulated (using an associative and commutative operation) in the same array element. Loop L2 represents a sweep over the edges of a mesh in which each mesh vertex is updated using the corresponding values of its neighbors (directly connected through edges). Each vertex of the mesh is updated as many times as the number of neighboring vertices. The implementation of the FORALL construct in FORTRAN D follows copy-in-copy-out semantics; that is, the loop-carried dependencies are not defined. In the present implementation, loop-carried dependences that arise due to reduction operations are allowed. The reduction operations are specified in a FORALL construct using the FORTRAN D REDUCE construct. Reduction inside a FORALL construct is important for representing computations such as those found in sparse and unstructured problems. This representation also preserves explicit parallelism available in the underlying computations.

Once data arrays are partitioned, computational work must also be partitioned. One convention is to compute a program assignment statement S in the processor that owns the distributed array element on the lhs of S. This convention is normally referred to as the owner-computes rule. If the lhs of S references a replicated variable, then the work is carried out in all processors. One drawback to the owner-computes rule in sparse codes is that communication might be required within loops, even in the absence of loop-carried dependencies. For example, consider the following loop:

```
FORALL i = 1, N
S1 x ( ib ( i ) ) = . . . . . .
S2 y ( ia ( i ) ) = x ( ib ( i ) )
END FORALL
```

This loop has a loop-independent dependence between S1 and S2, but no loop-carried dependencies. If work is assigned using the owner-computes rule, for iteration i, statement S1 would be computed on the owner of x(ib(i)), OWNER(x(ib(i))), whereas statement S2 would be computed on the owner of y(ia(i)), OWNER(y(ia(i))). The value of y(ib(i)) would have to be communicated whenever OWNER(x(ib(i))) = OWNER(y(ia(i))). In FORTRAN D and Vienna FORTRAN, a user can specify on which processor to carry out a loop iteration using the ON clause. For example, in FORTRAN D, a loop could be written as:

```
FORALL  i  =  1, N on HOME ( x (i) )
S1  x(ib (i) )   =  . . . . . .
S2  y(ia (i) )   =  x ( ib (i) )
END  FORALL
```

This means that iteration i must be computed on the processor on which x(i) resides, where the sizes of arrays ia and ib are equal to the number of iterations. A similar HPF directive EXECUTE-ON-HOME, proposed in the journal of development, provides such a capability. A method proposed by Ponnusamy et al. [68] employs a scheme that executes a loop iteration on the processor that is the home of the largest number of distributed array references in that iteration. This is referred to as the almost owner computes rule.

Thus far, the runtime support for irregular problems has been presented in the context of the FORTRAN D system; these methods can be used in HPF compilers as well. The current version of HPF does not support nonstandard distributions. However, HPF can indirectly support such distributions by reordering array elements in ways that lead to reduced communication requirements. Applications scientists have frequently employed variants of this approach when porting irregular codes to parallel architectures.

There are also other efforts for handling irregular data accesses within a data parallel framework. In [79], the authors extend the functionality of data parallel languages to address sparse codes with an efficient code generation. Ujaldon et al. [80] describe novel methods for the representation and distribution of such data on distributed memory message passing, and propose simple language features that permit the user to characterize a matrix as "sparse" and specify the associated representation. Together with the data distribution for the matrix, this enables the compiler and runtime system to translate sequential sparse code into explicitly parallel message-passing code. They develop new compilation and runtime techniques, which focus on achieving storage economy and reducing communication overhead in the target program. The overall result is a powerful mechanism for dealing efficiently with sparse matrices in data parallel languages and their compilers for distributed-memory message-passing architectures.

Some recent work has focused on optimizing semiregular distributions. For example, Chakrabarti and Banerjee [18] demonstrate how efficient support for semiregular distributions can be incorporated in a uniform compilation framework for hybrid applications. Their strategy performs separate analyses for regular references and irregular references in a given hybrid application. After these analyses, the compiler is able to generate code to the PILAR runtime library. The compiler inserts calls to build up the preprocessing data structures, known as the inspector. This phase contains calls to the runtime library for generating translation tables and the trace arrays for every different access pattern. Calls are added for building the communication schedules once the translation tables and arrays are determined. The next phase is usually called the executor, where interprocessor communication takes place using the schedules generated in the first phase. The runtime processing includes building up of translation information for semiregular and irregular distributions, generation of trace arrays for the irregular references and computation of communication schedules. A significant feature of their runtime library is that it supports multiple internal representations suitable for pure regular, pure irregular and hybrid accesses. Experimental results on a 16-processor IBM SP-2 message-passing machine for a number of sparse applications using semiregular distributions show that such a scheme is feasible.

11.8 Input and Output Compilation

Despite the fact that the parallel file systems and runtime libraries for I/O-intensive computations provide considerable I/O performance, they require a considerable effort from the user as well. As a result, the user-optimized parallel I/O-intensive applications both consume precious time of the programmer who instead should focus on higher aspects of the program and also are not portable across a wide variety of distributed memory parallel machines (as each machine has its own application program interface [API] and support for I/O).

In this section, we concentrate on compiler techniques to optimize the I/O performance of scientific applications. In other words, we give the responsibility of keeping track of data transfers between disk subsystems and memory of parallel processors to the compiler. The main rationale behind this approach is the fact that the compiler is sometimes in a position to examine the overall access pattern of the application, and can perform I/O optimizations that conform to application behavior. Moreover, a compiler can establish a coordination with the underlying distributed memory architecture, native parallel file system of the machine and I/O libraries so that the optimizations can obtain good speedups and execution times. An important challenge for the compiler approach to I/O on distributed memory parallel machines is that the disk use, parallelism and communication (synchronization) need to be considered together to obtain a satisfying I/O performance. A compiler-based approach to the I/O problem should be able to restructure the disk resident data and computations; insert calls to the parallel file systems or libraries or both; and perform some low-level I/O optimizations.

To elaborate more on the difficulty of designing efficient compiler optimizations, let us consider an I/O-intensive data parallel program running on a distributed memory parallel machine. The primary data sets of the program can be accessed from files stored on disks. Assume that the files are striped across several disks. We can define four different working spaces [12] in which this I/O-intensive parallel program operates: a program space that consists of all the data declared in the program, a processor space that consists of all the data belonging to a processor, a file space that consists of all the data belonging to a local file of a processor and finally a disk space that contains some subset of striping units belonging to a local file. An important challenge before the compiler writers for I/O-intensive applications is to maintain the maximum degree of locality across these spaces. During the execution of I/O-intensive programs, data need to be fetched from external storage into memory. Consequently, performance of such a program depends mainly on the time required to access data. To achieve reasonable speedups, the compiler or user needs to minimize the number of I/O accesses. One way to achieve this goal is to transform the program and data sets such that the localities between those spaces are maintained. This problem is similar to that of finding appropriate compiler optimizations to enhance the locality characteristics of in-core programs; however, due to the irregular interaction between working spaces, it is more difficult. To improve the I/O performance, any application should access as much consecutive data as possible from disks. In other words, the program locality should be translated into spatial locality in disk space. Because maintaining the locality in disk space is very difficult in general, compiler optimizations attempt to maintain the locality in the file space instead.

Early work on optimizing the performance of I/O subsystem by compilation techniques came from researchers dealing with virtual memory issues. The most notable work is from Abu-Sufah et al. [1], which deals with optimizations to enhance the locality properties of programs in a virtual memory environment. Among the program transformations used are loop fusion, loop distribution and tiling (page indexing).

More recent work has concentrated on compilation of out-of-core computations using techniques based on explicit file I/O. The main difficulty is that neither sequential nor data parallel languages like HPF provide the appropriate framework for programming I/O-intensive applications. Work in the language arena offered some parallel I/O directives [16] to give hints to the compiler and runtime system about the intended use of the disk resident data. Because implementation of these language directives strongly depends on the underlying system, no general consensus exists on what kinds of primitives should be supported and how. Generally, the two feasible ways to give compiler support to I/O intensive programs are (1) using parallel file systems and (2) using parallel runtime libraries. The research has generally concentrated on using runtime libraries from a compilation framework [16].

An I/O-intensive program running on a distributed memory machine can be optimized in two ways:

- Computation transformations [13, 66]
- Data transformations [54]

The techniques based on computation transformations attempt to choreograph I/O, given high-level compiler directives mentioned earlier. The computation transformations used by the compiler for handling disk resident arrays can roughly be divided into two categories: (1) approaches based on tiling and (2) approaches based on loop permutations. Unlike in-core computation, where main data structures can be kept in memory, in I/O-intensive applications tiling is a necessity. The compiler should stage the data into memory in small granules called *data tiles*. The computation can only be performed on data tiles currently residing in memory. The computation required for other data tiles should be deferred until they are brought into memory [66]. By using the information given by directives, the compiler statically analyzes the program and performs an appropriate tiling. Note that these directives can be handled along with the ALIGN and DISTRIBUTE directives of the HPF. After tiling, the compiler has to insert the necessary I/O statements (if any) into program. Another important issue is to optimize the spatial locality in files as much as possible. This can be performed by permuting the tiling loops in the nest. Alternatively, permutation can be applied before the tiling transformations are performed. Given the fact that the accesses to the disk are much slower than accesses to processor registers, cache memory and main memory, optimizing spatial locality in files to minimize number as well as volume of the I/O transfers is extremely important.

Although for many applications, transformation based on reordering computations is quite successful, for some applications to obtain the best I/O performance, data in files should also be redistributed [54]. Unfortunately, although the computation transformations can get full benefit from the work that has been done for cache memories, there has not been much interest on data transformations until recently. This is especially true for disk resident data consumed by parallel processors. The main issue in this situation is to reach a balance between optimizing locality and maintaining a decent level of parallelism. More advanced techniques requiring unified data and computation transformations are necessary if future compilers for I/O-intensive applications are to be successful. Of course, all the compiler transformations performed to optimize disk performance of I/O-intensive programs should be followed by techniques for optimizing the accesses to data tiles currently residing in memory. Fortunately, many efforts are under way in academia for optimizing the main memory and cache performance [82, 83].

To further illustrate compilation for high-performance I/O, let us focus on a technique used for compiling out-of-core applications, details of which can be found in [56]. To translate out-of-core programs to an efficient node program (i.e., the program that can run on each node of a message-passing parallel architecture), in addition to the steps used for in-core compilation, the compiler also has to schedule explicit I/O accesses to data on disks. To accomplish this, the compiler has to take into account the data distribution on disks, the number of disks used for storing data and the prefetching and caching strategies available. The portions (known as data tiles or simply tiles) of local arrays currently required for computation are fetched from disk into memory. The larger the tiles the better, because the number of disk accesses is reduced. At a given time, each processor performs computations on the data in its tiles. Notice that the node memory should be divided suitably among tiles of different out-of-core local arrays. Thus, during the course of execution, a number of data

```
DO i = $L_i$,$U_i$,$S_i$
  DO j = $L_j$,$U_j$,$S_j$
    computation
  ENDDO j
ENDDO i
```

(a)

```
DO IT = $L_{IT}$,$U_{IT}$,$S_{IT}$
  DO JT = $L_{JT}$,$U_{JT}$,$S_{JT}$
    read data-tile(s) from local file(s)
    handle communication and storage of nonlocal data
    DO IE = $L_{IE}$,$U_{IE}$,$S_{IE}$
      DO JE = $L_{JE}$,$U_{JE}$,$S_{JE}$
        perform computation on data-tile(s) to compute
        the new data values
      ENDDO JE
    ENDDO IE
    handle communication and storage of nonlocal data
    write data-tile(s) into local file(s)
  ENDDO JT
ENDDO IT
```

(b)

```
DO IT = $L_{IT}$,$U_{IT}$,$S_{IT}$
  DO JT = $L_{JT}$,$U_{JT}$,$S_{JT}$
    read data-tile(s) from local file(s)
    execution
    write data-tile(s) into local file(s)
  ENDDO JT
ENDDO IT
```

(c)

FIGURE 11.6 (a) Example loop nest; (b) resulting node program; (c) simplified node program.

tiles belonging to a number of different out-of-core local arrays are brought into memory, the new values for these data-tiles are computed and the tiles are stored back into appropriate local files, if necessary.

It should be noted that, for now, we are concentrating on the compilation of a single-loop nest that accesses a number of out-of-core arrays. The compilation of an out-of-core application consists of two phases. In the first phase, called the *in-core phase*, the out-of-core global arrays in the source HPF program are partitioned according to the mapping information provided by compiler directives and the bounds for each local out-of-core array file are computed. Array expressions are then analyzed to compute communication sets. In other words, compilation during this phase proceeds in the same way as an in-core HPF compiler. The second phase, called the *out-of-core phase*, involves adding appropriate statements to perform explicit I/O and communication. The local arrays are first tiled based on the amount of node memory available in each processor. The resulting tiles are analyzed for communication and I/O, and finally the loops are modified to insert necessary I/O calls.

Consider the loop nest shown in Figure 11.6(a), where L_k, U_k and S_k are the lower bound, upper bound and step size for loop k, respectively. A key aspect of the compilation process is the use of tiling. Tiling (also known as *blocking*) [83] is a technique used to improve locality and parallelism, and is a combination of strip mining and loop permutation. It creates blocked (submatrix) versions of programs; when applied to a loop, it replaces it with two loops: a tiling loop and an element loop. The loop nest in Figure 11.6(a) can be translated by the compiler into the node program shown in Figure 11.6(b); in this translated code, the loops IT and JT are the tiling loops, and loops IE and JE are the element loops. Communication is allowed only at tile boundaries (outside the element loops).

It should be emphasized that the communication of the nonlocal data may involve extra file I/O on the owner side if the requested data are not currently available in the local memory of the owner processor [12]. In the rest of the chapter, for the sake of clarity, we write this translated version as shown in Figure 11.6(c). All communication statements and element loops are omitted and in the *execution* part, each reference inside the element loops is replaced by its corresponding submatrix version. For example, a reference such as A(i,j) to an out-of-core array A can be replaced by A[IT, JT]. We also show the bounds and the step sizes symbolically to keep the presentation

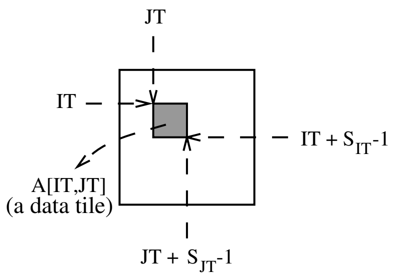


FIGURE 11.7 A data tile and its coordinates.

simple. Without loss of generality, the original nests in this chapter, such as those in Figure 11.6(a), can be assumed to iterate from 1 to n (an upper bound) with unit stride. A reference such as A[IT, JT] denotes a data tile of size $S_{IT} \times S_{JT}$ from file coordinates (IT, JT) as upper left corner to $(IT + S_{IT} - 1, JT + S_{JT} - 1)$ as lower right corner. In other words, such a reference represents a block of data instead of a single data element (see Figure 11.7). A reference such as A[IT, 1:n], on the other hand, denotes a data tile of size $S_{IT} \times n$ from $(IT, 1)$ to $(IT + S_{IT} - 1, n)$ (i.e., a block of S_{IT} consecutive rows of the out-of-core matrix A).

Because the I/O time (cost) is the dominating term in the overall cost expression in out-of-core computations and node memory is at a premium, the proposed techniques decompose the node memory among competing out-of-core local arrays such that total I/O time is minimized. Experience with out-of-core applications demonstrates that partitioning the node memory equally among competing out-of-core local arrays generally results in poor performance.

We now explain a three-step heuristic approach for optimizing I/O in out-of-core computations. First, we should explain the distinction between file layouts and disk layouts. Depending on the storage style used by the underlying file system, a file can be striped across several disks. Accordingly, an I/O call in the program may correspond to several system calls to disks. The technique described in this text attempts to reach the optimized file layouts and to minimize the number of I/O calls to the files. Of course, a reduction in I/O calls to files leads, in general, to a reduction in calls to the disks. The relationship, though, is system dependent. To illustrate the significance of I/O optimizations, we consider the example shown in Figure 11.8(a), which assumes that the arrays A, B and C are out-of-core and reside on (logical) disk in column-major order. It should be noted that in all HPF illustrations provided in this chapter, the compiler directives apply to data on files [13]. For example, a directive such as DISTRIBUTE X(BLOCK, BLOCK) ONTO P(4) partitions the out-of-core array X in (BLOCK, BLOCK) fashion across the local disks of four processors.

The compilation is performed in two phases as described before. In the in-core phase, using the array distribution information, the compiler computes the local array bounds and partitions the computation among processors. It then analyzes the local computation assigned to a processor to determine communication sets. In the second phase, tiling of the out-of-core data is carried out using the information concerning available node memory size. The I/O calls to fetch necessary data tiles for A, B and C are inserted and finally the resulting node program with explicit I/O and communication calls is generated. Figure 11.8(b) shows the resulting simplified node program (tiling loops) without any I/O optimization having been performed. Note that this translation is a straightforward extension of the in-core compilation strategy. Although it requires a relatively simple effort to generate this code, the performance of this code may be rather poor.

Our goal is to automatically derive data and computation transformations for out-of-core codes. The loop bounds and array subscripts are assumed to be affine functions of the enclosing loop indices and symbolic constants. We assume HPF-like distributions such as (BLOCK, *), (*, BLOCK) and

```
        PARAMETER (n=...,p=...)
        REAL A(n,n), B(n,n), C(n,n)
!hpf$ PROCESSORS P(p)
!hpf$ TEMPLATE D(n)
!hpf$ DISTRIBUTE D(BLOCK) ON P
!hpf$ ALIGN (:,*) WITH D :: A
!hpf$ ALIGN (*,:) WITH D :: B
        DO i = $L_i$,$U_i$,$S_i$
         DO j = $L_j$,$U_j$,$S_j$
          DO k = $L_k$,$U_k$,$S_k$
           DO l = $L_l$,$U_l$,$S_l$
            A(i,j)=A(i,j)+B(k,i)+C(l,k)+1
           ENDDO l
          ENDDO k
         ENDDO j
        ENDDO i
        END
```

(a)

```
DO IT = $L_{IT}$,$U_{IT}$,$S_{IT}$
 DO JT = $L_{JT}$,$U_{JT}$,$S_{JT}$
  read data-tile for A
  DO KT = $L_{KT}$,$U_{KT}$,$S_{KT}$
   read data-tile for B
   DO LT = $L_{LT}$,$U_{LT}$,$S_{LT}$
    read data-tile for C
    A[IT,JT]=A[IT,JT]+B[KT,IT]+C[LT,KT]+1
   ENDDO LT
  ENDDO KT
  write data-tile for A
 ENDDO JT
ENDDO IT
```

(b)

```
DO IT = $L_{IT}$,$U_{IT}$,$S_{IT}$
 read data-tile for A
 read data-tile for B
 DO LT = $L_{LT}$,$U_{LT}$,$S_{LT}$
  read data-tile for C
  A[IT,1:n]=A[IT,1:n]+B[1:n,IT]+C[LT,1:n]+1
 ENDDO LT
 write data-tile for A
ENDDO IT
```

(c)

FIGURE 11.8 (a) Example out-of-core loop nest; (b) node program resulting from a straightforward compilation; (c) I/O optimized node program.

(BLOCK, BLOCK) (i.e., each dimension of an array is either block distributed or is not distributed). As noted earlier, in out-of-core computations, the mapping directives apply to the data on files.

The loops may be imperfectly nested, but the computation inside the nest is assumed to be blockable (tile-able). Because in out-of-core computations the size of the memory is much smaller as compared with the size of the data involved, tiling is mandatory. Given a loop nest of depth g, loops i_a through i_b can be tiled if and only if for each dependence vector $(d_1, d_2, \ldots, d_a, \ldots, d_b, \ldots, d_g)^T$, either $(d_1, d_2, \ldots, d_{a-1})^T$ is lexicographically positive, or $(d_a, \ldots, d_b)^T$ is nonnegative. If necessary, the compiler performs the necessary loop transformations (e.g., loop skewing) to make tiling legal.

The number of processors, problem size and size of the available memory do not need to be known at compile time; thus, this approach involves some manipulation of symbolic expressions. Our technique does not consider chunking — a method by which rectilinear blocks are stored in files consecutively — because it seems difficult to choose suitable chunks for multiple loop nests accessing common arrays given the state-of-the-art optimizing compiler technology.

Each I/O request incurs a fixed start up time in addition to a time proportional to the amount of data requested. The start-up cost for an I/O access (e.g., file read or write), C_{io}, can be thought of as the sum of the average seek time, average rotation latency, controller delay [47] and software overhead, to initiate the data transfer. The seek time is roughly the time needed to move the disk arm to the desired track. The time for the requested sector to come under the disk head is called the *rotation latency*. Controller delay is the time spent by the disk controller satisfying the request. Finally, the software overhead includes the time spent in every I/O software layer (e.g., runtime system, file system and operating system [OS]). Let the cost of reading (or writing) a single datum from (into) a file be t_{io}. Thus, the cost of reading (writing) of ℓ consecutive elements from (into) a file can be modeled as $T = C_{io} + \ell t_{io}$. This model is highly simplified and does not take into account the fact that the file is actually striped across several disks. Our experience shows that, in general, $C_{io} \gg t_{io}$.

We assume that each processor has a memory of size M allocated for a given out-of-core computation and this memory is divided equally among all the out-of-core local arrays. We also assume that (for simplicity) each array is $n \times n$, where n is also assumed to be the trip count (number of iterations) of all the loops. For all the examples that we consider in this chapter, we assume that $dn \leq M \ll n^2$ for an integer d. The cases where $n > M$ require a radically different compilation strategy will be considered in a future work.

Suppose that the compiler works on square tiles of size $S_a \times S_a$ as shown in Figure 11.9(a). The I/O cost of a tile of size $S_a \times S_a$ is $S_a C_{io} + S_a^2 t_{io}$ and $n^2/(pS_a^2)$ of these tiles are read. The total I/O cost of the out-of-core array A is, then, $T_A = (n^2/(S_a^2 p))[S_a C_{io} + S_a^2 t_{io}]$. The I/O costs for the other arrays are computed similarly. Therefore, the overall I/O cost (time) of the nest shown in Figure 11.8(b) ($T_{overall}$) considering the file reads alone can be calculated as follows:

$$\begin{aligned}
T^a_{overall} &= \frac{n^2}{pS_a^2}\left(S_a C_{io} + S_a^2 t_{io} + \frac{n(S_a C_{io} + S_a^2 t_{io})}{S_a} + \frac{n^2(S_a C_{io} + S_a^2 t_{io})}{S_a^2}\right) \\
&= \underbrace{\frac{n^2 C_{io}}{pS_a} + \frac{n^2 t_{io}}{p}}_{T_A} + \underbrace{\frac{n^3 C_{io}}{pS_a^2} + \frac{n^3 t_{io}}{pS_a}}_{T_B} + \underbrace{\frac{n^4 C_{io}}{pS_a^3} + \frac{n^4 t_{io}}{pS_a^2}}_{T_C} \\
&= C_{io}\left(\frac{n^2}{pS_a} + \frac{n^3}{pS_a^2} + \frac{n^4}{pS_a^3}\right) + t_{io}\left(\frac{n^2}{p} + \frac{n^3}{pS_a} + \frac{n^4}{pS_a^2}\right)
\end{aligned}$$

under the memory constraint $3S_a^2 \leq M$ (with p the number of processors). T_A, T_B and T_C denote the I/O costs for the out-of-core arrays A, B and C, respectively. By assuming that the entire available memory can be used (i.e., $3S_a^2 = M$), the last of preceding formulations can be rewritten in terms of M as:

$$\begin{aligned}
T^a_{overall} &= C_{io}\left(\frac{n^2}{p\sqrt{M/3}} + \frac{n^3}{p(M/3)} + \frac{n^4}{p(M/3)\sqrt{M/3}}\right) \\
&\quad + t_{io}\left(\frac{n^2}{p} + \frac{n^3}{p\sqrt{M/3}} + \frac{n^4}{p(M/3)}\right).
\end{aligned}$$

We believe that this straightforward translation can be improved substantially by taking more care when choosing file layouts, loop ordering and tile allocations. This technique transforms the loop nest shown in Figure 11.8(a) to the nest shown in Figure 11.8(c), and associates row-major file layout for arrays A and C and column-major file layout for array B; and then the technique allocates data tiles of size $S_d \times n$ for A and C and a data tile of size $n \times S_d$ for B, as shown in Figure 11.9(d). We have assumed that the trip counts are equal to array sizes in corresponding dimensions. Otherwise, either trip count, if it is known at compile time (or an estimation of it based on the array size or a value obtained by profiling), should be used instead of n. Also because the compiler reads tiles of

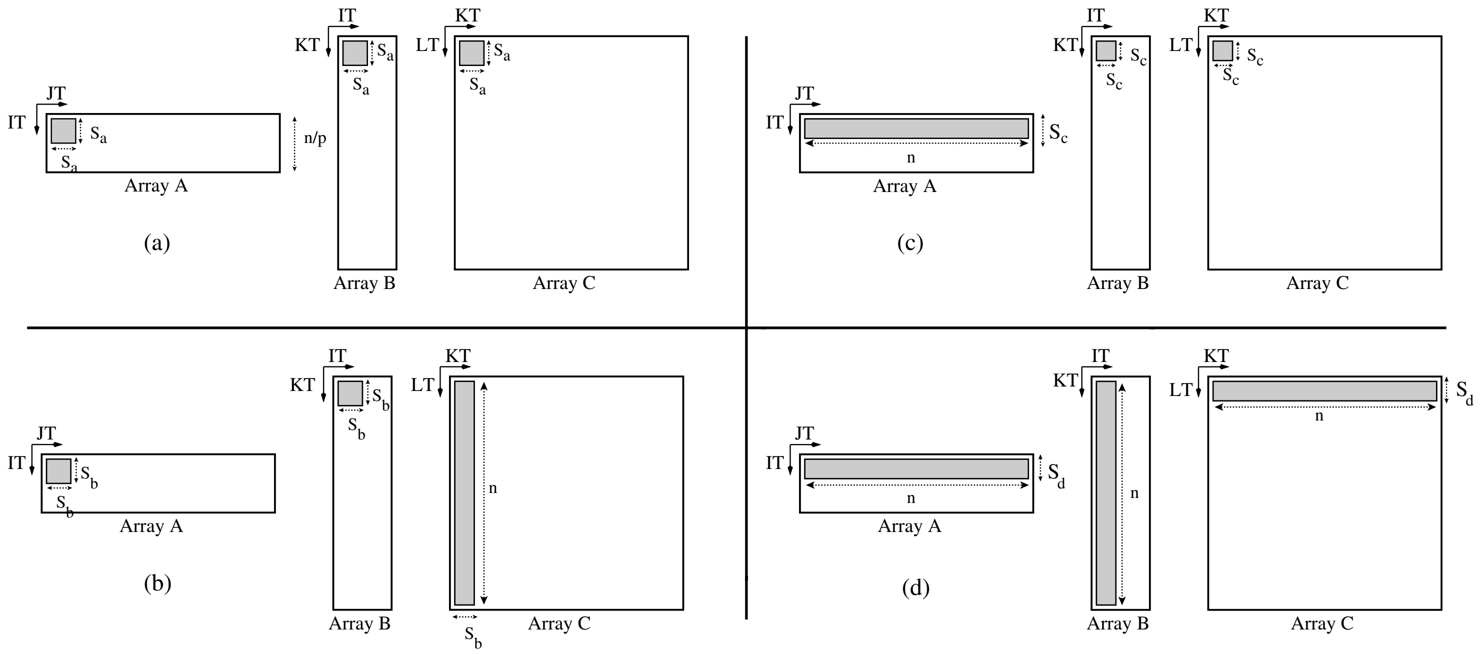


FIGURE 11.9 Out-of-core local arrays and different tile allocations for the example shown in Figure 11.8. Notice that because the size of the available memory is fixed for all memory allocations, the values for S_a, S_b, (S_c) and S_d are different from each other.

size $S_d \times n$ and $n \times S_d$, the tiling loops JT and KT disappear. The overall I/O cost of this new loop order and allocation scheme is:

$$\begin{aligned}
T^d_{overall} &= \frac{n}{pS_d}\left(S_dC_{io} + S_dnt_{io} + S_dC_{io} + nS_dt_{io} + \frac{n(S_dC_{io} + S_dnt_{io})}{S_d}\right) \\
&= \underbrace{\frac{nC_{io}}{p} + \frac{n^2t_{io}}{p}}_{T_A} + \underbrace{\frac{nC_{io}}{p} + \frac{n^2t_{io}}{p}}_{T_B} + \underbrace{\frac{n^2C_{io}}{pS_d} + \frac{n^3t_{io}}{pS_d}}_{T_C} \\
&= C_{io}\left(\frac{2n}{p} + \frac{n^2}{pS_d}\right) + t_{io}\left(\frac{2n^2}{p} + \frac{n^3}{pS_d}\right)
\end{aligned}$$

provided that $3nS_d \leq M$. Again, assuming that the maximum available memory is used (i.e., $3nS_d = M$), we can rewrite this last equation as:

$$T^d_{overall} = C_{io}\left(\frac{2n}{p} + \frac{3n^3}{pM}\right) + t_{io}\left(\frac{2n^2}{p} + \frac{3n^3}{pM}\right)$$

When the maximum available memory is used, the cost for Figure 11.9(d) is much better than that of Figure 11.9(a) (the original version). Also, to keep calculations simple, we have assumed that at most n elements can be requested in a single I/O call.

We next explain how to obtain a good combination of file layout, loop order and memory allocation, given a single-loop nest. Later in the chapter, we show how to handle the multiple loop nest case. Our approach consists of the following three steps:

1. Determination of the most appropriate file layouts for all arrays referenced in the loop nest (Section 11.8.1)
2. Permutation of the loops in the nest to maximize spatial and temporal locality (Section 11.8.2)
3. Partitioning the available memory among references based on I/O cost estimation (Section 11.8.3)

For now, we assume that the file layout for an out-of-core array may be either row-major or column-major and only one distinct reference per array exists. To be precise, we are assuming only one uniformly generated reference set (UGRS) [29] per array. For our purposes, all references in a UGRS can be treated as a single reference. We later show how our approach can be extended to handle all types of permutation-based memory layouts (i.e., row-major, column-major and higher dimensional equivalents of these layouts) and programs with multiple references to the same array. Unless otherwise stated, the word loop in the rest of the chapter refers to tiling loop. First, we make the following definitions.

Assume a loop index *IT*, an array reference R with associated file layout, and an array index position (array dimension, or subscript position) r. Also assume a data tile with size S in each dimension (subscript position) except rth dimension where its size is n provided that $n = \Theta(N) \gg S$, where N is size of the array in rth dimension. Then, *Index I/O Cost* of *IT* with respect to R, *layout* and r is the number of I/O calls required to read such a tile from the associated file into memory, if *IT* appears in the rth position of R; otherwise *Index I/O Cost* is zero. Index I/O Cost is denoted by $ICost(IT, R, r, layout)$ where *layout* can be either row-major (*row-major*) or column-major (*col-major*).

Assume a file *layout*, *Basic I/O Cost* of a loop index IT with respect to a reference R is the sum of index I/O costs of IT with respect to all index positions of reference R. That is:

$$BCost(IT, R, layout) = \sum_r ICost(IT, R, r, layout)$$

Index	Reference	Layout	BCost	Index	Reference	Layout	BCost
IT	$A[IT, JT]$	*col-major*	S	JT	$A[IT, JT]$	*col-major*	n
IT	$A[IT, JT]$	*row-major*	n/p	JT	$A[IT, JT]$	*row-major*	S
IT	$B[KT, IT]$	*col-major*	n/p	JT	$B[KT, IT]$	*col-major*	0
IT	$B[KT, IT]$	*row-major*	S	JT	$B[KT, IT]$	*row-major*	0
IT	$C[LT, KT]$	*col-major*	0	JT	$C[LT, KT]$	*col-major*	0
IT	$C[LT, KT]$	*row-major*	0	JT	$C[LT, KT]$	*row-major*	0
KT	$A[IT, JT]$	*col-major*	0	LT	$A[IT, JT]$	*col-major*	0
KT	$A[IT, JT]$	*row-major*	0	LT	$A[IT, JT]$	*row-major*	0
KT	$B[KT, IT]$	*col-major*	S	LT	$B[KT, IT]$	*col-major*	0
KT	$B[KT, IT]$	*row-major*	n	LT	$B[KT, IT]$	*row-major*	0
KT	$C[LT, KT]$	*col-major*	n	LT	$C[LT, KT]$	*col-major*	S
KT	$C[LT, KT]$	*row-major*	S	LT	$C[LT, KT]$	*row-major*	n

FIGURE 11.10 *BCost* values for the program show in Figure 11.8 (p is the number of processors).

Array	Layout	ACost	Array	Layout	ACost	Array	Layout	ACost
A	*col-major*	$S + n + 0 + 0$	B	*col-major*	$n/p + 0 + S + 0$	C	*col-major*	$0 + 0 + n + S$
A	*row-major*	$n/p + S + 0 + 0$	B	*row-major*	$S + 0 + n + 0$	C	*row-major*	$0 + 0 + S + n$

FIGURE 11.11 *ACost* values for the program shown in Figure 11.8.

Notice that *BCost* function can be used in two different ways: (1) if IT is fixed and (*R,layout*) pair is changed, it gives the I/O cost induced by loop index IT for different local array layouts, and (2) if (*R,layout*) pair is fixed and *IT* is changed over all loop indices in the nest, then it gives the I/O cost induced by R with the associated layout. The following definition employs the latter usage.

Array Cost of an array reference R, assuming a layout, is the sum of *BCost* values for all loop indices with respect to reference R. In other words:

$$ACost(R,\ layout) = \sum_{IT} BCost(IT,\ R,\ layout)$$

11.8.1 Determining File Layouts

The heuristic for determining file layouts for out-of-core local arrays first computes the *ACost* values for all arrays. It then chooses the combination that allows the compiler to perform efficient file accesses. Consider the statement $A[IT, JT] = A[IT, JT] + B[KT, IT] + C[LT, KT]$+1 in Figure 11.8(b). The *BCost* values are computed from *ICost* values as described in the last subsection. For example, $ICost(IT, A[IT, JT], 1, col\text{-}major)$ is S, because S I/O calls are needed to read a data-tile of size $n \times S$ from associated file with column-major layout. On the other hand, *ICost*(IT, $A[IT, JT]$, 2, *col-major*) is 0, because the loop index *IT* does not appear in the second subscript position of $A[IT, JT]$. Other *ICost* values can be computed similarly. Thus, for example:

$$\begin{aligned} BCost(IT, A[IT, JT], col\text{-}major) &= ICost(IT, A[IT, JT], 1, col\text{-}major) \\ &\quad + ICost(IT, A[IT], 2, col\text{-}major) \\ &= S + 0 \\ &= S \end{aligned}$$

For our loop nest, the *BCost* values are given in Figure 11.10. By using these values, the array costs (*ACost* values) for the out-of-core arrays A, B and C can be calculated, as shown in Figure 11.11.

Notice that *ACost* values are listed term by term and each term corresponds to the *BCost* of a loop index (*IT, JT, KT* and *LT*, in that order) under the given layout of the file. Next, our heuristic

Combination	Array A	Array B	Array C	Cost
1	*col-major*	*col-major*	*col-major*	$(S+n/p)+n+(S+n)+S$
2	*col-major*	*col-major*	*row-major*	$(S+n/p)+n+2S+n$
3	*col-major*	*row-major*	*col-major*	$2S+n+2n+S$
4	*col-major*	*row-major*	*row-major*	$2S+n+(S+n)+n$
5	*row-major*	*col-major*	*col-major*	$2n/p+S+(S+n)+S$
6	*row-major*	*col-major*	*row-major*	$2n/p+S+2S+n$
7	*row-major*	*row-major*	*col-major*	$(n/p+S)+S+2n+S$
8	*row-major*	*row-major*	*row-major*	$(n/p+S)+S+(n+S)+n$

FIGURE 11.12 Different layout combinations for the program shown in Figure 11.8.

considers all possible (row-major and column-major) layout combinations by summing up the *ACost* values for the components of each combination term by term. Because, in our example eight possible combinations exists, the term-by-term additions of *ACost* values are obtained (Figure 11.12).

The order of a term is the greatest symbolic value it contains. For example the order of $(S + n/p)$ is n whereas the order of $(S + 0)$ is S. A term that contains neither n nor S is called a *constant-order* term.

After creating a table of layout combinations term by term, our layout determination algorithm chooses the combination with the minimum number of n-order terms. If more than one combination occurs with the minimum number of n-order terms, we choose the one with the minimum (symbolic) cost. It is easy to see from Figure 11.12 that for our example, row-major file layout for the out-of-core arrays A and C and column-major file layout for the array B (i.e., combination 6) form a suitable (minimum I/O cost) combination, because it contains two n-order terms only: $2n/p$ and n. The other two terms in this combination are of the S-order.

We try to minimize the number of n-order terms because each n-order term indicates that the I/O cost of performing a read (or write) with the associated loop index in the innermost loop is in the order of n. Minimizing the number of n-order terms maximizes the number of constant-order and S-order terms, which, in turn, results in optimized I/O accesses.

11.8.2 Loop Order

After selecting file layouts for each array, our compiler next determines an optimum (tiling) loop order that enables efficient file accesses.

The *Total I/O Cost* of a loop index *IT* is the sum of the *Basic I/O costs* (*BCost*) of *IT* with respect to each distinct array reference it encloses. Generally speaking, *TCost*(IT) is the estimated I/O cost caused by loop IT when all array references are considered, that is:

$$T\,Cost(IT) = \sum_{R,\,layout_R} B\,Cost(IT,\, R,\, layout_R)$$

where R is the array reference and $layout_R$ is the layout of the associated file as determined in the previous step (Section 11.8.1).

The algorithm for desired loop permutation is rather simple:

1. Calculate *TCost*(IT) for each tiling loop IT.
2. If loop permutation is legal, then permute the tiling loops from outermost to innermost position according to nonincreasing values of *TCost*; if loop permutation is not legal, then the algorithm proceeds to the memory allocation step (Section 11.8.3).
3. Apply necessary loop interchanges to improve the temporal locality for (the tile of) the reference undergoing an update. This step is optional and prevents the file-write operation from occurring at innermost tiling loops.

+	n	S	C
n	n	n	n
S	n	S	S
C	n	S	C

FIGURE 11.13 Addition operation (n, S and C refer to n-order, S-order and constant-order, respectively).

Loop indices that do not have any n expression should be placed into innermost positions. Other loop indices, however, can be interchanged with one another if doing so promotes temporal locality for the references undergoing an update in the nest. Returning to our example, under the chosen file layouts (combination 6 in Figure 11.12), $TCost(IT) = 2n/p$, $TCost(JT) = S$, $TCost(KT) = 2S$ and $TCost(LT) = n$. Therefore, the desired loop permutation from the outermost to innermost position is LT, IT, KT, JT, assuming $p \geq 2$. While considering the temporal locality for the array written to, the compiler interchanges LT and IT, and reaches the order IT, LT, KT, JT.

A simple implementation of the functions defined so far (e.g., *BCost*, *ACost* and *TCost*) can be based on a suitable representation of different order terms. The compiler attaches any one of three states for each computation (and term): n (for n-order), S (for S-order) and C (for constant-order). Consequently, the addition operation used in our functions can be implemented as shown in Figure 11.13.

11.8.3 Memory Allocation

Because each node has a limited amount of memory and, in general, a loop nest may contain a number of out-of-core arrays, the node memory should be partitioned among these out-of-core arrays suitably so that the total I/O cost is minimized.

We assume that the size of every array along each dimension is greater than one. Given an array and an associated layout, we define the layout-conformant position as the index of the fastest changing dimension of the array in the layout. For column- and row-major layouts, these positions are called the column-conformant (first subscript) position and row-conformant (second subscript) position, respectively.

Our memory allocation scheme is as follows. The compiler divides the array references in the nest into two disjoint groups: a group whose associated files have row-major layout, and a group whose associated files have column-major layout. For the row-major (column-major) layout group, the compiler considers all row-conformant (column-conformant) positions in turn. If a loop index appears in the conformant position of a reference and does not appear in any other position (except the conformant) of any reference in that group, then it sets the tile size for the conformant position to n; otherwise it sets the tile size to S. For all other index positions of that reference, the tile size is set to S. After all the tile sizes for all dimensions of all array references are determined, our approach takes the size of the available memory (M) into consideration and computes the actual value for S.

As an example, suppose that in a four-deep nest in which four two-dimensional arrays A, B, C and D are referenced, our layout determination algorithm has assigned row-major file layout for the arrays A, B and C, and column-major file layout for the array D. Also assume that the references to those arrays are A[IT, KT], B[JT, KT], C[IT, JT] and D[KT, LT], where KT is the innermost loop. Our memory allocation scheme divides those references into two groups: A[IT, KT], B[JT, KT], C[IT, JT] in the row-major group, and D[KT, LT] in the column-major group. Because KT appears in the row-conformant positions of A[IT, KT] and B[JT, KT], and does not appear in any other position of any reference in this group, the tile sizes for A and B are determined as $S \times n$. Notice that JT also appears in a row-conformant position (of the reference C[IT, JT]). Because it also appears in other positions of some other references (namely, in the first subscript position of B) in this group, the compiler determines the tile size for C[IT, JT] as $S \times S$. Then, it proceeds to the other group, which contains the reference D[KT, LT] alone. Because KT is in the column-conformant position, and does not appear at any other index position of D, the compiler allocates a data tile of size $n \times S$ for D[KT, LT]. After those allocations the final memory constraint is determined as $3nS + S^2 \leq M$. Given a value for M, the value of S that utilizes all the available memory can easily be determined by solving the second-order equation $S^2 + 3nS = M$ for positive S, that is:

$$S = \left\lfloor \frac{\sqrt{9n^2 + 4M} - 3n}{2} \right\rfloor$$

Of course, the memory constraint should be adjusted accordingly. It should be noted that after these adjustments any inconsistency between those two groups (due to a common loop index) should be resolved by setting the tile size to S. For example, if, for an array with row-major layout, an innermost loop KT implies that for a certain dimension the tile size should be n, and for an array with column-major layout, for the same dimension it should be S, then the final tile size is set to S.

For our running example (Figure 11.8), the compiler divides the array references into two groups: A[IT, JT] and C[LT, KT] in the first group, and B[KT, IT] in the second group. Because JT and KT appear in the row-conformant positions of the first group and do not appear elsewhere in this group, our algorithm allocates data tiles of size $S \times n$ for A[IT, JT] and C[LT, KT]. Similarly, because KT appears in the column-conformant position of the second group and does not appear elsewhere in this group, our algorithm allocates a data tile of size $n \times S$ for B[KT, IT] as shown in Figure 11.9(d). Notice that after these tile allocations, tiling loops KT and JT disappear and the node program shown in Figure 11.8(c) is obtained.

The following discourse demonstrates that neither a fixed column-major file layout nor a fixed row-major file layout for all arrays results in optimal I/O cost in this example.

- *Figure 11.5(b).* Assume a fixed column-major file layout for all arrays. In that case, $TCost(IT) = S_b + n/p$, $TCost(JT) = n$, $TCost(KT) = S_b + n$, and $TCost(LT) = S_b$ (using the first row of Figure 11.12). Therefore (from outermost to innermost position), KT, JT, IT, LT is the desirable loop permutation. Again, considering the temporal locality for the array written to, the compiler interchanges KT and IT, and the order IT, JT, KT, LT are obtained. In other words, for a fixed column-major layout, the initial loop order is the most appropriate one. Our memory allocation scheme allocates a tile of size $n \times S_b$ for C, and tiles of size S_b^2 for each of A and B as shown in Figure 11.9(b). The reason for assigning tiles of size S_b^2 is that although IT and KT appear in column-conformant positions, they appear in other positions as well. We stress that for this and the following case (fixed row-major layouts), during memory allocation, there is only one group, and all array references belong to it. The overall I/O cost of this approach is:

$$\begin{aligned} T^b_{overall} &= \frac{n^2}{pS_b^2}\left(S_bC_{io} + S_b^2t_{io} + \frac{n(S_bC_{io} + S_b^2t_{io})}{S_b} + \frac{n(S_bC_{io} + nS_bt_{io})}{S_b}\right) \\ &= \underbrace{\frac{n^2C_{io}}{pS_b} + \frac{n^2t_{io}}{p}}_{T_A} + \underbrace{\frac{n^3C_{io}}{pS_b^2} + \frac{n^3t_{io}}{pS_b}}_{T_B} + \underbrace{\frac{n^3C_{io}}{pS_b^2} + \frac{n^4t_{io}}{pS_b^2}}_{T_C} \end{aligned}$$

  under the memory constraint $2S_b^2 + nS_b \leq M$.
- *Figure 11.5(c).* Assume a fixed row-major layout for all arrays. In that case, $TCost(IT) = n/p + S_c$, $TCost(JT) = S_c$, $TCost(KT) = n + S_c$, and $TCost(LT) = n$ (from the last row of Figure 11.12). From the outermost to innermost position KT, LT, IT, JT is the desirable

loop permutation. Once more considering the temporal locality, our compiler takes IT to the outermost position. Thus, the final loop order is IT, KT, LT, JT. The compiler allocates a tile of size $S_c n$ for A, and tiles of size S_c^2 for each of B and C as shown in Figure 11.9(c). The overall I/O cost of this approach is

$$\begin{aligned} T^c_{overall} &= \frac{n}{pS_c}\left(S_cC_{io} + S_cnt_{io} + \frac{n(S_cC_{io} + S_c^2t_{io})}{S_c} + \frac{n^2(S_cC_{io} + S_c^2t_{io})}{S_c^2}\right) \\ &= \underbrace{\frac{nC_{io}}{p} + \frac{n^2t_{io}}{p}}_{T_A} + \underbrace{\frac{n^2C_{io}}{pS_c} + \frac{n^2t_{io}}{p}}_{T_B} + \underbrace{\frac{n^3C_{io}}{pS_c^2} + \frac{n^3t_{io}}{pS_c}}_{T_C} \end{aligned}$$

under the memory constraint $2S_c^2 + nS_c \leq M$.

It should be noted that, although for reasonable values of M these costs are better than that of the original version, they are worse than the one obtained by our approach.

An important conclusion from the preceding analysis is that simply using the locality-enhancing techniques originally developed for cache-main memory hierarchy (e.g., [82]) in optimizing the out-of-core codes may not be sufficient in some cases. The main cause for this insufficiency is that these techniques assume a fixed memory layout for all arrays referenced in loop nests. There is a good reason for this assumption: all popular imperative languages (e.g., FORTRAN, C) use fixed layout representations for all arrays. Therefore, adopting different layouts for different arrays would necessitate inserting copy loops in the code that dynamically implement the desired layout at runtime. The same languages, however, do not force any fixed file layout for multidimensional arrays; thus, we have the flexibility of selecting different file layouts for different arrays through which the I/O performance of out-of-core codes can be improved significantly.

11.8.4 Overall Algorithm

Figure 11.14 shows the overall algorithm for optimizing file layouts and access patterns for the single nest case. The steps marked with $\bullet$, $\star$ and $\circ$ belong to layout determination, loop permutation and memory allocation sections of the algorithm, respectively.

11.9 Other Topics and Future Trends

Data parallelism is not the only form of parallelism that exists on distributed memory message-passing machines. Chandy et al. [20] describe two parallel programming paradigms for supporting data and task parallelism within the same framework, and show how they can be integrated. In the task parallelism, the program consists of a set of parallel tasks that interact through explicit communication. FORTRAN M is a language with which the programmer can exploit task level parallelism. As noted by Chandy et al., a major advantage of task parallelism is its flexibility, that is, its ability to exploit both structured and unstructured types of parallelism. Note that data parallelism can only exploit structured forms of parallelism. The compiler's job in such an environment is to take as input a source code annotated by task parallelism related directives, detect the communication and synchronization requirements, and optimize away communication as much as possible. In cases where the programmer explicitly uses synchronization and communication directives and commands, the compiler's job is much easier.

Data parallel languages were initially focused on regular computations and then extended with some general concepts for irregular or sparse matrix applications that do not retain the efficiency in most of the complex cases. One of the major obstacles is formed by the fact that sparse programs

```
INPUT: A loop nest that accesses a number of out-of-core array references

OUTPUT: For each out-of-core array, a le layout; for the nest a loop order; and tile sizes for each dimension of each array

NOTATION:
    $IT$: a (tiling) loop index
    $R$: an array reference (may represent a UGRS)
    $layout$: le layout for an out-of-core array ($layout \in \{$row-major, column-major$\}$)
    $layout_R$: assigned le layout for the reference $R$
    $r$: an array dimension (index position, subscript position)
    $group$: a group of references with the same le layout
    $n$: array size (and upper bound for the loops)
    $S$: a parameter that satises $S \ll n$

ALGORITHM:
    • Forall $IT, R, r$, and $layout$, compute $ICost(IT, R, r, layout)$
    • Forall $IT, R$, and $layout$, compute $BCost(IT, R, layout) = \sum_r ICost(IT, R, r, layout)$
    • Forall $R$ and $layout$, compute $ACost(R, layout) = \sum_{IT} BCost(IT, R, layout)$
    • Consider possible (row-major, column-major) layout combinations
          and choose the one with the minimum number of $n$-order terms
    • According to the chosen combination in the previous step, associate each array reference $R$ with a $layout_R$
    ⋆ Forall $IT$, compute $TCost(IT) = \sum_{R, layout_R} BCost(IT, R, layout_R)$
    ⋆ Permute the tiling loops from outermost to innermost position according to nonincreasing values of $TCost$
    ⋆ Apply necessary loop interchange(s) to improve temporal locality
    ◦ Divide the array references into $group$s according to the le layouts of the associated le s
    ◦ Forall $group$s
        ◦ Forall $R \in group$
            ◦ If a loop index appears in the conformant position of this reference
              and does not appear in any other position (except conformant) of any reference in this group,
              then set the tile size for the conformant position to $n$; otherwise set the tile size to $S$
            ◦ If there is any conict with the previous group(s), set the tile size of the conformant position to $S$
            ◦ For all other index positions set the tile size to $S$
    ◦ Determine the value of $S$ by considering the value of $M$
```

FIGURE 11.14 Overall algorithm for optimizing locality in a single-loop nest.

deal explicitly with the particular data structures selected for storing sparse matrices. This explicit data structure handling obscures the functionality of a code to such a degree that the optimization of the code is prohibited. A popular strategy for handling sparse matrix computations on distributed memory machines is the inspector or executor paradigm.

An interesting research topic that needs more work is compilation for clusters of workstations. With widespread use of clusters, we expect this problem to be even more important in future. A cluster of workstations presents a very interesting target environment for an optimizing compiler. Because the processors within a machine have access to shared memory and processors on different machines communicate through message passing, the compilation framework should employ both message-passing and shared memory compilation techniques. Obviously, this requires different parallelization techniques than current approaches because it means combining two different compilation paradigms. In addition to this, in a cluster environment, maintaining data locality is of critical importance because cost (and mechanism) of a data access depends on its location within the cluster. We particularly consider the following problems very important:

- Design and implementation of new parallelization techniques for clusters. As mentioned earlier, these techniques should integrate message-passing and shared memory paradigms. The purpose of these techniques should be minimizing intermachine communication and eliminating intramachine false sharing (i.e., false sharing of data between processors that share a common address space).

- Enhancing current data locality optimization techniques to work with cluster environments. These techniques should include both cache locality optimizations and processor locality optimizations. The latter corresponds to ensuring that the majority of data references (that miss in cache) are satisfied from local memories instead of remote memories.
- Integrating locality optimizations and parallelization techniques in a unified framework. This framework should employ both loop transformations and data transformations. Although loop transformations are local optimizations in the sense that they optimize a given program by processing one loop at a time, their application to the program is constrained by inherent data dependences. The data (or memory layout) transformations, on the other hand, are not affected by data dependences. However, modifying memory layout of a given data structure has a global effect in the program, thereby rendering the job of selecting a suitable memory layout for a given data structure very difficult. In principle, the best locality performance should be obtained using some combination of loop and data transformations. The research on this topic should investigate combined loop and data transformations for cluster environments. To investigate the trade-offs between locality enhancing transformations and parallelism optimizations, accurate cost formulations (for data accesses and communication) should be developed.
- Automatic (compiler-directed) optimization of sparse codes in cluster environments. The objective here should be embedding techniques that have previously been used in sparse runtime libraries in an optimizing compiler. The resulting framework should also be compared against existing sparse libraries.
- Integrating high-performance compiler optimizations with I/O optimizations discussed earlier in a unified framework. This is important because many data-intensive applications are also I/O intensive. In addition, the optimizations that target I/O behavior might conflict with parallelization and data locality optimizations (as both manipulate data structures and program constructs). The work is needed on techniques to resolve these conflicts when they arise.

11.10 Conclusions

This chapter presented an overview of existing techniques for distributed memory compilation and pointed to promising future directions. Given that distributed memory compilation techniques are vast and varied, we focused only on the most important compilation problems and strategies. These included techniques for data partioning, communication detection and optimization, locality optimization, handling I/O and irregular problems and many optimizations for each area.

Although many ways of generating code exist for a distributed memory architecture (depending on the type of underlying abstraction, the communication primitives and the programming interface), most of this chapter focuses on message-passing machines with two-way communication primitives and on data parallelism. This chapter should be seen as a tutorial and guide to learning about prevailing techniques for compiling for distributed memory machines, and about potential future directions.

References

[1] W. Abu-Sufah et al., On the performance enhancement of paging systems through program analysis and transformations, *IEEE Trans. Comput.*, C-30(5), 341–355, May 1981.

[2] V. Adve, J. Mellor-Crummey and A. Sethi, An integer set framework for HPF analysis and code generation, Technical report TR97-275, Computer Science Department, Rice University, Houston, TX, 1997.

[3] V. Adve and J. Mellor-Crummey, Advanced code generation for High Performance Fortran, In *Languages, Compilation Techniques, and Run-Time Systems for Scalable Parallel Systems, Lecture Notes in Computer Science,* Springer-Verlag, New York, 1998, Chapter 18.

[4] G. Agrawal and J. Saltz, Interprocedural data flow based optimizations for distributed memory compilation, *Software Pract. Exp.*, 27(5), 519–545, 1997.

[5] A.V. Aho, R. Sethi and J. Ullman, *Compilers: Principles, Techniques, and Tools.*, 2nd ed., Addison-Wesley, Reading, MA, 1986.

[6] S. Amarasinghe and M. Lam, Communication Optimization and Code Generation for Distributed Memory Machines, In Proc. SIGPLAN '93 Conference on Programming Language Design and Implementation, Albuquerque, NM, June 1993, pp. 126–138.

[7] A. Ancourt, F. Coelho, F. Irigoin and R. Keryell, A linear algebra framework for static HPF code distribution, *Sci. Programming,* 6(1), 3–28, Spring 1997.

[8] J.M. Anderson and M.S. Lam, Global Optimizations for Parallelism and Locality on Scalable Parallel Machines, in Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation, June 1993.

[9] V. Balasundaram, G. Fox, K. Kennedy and U. Kremer, An Interactive Environment for Data Partitioning and Distribution, in 5th Distributed Memory Computing Conference, Charleston, SC, April 1990.

[10] P. Banerjee, J. Chandy, M. Gupta, E. Hodges, J. Holm, A. Lain, D. Palermo, S. Ramaswamy and E. Su, The PARADIGM compiler for distributed-memory multicomputers, *IEEE Comput.*, 28(10), 37–47, October 1995.

[11] E. Barton, J. Cownie and M. McLaren, Message passing on the Meiko CS-2, *Parallel Comput.,* 20(4), 497–507, April 1994.

[12] R. Bordawekar, Techniques for Compiling I/O Intensive Parallel Programs, Ph.D. dissertation, ECE Department, Syracuse University, Syracuse, NY, May 1996.

[13] R. Bordawekar, A. Choudhary, K. Kennedy, C. Koelbel and M. Paleczny, A Model and Compilation Strategy for out-of-core Data-Parallel Programs, in Proceedings 5th ACM Symposium on Principles and Practice of Parallel Programming, July 1995.

[14] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt and S. Ranka, A compilation approach for Fortran 90D/HPF compilers, *Languages and Compilers for Parallel Computing,* U. Banerjee et al., Eds., *Lecture Notes in Computer Science*, Vol. 768, Springer-Verlag, New York, 1994, pp. 200–215.

[15] T. Brandes, Adaptor: a compilation system for data parallel Fortran programs, *Automatic Parallelization — New Approaches to Code Generation, Data Distribution, and Performance Prediction,* Vieweg, Wiesbaden, 1994.

[16] P. Brezany, T.A. Mueck and E. Schikuta, Language, compiler and parallel database support for I/O intensive applications, in *Proceedings High Performance Computing and Networking 1995 Europe*, Springer-Verlag, Heidelberg, 1995.

[17] D. Callahan and K. Kennedy, Analysis of interprocedural side effects in a parallel programming environment, *J. Parallel Distributed Comput.,* 5(5), 517–550, October 1988.

[18] D.R. Chakrabarti and P. Banerjee, A Novel Compilation Framework for Supporting Semi-Regular Distributions in Hybrid Applications, in Proceedings of the 13th International Parallel Processing Symposium and the 10th Symposium on Parallel and Distributed Processing (IPPS/SPDP '99), San Juan, Puerto Rico, April 1999.

[19] S. Chakrabarti, M. Gupta and J.-D. Choi, Global Communication Analysis and Optimization, in Proceedings ACM SIGPLAN Conference on Programming Language Design and Implementation, Philadelphia, PA, May 1996, pp. 68–78.

[20] M. Chandy, I. Foster, K. Kennedy, C. Koelbel and C.W. Tseng, Integrated support for task and data parallelism, *Int. J. Supercomputer Appl.,* 1993.

[21] B. Chapman, P. Mehrotra and H. Zima, Programming in Vienna Fortran, *Sci. Programming,* 1(1), 31–50, Fall 1992.

[22] Cray Research Inc., Cray T3D System Architecture Overview, 1993.

[23] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken and K. Yelick, Parallel Programming in Split-C, Proceedings in Supercomputing '93, Portland, OR, November 1993.

[24] R. Das, M. Uysal, J. Saltz and Y.-S. Hwang, Communication optimizations for irregular scientific computations on distributed memory architectures, *J. Parallel Distributed Comput.*, 22(3), 462–479, September 1994; also available as University of Maryland Technical report CS-TR-3163 and UMIACS-TR-93-109.

[25] A. Dierstein, R. Hayer and T. Rauber. Automatic parallelization for distributed memory multiprocessors, in *Automatic Parallelization,* Vieweg Advanced Studies in Computer Science, Vieweg Verlag, 1994, pp. 192–217.

[26] E. Duesterwald, R. Gupta and M.L. Soffa, A Practical Data-Flow Framework for Array Reference Analysis and its Application in Optimizations, in Proceedings ACM SIGPLAN '93 Conference on Programming Language Design and Implementation, Albuquerque, NM, June 1993, pp. 68–77.

[27] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.W. Tseng and M.-Y. Wu, Fortran D language specification, Technical report CRPC-TR90079, Center for Research on Parallel Computation, Rice University, Houston, TX, December 1990.

[28] M. Frumkin, H. Jin and J. Yan, Implementation of NAS Parallel Benchmarks in High-Performance FORTRAN, Proceedings IPPS/SPDP '99, San Juan, Puerto Rico, April 12–19, 1999.

[29] D. Gannon, W. Jalby and K. Gallivan, Strategies for cache and local memory management by global program transformations, *J. Parallel Distributed Comput.*, 5, 1988, 587–616.

[30] A. Geist, PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing, Scientific and Engineering Computation Series, 1994.

[31] M. Gerndt, Updating distributed variables in local computations, in *Concurrency — Pract. Exp.*, 2(3), 171–193, September 1990.

[32] J. Garcia, E. Ayguade and J. Labarta, Dynamic Data Distribution with Control Flow Analysis, in Proceedings IEEE/ACM Supercomputing '96 Conference, Pittsburgh, PA, November 1996.

[33] J. Garcia, E. Ayguade and J. Labarta, Using a 0-1 integer programming model for automatic static data distribution, *Parallel Process. Lett.*, 1996.

[34] C. Gong, R. Gupta and R. Melhem, Compilation Techniques for Optimizing Communication on Distributed-Memory Systems, in Proceedings International Conference on Parallel Processing, Vol. 2, St. Charles, IL, August 1993, pp. 39–46.

[35] E. Granston and A. Veidenbaum, Detecting Redundant Accesses to Array Data, in Proceedings Supercomputing '91, Albuquerque, NM, November 1991, pp. 854–865.

[36] M. Gupta and P. Banerjee, Demonstration of automatic datapartitioning techniques for parallelizing compilers on multicomputers, *IEEE Trans. Parallel Distributed Syst.,* 3(2), 179–193, March 1992.

[37] M. Gupta and E. Schonberg, Static Analysis to Reduce Synchronization Costs in Data-Parallel Programs, in Proceedings ACM Conference on Principles of Programming Languages, St. Petersburg, FL, 1996, pp. 322–332.

[38] M. Gupta, E. Schonberg and H. Srinivasan, A unified data-flow framework for optimizing communication, in *Languages and Compilers for Parallel Computing,* K. Pingali et al., Eds., *Lecture Notes in Computer Science*, Vol. 892, Springer-Verlag, 1995, pp. 266–282.

[39] M. Gupta, S. Midkiff, E. Schonberg, V. Seshadri, D. Shields, K. Wang, W. Ching and T. Ngo, An HPF Compiler for the IBM SP2, in Proceedings in Supercomputing 95, San Diego, CA, December 1995.

[40] M.W. Hall, S. Hiranandani, K. Kennedy and C.-W. Tseng, Interprocedural Compilation of Fortran D for MIMD Distributed-Memory Machines, in Proceedings Supercomputing '92, Minneapolis, MN, November 1992.

[41] M.W. Hall, B. Murphy, S. Amarasinghe, S. Liao and M. Lam, Inter-procedural Analysis for Parallelization, in Proceedings 8th International Workshop on Languages and Compilers for Parallel Computers, Columbus, Ohio, August 1995, pp. 61–80.

[42] R. Von Hanxeleden and K. Kennedy, A Code Placement Framework and Its Application to Communication Generation, Technical report CRPC-TR93337-S, CRPC, Rice University, Houston, TX, October 1993.

[43] K. Hayashi, T. Doi, T. Horie, Y. Koyanagi, O. Shiraki, N. Imamura, T. Shimizu, H. Ishihata and T. Shindo, AP1000+: Architectural Support of Put/Get Interface for Parallelizing Compiler, in Proceedings 6th International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, October 1994, pp. 196–207.

[44] R. Von Hanxeleden and K. Kennedy, Give-N-Take — A Balanced Code Placement Framework, in Proceedings ACM SIGPLAN '94 Conference on Programming Language Design and Implementation, Orlando, FL, June 1994.

[45] R. Von Hanxeleden, K. Kennedy and J. Saltz, Value-Based Distributions in Fortran D: A Preliminary Report, Technical report CRPC-TR93365-S, Center for Research on Parallel Computation, Rice University, Houston, TX, December 1993.

[46] J. Heinlein, K. Gharachorloo, S. Dresser and A. Gupta, Integrating Message Passing in the Stanford FLASH Multiprocessor, in Proceedings 6th International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, October 1994.

[47] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach,* 2nd ed., Morgan Kaufman, San Francisco, 1995.

[48] High Performance Fortran Forum, High Performance Fortran language specification, *Sci. Programming,* 2(1–2), 1–170, 1993.

[49] S. Hinrichs, Compiler-Directed Architecture-Dependent Communication Optimizations, Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, June 1995.

[50] S. Hinrichs, Synchronization Elimination in the Deposit Model, in Proceedings 1996 International Conference on Parallel Processing, St. Charles, IL, 1996, pp. 87–94.

[51] S. Hiranandani, K. Kennedy and C. Tseng, Compiling Fortran D for MIMD distributed-memory machines, in *Commun. ACM*, 35(8), 66–80, August 1992.

[52] C.S. Ierotheou, S.P. Johnson, M. Cross and P.F. Leggett, Computer aided parallelization tools (CAPTools) — conceptual overview and performance on the parallelization of structured and mesh codes, *Parallel Comput.*, 22(2), pp. 163–195, 1996.

[53] M. Kandemir, P. Banerjee, A. Choudhary, J. Ramanujam and N. Shenoy, A global communication optimization technique based on data-flow analysis and linear algebra, *ACM Trans. Programming Languages Syst.,* 21(6), 1251–1297, November 1999.

[54] M. Kandemir, R. Bordawekar and A. Choudhary, Data Access Reorganizations in Compiling Out-of-Core Data Parallel Programs on Distributed Memory Machines, in Proceedings of the International Parallel Processing Symposium, April 1997.

[55] M. Kandemir, A. Choudhary, P. Banerjee, J. Ramanujam and N. Shenoy, Minimizing data and synchronization costs in one-way communication, *IEEE Trans. Parallel Distributed Syst.,* 11(12), 1232–1251, December 2000.

[56] M. Kandemir, A. Choudhary and R. Bordawekar, Data Access Reorganization in Compiling Out-of-Core Data Parallel Programs on Distributed Memory Machines, in Proceedings International Parallel Processing Symposium, Geneva, Switzerland, April 1997.

[57] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman and D. Wonnacott, Omega Library Interface Guide, Technical report CS-TR-3445, CS Department, University of Maryland, College Park, MD, March 1995.

[58] K. Kennedy and N. Nedeljkovic, Combining Dependence and Data-Flow Analyses to Optimize Communication, in Proceedings 9th International Parallel Processing Symposium, Santa Barbara, CA, April 1995, pp. 340–346.

[59] K. Kennedy and A. Sethi, Resource-based communication placement analysis, *Languages and Compilers for Parallel Computing,* D. Sehr et al., Eds., *Lecture Notes in Computer Science*, Vol. 1239, Springer-Verlag, 1997, pp. 369–388.

[60] J. Knoop, O. Ruthing and B. Steffen, Optimal code motion: theory and practice, in *ACM Trans. Programming Languages Syst.,* 16(4), 1117–1155, July 1994.

[61] C. Koelbel, D. Lovemen, R. Schreiber, G. Steele and M. Zosel, *High Performance Fortran Handbook*, MIT Press, Cambridge, MA, 1994.

[62] D. Kranz, K. Johnson, A. Agarwal, J. Kubiatowicz and B.H. Lim, Integrating Message Passing and Shared Memory: Early experience, in Proceedings 4th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP '93), San Diego, CA, May 1993.

[63] U. Kremer, Automatic Data Layout for Distributed Memory Machines, Ph.D. thesis, Technical report CRPC–TR95559–S, Center for Research on Parallel Computation, October 1995.

[64] Message Passing Interface Forum, MPI-2: Extensions to the message-passing interface, July 1997.

[65] M. O'Boyle and F. Bodin, Compiler Reduction of Synchronization in Shared Virtual Memory Systems, in Proceedings International Conference on Supercomputing, Barcelona, Spain, July 1995, pp. 318–327.

[66] M. Paleczny, K. Kennedy and C. Koelbel, Compiler Support for Out-of-Core Arrays on Parallel Machines, in Proceedings of the IEEE Symposium on the Frontiers of Massively Parallel Computation, February 1995, pp. 110–118.

[67] C. Polychronopoulos, M.B. Girkar, M.R. Haghighat, C.L. Lee, B.P. Leung and D.A. Schouten, Parafrase-2: An Environment for Parallelizing, Partitioning, Synchronizing, and Scheduling Programs on Multiprocessors, in Proceedings the International Conference on Parallel Processing, St. Charles, IL, August 1989, pp. II 39–48.

[68] R. Ponnusamy, J. Saltz, A. Choudhary, Y.S. Hwang and G. Fox, Runtime Support and Compilation Methods for User-Specified Data Distributions, Technical report CS-TR-3194 and UMIACS-TR-93-135, Department of Computer Science and UMIACS, University of Maryland, College Park, November 1993.

[69] W. Pugh, A practical algorithm for exact array dependence analysis, *Commun. ACM,* 35(8), 102–114, August 1992.

[70] S. Reinhardt, J. Larus and D. Wood, Tempest and Typhoon: User-Level Shared Memory, in Proceedings 21st International Symposium on Computer Architecture, April 1994, pp. 325–336.

[71] A. Rogers and K. Pingali, Process Decomposition through Locality of Reference, in Proceedings SIGPLAN '89 Conference on Programming Language Design and Implementation, June 1989, Portland, OR, pp. 69–80.

[72] S.L. Scott, Synchronization and Communication in the T3E Multiprocessor, in Proceedings 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '96), October 1996, pp. 26–36.

[73] N. Shenoy, V.P. Bhatkar, S. Kohli and Y.N. Srikant, Automatic data partitioning by hierarchical genetic search, *J. Parallel Algorithms Architecture,* 1999.

[74] T. Stricker, J. Stichnoth, D. O'Hallaron, S. Hinrichs and T. Gross, Decoupling Synchronization and Data Transfer in Message Passing Systems of Parallel Computers, in Proceedings 9th ACM International Conference on Supercomputing, Barcelona, Spain, July 1995, pp. 1–10.

[75] E. Su, Compiler Framework for Distributed Memory Multicomputers, Ph.D. thesis, University of Illinois Urbana-Champaign, IL, March 1997.

[76] J. Subhlok, Analysis of Synchronization in a Parallel Programming Environment, Ph.D. thesis, Rice University, Houston, TX, 1990.

[77] C. Thekkath, H. Levy and E. Lazowska, Separating Data and Control Transfer in Distributed Operating Systems, in Proceedings 6th International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, October 1994, pp. 1–12.

[78] C.-W. Tseng, Compiler Optimizations for Eliminating Barrier Synchronization, in Proceedings of the 5th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP '95), Santa Barbara, CA, July 1995.

[79] M. Ujaldon and E.L. Zapata, Development and Implementation of Data-Parallel Compilation Techniques for Sparse Codes, in Proceedings of 5th Workshop on Compilers for Parallel Computers, June 1995.

[80] M. Ujaldon, E.L. Zapata, B.M. Chapman and H.P. Zima, Vienna-Fortran/HPF extensions for sparse and irregular problems and their compilation, *IEEE Trans. Parallel Distributed Syst.*, 8(10), 1068–1083, October 1997.

[81] T. von Eicken, D. Culler, S. Goldstein and K. Schauser, Active Messages: A Mechanism for Integrated Communication and Computation, in Proceedings 19th International Symposium on Computer Architecture, May 1992, pp. 256–266.

[82] M. Wolf and M. Lam, A Data Locality Optimizing Algorithm, in Proceedings ACM SIGPLAN 91 Conf. Programming Language Design and Implementation, June 1991, pp. 30–44.

[83] M. Wolfe, *High Performance Compilers for Parallel Computing*, Addison-Wesley, Redwood City, CA, 1996.

[84] X. Yuan, R. Gupta and R. Melhem, An Array Data Flow Analysis Based Communication Optimizer, in Proceedings 10th Annual Workshop on Languages and Compilers for Parallel Computing, Minneapolis, MN, August 1997.

[85] X. Yuan, R. Gupta and R. Melhem, Demand-driven data flow analysis for communication optimization, *Parallel Process. Lett.*, 7(4), 359–370, December 1997.

[86] H. Zima, H. Bast and M. Gerndt, SUPERB: A tool for semi-automatic MIMD/SIMD parallelization, *Parallel Comput.*, 6:1–18, 1988.

[87] H. Zima and B. Chapman, *Supercompilers for Parallel and Vector Computers,* ACM Press, New York, 1991.

12
Automatic Data Distribution

J. Ramanujam
Louisiana State University

12.1 Introduction

Distributed memory machines (DMMs) offer great promise because of their scalability and potential for enormous computational power. Yet, their widespread use has been hindered by the difficulty of parallel programming. Scientific programmers have had to write explicitly parallel code, and face many efficiency issues in deriving satisfactory performance. When a parallel program is run on a DMM, data need to be distributed among processors, with each processor having its own local address space; and explicit communication must be provided for the exchange of data among processors that is inherent in many programs. The processors in a DMM communicate by exchanging messages whenever a processor needs data that are located in some other processor's local memory. This exchanging of data through software is commonly referred to in the literature as *message passing* and DMMs are usually known as *message-passing computers.* Local memory accesses on these machines are much faster than those involving interprocessor communication. Deciding when to insert messages in a program, thus implementing data and computation partitioning, and which partitioning of data is optimal are no easy tasks; and much effort has gone into developing ways to relieve the programmer from this burden.

Data and computation partitioning are at the heart of the compilation process or transformation of a single processor sequential program into a single program multiple data (SPMD)

0-8493-1240-7/03/$0.00+$1.50

program to be run on a DMM. The main goal of parallelization of code is increased performance measured by reasonable speedups. However, if code is not properly parallelized, the result could be a parallelized code that may be even slower than sequential code as reported by Blume and Eigenmann [16]. One of the main sources of this undesirable degradation in execution time is the communication among processors and the overhead incurred by this communication. In most situations communication is unavoidable due to the characteristics of the code; however, it can be reduced in many instances.

The programmer faces the enormously difficult task of *orchestrating* the entire parallel execution. The programmer is forced to manually distribute code and data in addition to managing communication among tasks explicitly. This task, not only is error prone and time consuming, but also generally leads to nonportable code. Hence, parallelizing compilers for these machines have been an active area of research from the late 1980s [17, 18, 24, 31, 48, 67, 68, 70, 80, 87]. The enormity of the task is to some extent relieved by the hypercube programmer's paradigm [27] where attention is paid to the partitioning of tasks alone, while assuming a fixed data partition or a programmer-specified (in the form of annotations) data partition [28, 31, 48, 68]. In an effort to make this task easier, the High-Performance FORTRAN Forum met several times in 1992 and announced the first version of high-performance FORTRAN (HPF) in May 1993. The language was further extended in a series of meetings in 1995 and 1996. The main objective behind the efforts such as HPF, FORTRAN D, and Vienna FORTRAN (which grew out of the earlier SUPERB effort) is to raise the level of programming on DMMs while retaining the object code efficiency (derived, for example, from message passing). To achieve this objective, improved compilers and runtime systems are needed in addition to languages such as HPF.

In an important article on programming of multiprocessors, Karp [43] observed:

> *... we see that data organization is the key to parallel algorithms even on shared memory systems .... The importance of data management is also a problem for people writing automatic parallelization compilers. ... If such compilers are to be successful, particularly on message-passing systems, a new kind of analysis will have to be developed. This analysis will have to match the data structures to the executable code in order to minimize memory traffic.*

This chapter presents work aimed at providing this kind of analysis; in particular, we discuss in detail linear algebra-based techniques for partitioning data and transforming programs in a coordinated fashion. This chapter addresses the optimization of communication among processors in a DMM by properly aligning data and computation, by finding good distributions and by applying transformations that allow the use of lower overhead message-passing solutions such as message vectorization whenever possible.

The rest of this chapter is organized as follows. Section 12.2 presents a brief overview of data mapping including alignment and distribution. Section 12.3 motivates the need for communication-free partitioning and provides some examples. Section 12.4 presents detailed development and discussion of a matrix-based formulation of the problem of determining the existence of communication-free partitions of arrays; we then present conditions for the case of constant offset array access. In addition, a series of examples are presented to illustrate the effectiveness of the technique for linear references; also, we show the use of loop transformations in deriving the necessary data decompositions, generalize the formulation for arbitrary linear references and then present a formulation that aids in deriving heuristics for minimizing communication when communication-free partitions are not feasible. Section 12.5 presents a method for finding good distributions of arrays and suitable (i.e., associated) loop restructuring transformations so that communication is minimized in the execution of nested loops on message-passing machines. Unlike other work that focuses either on data layout or on program transformations, this section combines both array distributions and loop transformations resulting in good performance. The techniques described are suitable for dense linear algebra codes. Section 12.6 presents a discussion of other work on data mapping. The chapter concludes with a summary in Section 12.7.

12.2 Data Mapping

Recall that data mapping onto the processors of a DMM is accomplished in data parallel languages through a two-step process consisting of alignment followed by distribution. This section reviews the alignment and distribution phases.

12.2.1 Overview of Alignment

Alignment in data parallel programs can take the form of static alignment, dynamic alignment and replication of arrays. Static alignment refers to the alignment that is determined at compile time and dynamic alignment refers to the alignment determined at runtime. Both static and dynamic alignment can be further classified as axis, stride and reversal, and offset alignment. Static alignment is specified in the HPF standard using the `ALIGN` declaration, whereas dynamic alignment is specified through the executable statement `REALIGN` [36]. Similarly, static distribution is accomplished through the `DISTRIBUTE` declaration and refers to compile time distribution, and dynamic distribution via the `REDISTRIBUTE` executable statement at runtime.

Realignment and redistribution may be needed in those cases where the access pattern of data changes during the course of a program. Some programs may access a particular array in one fashion during the execution of a loop nest and then access the same array in a different fashion. For example, we may have a loop inside which elements of an array X are computed as functions of the elements of an array Y such as $X[i] = f(Y[i])$; we may then have some computation performed on X and then another loop with an instruction $X[i] = g(Y[2i+5])$ as shown next. The preceding notation indicates that $X[i]$ is assigned a copy of some function f or g of some element of array Y:

```
DO $i = 1, N$
    $X[i] = Y[i]$
ENDDO
DO $i = 1, N$
    $X[i] = Y[2i + 5]$
ENDDO
```

For the first loop it is advantageous to align X and Y identically, but the second loop dictates a different alignment. To reduce communication, array Y needs to be realigned before the execution of the second loop.

A common case in scientific codes involving multidimensional arrays requires transposition of one of the arrays, for example:

```
DO $i = 1, N$
    $X[i, j] = Y[i, j]$
ENDDO
DO $i = 1, N$
    $X[i, j] = Y[j, i]$
ENDDO
```

In this case array Y needs to be transposed between the loops. A redistribution may also arise, for example, because the programmer decided that it was better to distribute an array in a certain manner if the number of processors that were available was greater than or equal to some number and to distribute it in another manner if it was otherwise [73].

A program may also have a need for replication of arrays if doing so results in a reduction of communication among processors or just simply because the programmer has specified it. For example, scalars and small read only arrays may be replicated onto the processors and, in so doing, completely eliminate the communication that could have arisen because a processor needed elements owned by some other processor. Also, we may have to replicate a one-dimensional array onto a multidimensional array for reasons similar to the ones previously stated. Consider the following code:

```
DO $i = 1, N$
    DO $j = 1, M$
        $X[i, j] = Y[i, j] \cdot Z[i]$
    ENDDO
ENDDO
```

In this code, the one-dimensional read-only array Z could be replicated such that each processor owns a copy and thus can perform its computation without having to communicate, which would be the case if Z is not replicated properly. In HPF terminology, we could replicate Z along the rows or columns of a two-dimensional template to which both arrays X and Y are aligned with the result that each processor owning an element of X and Y also owns the entire array Z.

12.2.2 Overview of Distribution

The distribution phase of the data mapping problem can be defined as the phase where the abstract template, and thus all the arrays aligned to it, is mapped onto the physical processors. This phase comes after the data structures have been aligned to the template. As with the alignment phase, the distribution phase can be subdivided into static distribution and dynamic distribution.

The most commonly used distributions, which are the only ones currently available in the HPF proposed standard [36], are the cyclic, cyclic(size), block, and block(size) distributions, where size is a parameter that specifies the number of data items from a template to be assigned to a processor. The cyclic distribution assigns one element to each processor in turn until all the processors assigned to that dimension of the template are exhausted; it then assigns a new element to each processor, and continues until all the elements on that dimension of the template are assigned. As explained by Gupta and Banerjee [34], this distribution is of special importance when load balancing needs to be achieved in the presence of iteration spaces where the lower or upper bound of an iteration variable is a function of an outer iteration variable (e.g., triangular iteration spaces). On the other hand, this type of distribution is not the best choice when a lot of nearest neighbor communication exists among processors, in which case a block distribution would be preferred [34]. See Figure 12.1 for examples of cyclic distributions and the code shown next for a triangular iteration space example. Note that the lower bound for loop j is an affine function of the outer loop index variable i:

```
DO $i = 1, N$
    DO $j = i, N$
        ⋮
    ENDDO
ENDDO
```

The cyclic(size) distribution provides the programmer with the ability of specifying the number of elements that the compiler should assign to each processor in a cyclic manner. Thus, cyclic(1) produces the same effect as cyclic.

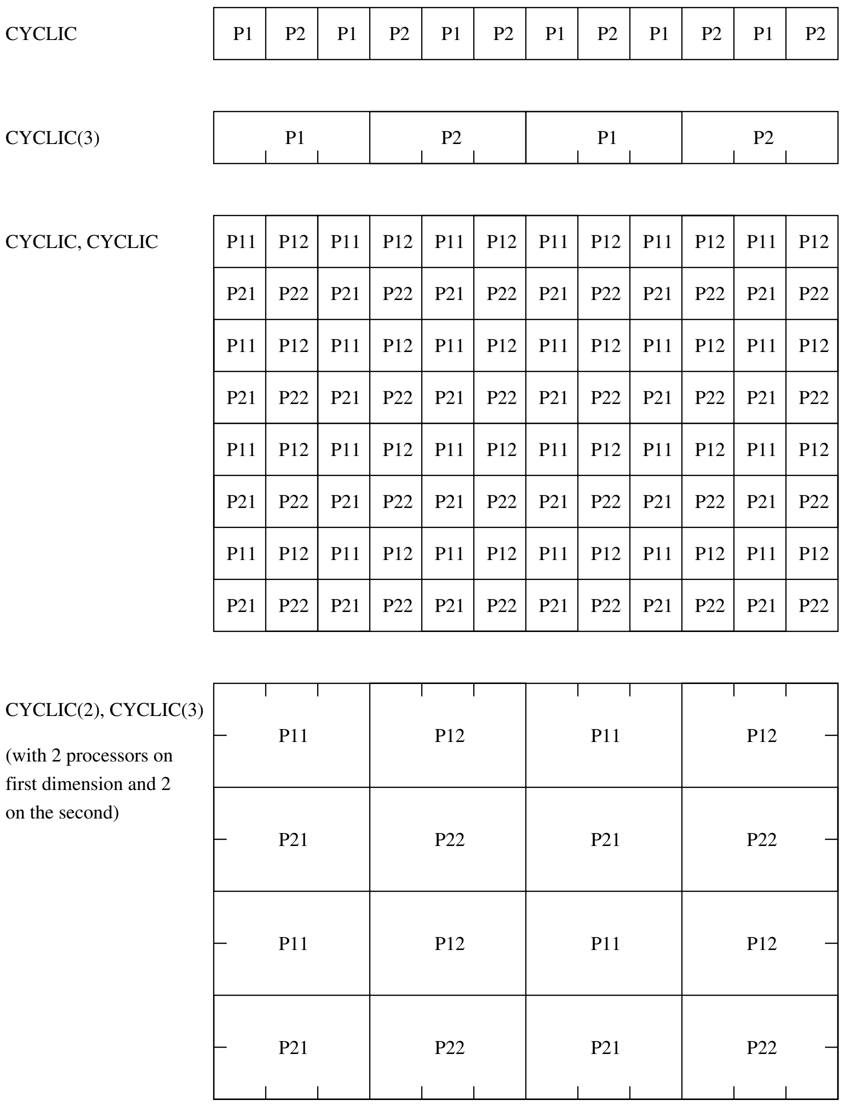

CYCLIC
P1 P2 P1 P2 P1 P2 P1 P2 P1 P2 P1 P2
CYCLIC(3)
P1 P2 P1 P2
CYCLIC, CYCLIC
P11 P12 P11 P12 P11 P12 P11 P12 P11 P12 P11 P12
P21 P22 P21 P22 P21 P22 P21 P22 P21 P22 P21 P22
P11 P12 P11 P12 P11 P12 P11 P12 P11 P12 P11 P12
P21 P22 P21 P22 P21 P22 P21 P22 P21 P22 P21 P22
P11 P12 P11 P12 P11 P12 P11 P12 P11 P12 P11 P12
P21 P22 P21 P22 P21 P22 P21 P22 P21 P22 P21 P22
P11 P12 P11 P12 P11 P12 P11 P12 P11 P12 P11 P12
P21 P22 P21 P22 P21 P22 P21 P22 P21 P22 P21 P22
CYCLIC(2), CYCLIC(3)
(with 2 processors on first dimension and 2 on the second)
P11 P12 P11 P12
P21 P22 P21 P22
P11 P12 P11 P12
P21 P22 P21 P22

FIGURE 12.1 Cyclic and cyclic(size) distribution examples.

The block distribution assigns a number of elements equal to the ceiling of the number of elements of the array in a particular dimension divided by the number of processors available for that dimension. Finally, the block(size) distribution assigns a programmer's specified number of elements to each processor. Examples are given in Figure 12.2. Note that both the block and the block(size) distributions can also be obtained from the cyclic(size) distribution. Block distributions are especially suited for rectangular iteration spaces and nearest neighbor (shift or offset) communication [34].

Skewed distributions are a more general class of distributions from which row, column, diagonal, parallelogram, etc. distributions could be derived. Both row and column distributions are one-dimensional distributions that can be obtained by skewing one dimension by a factor of zero with respect to another dimension. This factor has a nonzero value for diagonal distributions. These distributions are also referred to in the literature as hyperplanes. Skewed distributions, however general, are not currently supported by HPF [36].

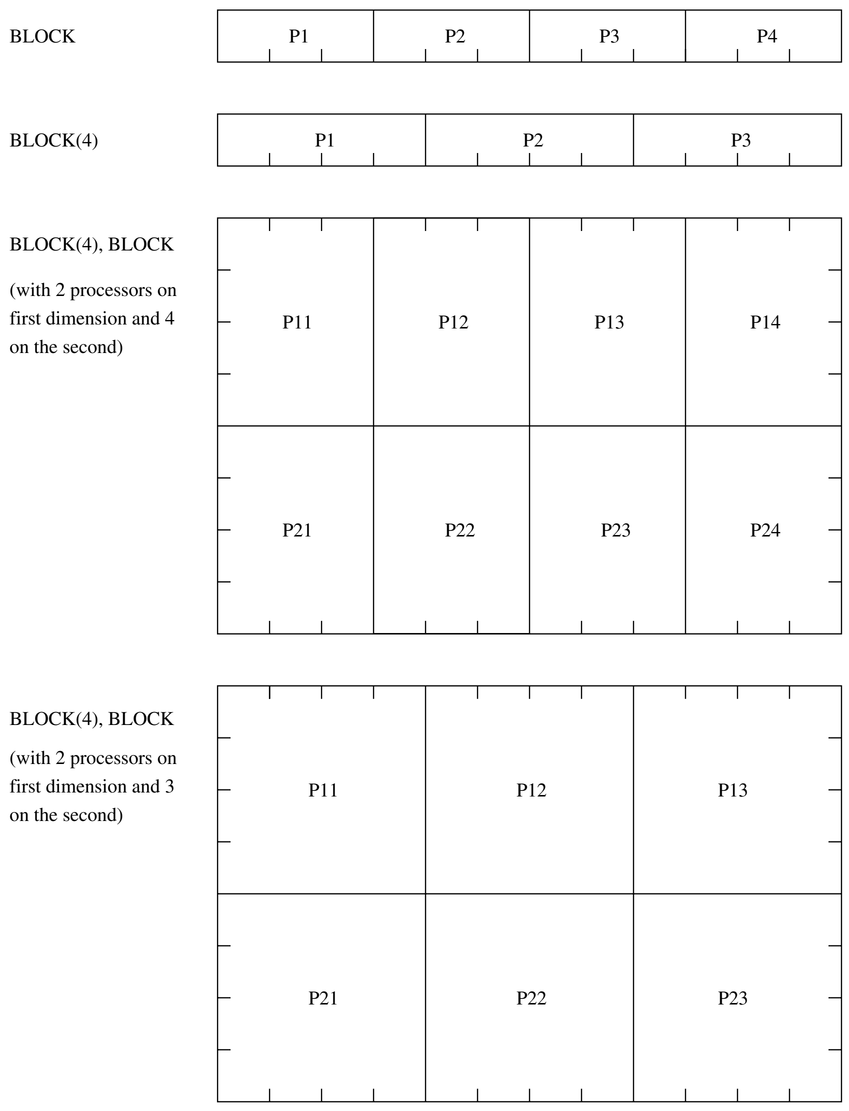

BLOCK
P1
P2
P3
P4
BLOCK(4)
P1
P2
P3
BLOCK(4), BLOCK
(with 2 processors on first dimension and 4 on the second)
P11
P12
P13
P14
P21
P22
P23
P24
BLOCK(4), BLOCK
(with 2 processors on first dimension and 3 on the second)
P11
P12
P13
P21
P22
P23

FIGURE 12.2 Block and block(size) distribution examples.

12.3 Communication-Free Partitioning

Communication in message-passing machines could arise from the need to synchronize and from the nonlocality of data. The impact of the absence of a globally shared memory on the compiler writer is severe. In addition to managing parallelism, it is now essential for the compiler writer to appreciate the significance of data distribution and decide when data should be copied or generated in local memory. We focus on distribution of arrays that are commonly used in scientific computation. Our primary interest is in arrays accessed during the execution of nested loops. We consider the following model where a processor owns a data element and has to make all updates to it, and there is exactly one copy. Even in the case of fully parallel loops, care must be taken to ensure appropriate distribution of data.

In the next sections, we explore techniques that a compiler can use to determine whether the data can be distributed such that no communication is incurred. Operations involving two or more operands require that the operands be aligned, that is, the corresponding operands are stored in the memory of the processor executing the operation. In the model we consider in this chapter, this means that the operands used in an operation must be communicated to the processor that holds the operand appearing on the left-hand side (lhs) of an assignment statement. Alignment of operands generally requires interprocessor communication.

Interprocessor communication is more time consuming than instruction execution on most message-passing machines. If insufficient attention is paid to the data allocation problem, then the amount of time spent in interprocessor communication might be so high as to seriously undermine the benefits of parallelism. It is therefore worthwhile for a compiler to analyze patterns of data usage in determining allocation to minimize interprocessor communication. We now present a machine-independent analysis of communication-free partitions. We make the following assumptions:

- Exactly one copy of every array element exists and the processor in whose local memory the element is stored is said to "own" the element.
- The owner of an array element makes all updates to the element (i.e., all instructions that modify the value of the element are executed by the "owner" processor).
- A fixed distribution of array elements exists. (Data reorganization costs are architecture specific.)

12.3.1 Examples of "Good" Partitions

Consider the following loop:

Example 12.1

```
for $i = 1$ to $N$
for $j = 4$ to $N$
$A[i, j] \leftarrow B[i, j-3] + B[i, j-2]$
```

If we allocate row i of array A and row i of array B to the local memory of the same processor, then no communication is incurred. If we allocate by columns or blocks, interprocessor communication is incurred. No data dependences occur in the preceding example; such loops are referred to as DOALL loops. It is easy to see that allocation by rows would result in zero communication because no offset in the access of B is along the first dimension. Figure 12.3 shows the partitions of arrays A and B.

In the next example, even though a nonzero offset is along each dimension, communication-free partitioning is possible.

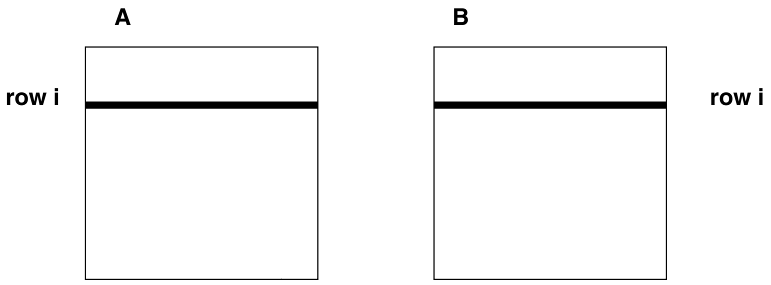


FIGURE 12.3 Partitions of arrays A and B for Example 12.1.

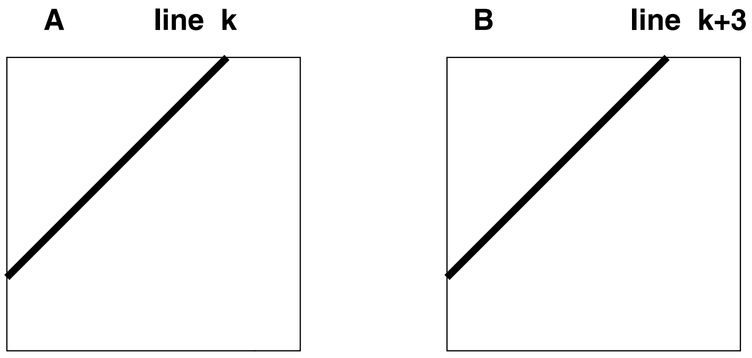


FIGURE 12.4 Partitions of arrays A and B for Example 12.2.

Example 12.2

```
for $i = 2$ to $N$
    for $j = 3$ to $N$
        $A[i, j] \leftarrow B[i + 1, j + 2] + B[i + 2, j + 1]$
```

In this case, row, column or block allocation of arrays A and B would result in interprocessor communication. In this case, if A and B are partitioned into a family of parallel lines whose equation is $i + j =$ constant (i.e., antidiagonals), no communication results. Figure 12.4 shows the partitions of A and B. The kth line in array A (i.e., the line $i + j = k$ in A and the line $i + j = k + 3$ in array B must be assigned to the same processor).

In the preceding loop structure, array A is modified as a function of array B; a communication-free partitioning in this case is referred to as a *compatible* partition. Consider the loop skeleton in Example12.3.

Example 12.3

```
for $i = 1$ to $N$
    $L_1$ : for $j = 1$ to $N$
        $A[i, j] \leftarrow f(B[i, j])$
    . . .
    . . .
    $L_2$ : for $j = 1$ to $N$
        $B[i, j] \leftarrow f(A[i, j])$
```

Array A is modified in loop L_1 as a function of elements of array B whereas loop L_2 modifies array B using elements of array A. Loops L_1 and L_2 are adjacent loops at the same level of nesting. The effect of a poor partition is exacerbated here because every iteration of the outer loop suffers interprocessor communication; in such cases, the communication-free partitioning where possible is extremely important. Communication-free partitions of arrays involved in adjacent loops are referred to as *mutually compatible* partitions.

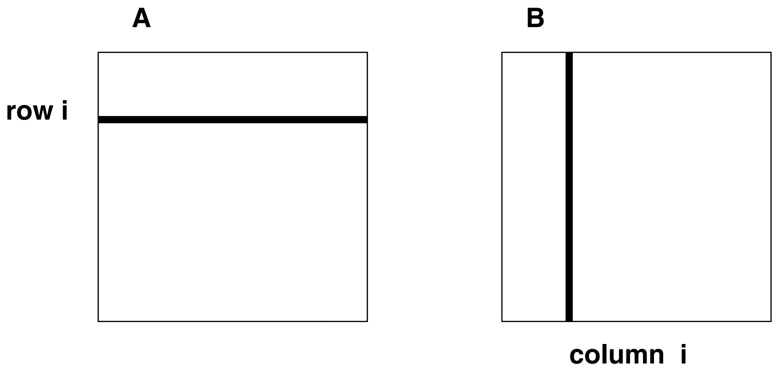


FIGURE 12.5 Array partitions for Example 12.4.

In all the preceding examples, we had the following array access pattern: for computing some element $A[i, j]$, element $B[i', j']$ is required where:

$$i' = i + c_i$$

and

$$j' = j + c_j$$

where c_i and c_j are constants. Consider Example 11.4.

Example 12.4

```
for $i = 1$ to $N$
    for $j = 1$ to $N$
        $A[i, j] \leftarrow B[j, i]$
```

In this example, allocation of *row i* of array A and *column i* of array B to the same processor would result in no communication. See Figure 12.5 for the partitions of A and B in this example. Note that i' is a function of j and j' is a function of i here.

In the presence of arbitrary array access patterns, the existence of communication-free partitions is determined by the connectivity of the data access graph, described later. To each array element that is accessed (either written or read), we associate a node in this graph. If k different arrays are accessed, this graph has k groups of nodes; all nodes belonging to a given group are elements of the same array. Let the node associated with the lhs of an assignment statement S be referred to as *write*(S) and the set of all nodes associated with the array elements on the right-hand side (rhs) of the assignment statement S be called *read-set*(S). There is an edge between *write*(S) and every member of *read-set*(S) in the data access graph. If this graph is connected, then no communication-free partition exists [67].

12.4 Communication-Free Partitioning Using Linear Algebra

Consider a nested loop of the following form that accesses arrays A and B:

Example 12.5

```
for $i = 1$ to $N$
for $j = 1$ to $N$
$A[i, j] \leftarrow B[i', j']$
```

where i' and j' are linear functions of i and j, that is:

$$i' = f(i, j) = a_{11}i + a_{12}j + a_{10} \tag{12.1}$$

$$j' = g(i, j) = a_{21}i + a_{22}j + a_{20} \tag{12.2}$$

With disjoint partitions of array A, can we find corresponding or required disjoint partitions of array B to eliminate communication? A partition of B is required for a given partition of A if the elements in that partition (of B) appear in the rhs of the assignment statements in the body of the loop that modify elements in the partition of A. For a given partition of A, the required partitions in B (are) referred to as its images or maps. We discuss array partitioning in the context of fully parallel loops. Though the techniques are presented for two-dimensional arrays, these generalize easily to higher dimensions.

In particular, we are interested in partitions of arrays defined by a family of parallel hyperplanes; such partitions are beneficial from the point of view of restructuring compilers in that the portion of loop iterations that are executed by a processor can be generated by a relatively simple transformation of the loop. Thus, the question of partitioning can be stated, Can we find partitions induced by parallel hyperplanes in A and B such that there is no communication? We focus our attention on two-dimensional arrays. A hyperplane in two dimensions is a line; hence, we discuss techniques to find partitions of A and B into parallel lines that incur zero communication. In most loops that occur in scientific computation, the functions f and g are linear in i and j.

The equation:

$$\alpha i + \beta j = c \tag{12.3}$$

defines a family of parallel lines for different values of c, given that α and β are constants and at most one of them is zero. For example:

$$\alpha = 0$$
$$\beta = 1$$

defines columns, whereas:

$$\alpha = -1,$$
$$\beta = 1$$

defines diagonals.

Given a family of parallel lines in array A defined by:

$$\alpha i + \beta j = c$$

can we find a corresponding family of lines in array B given by:

$$\alpha' i' + \beta' j' = c' \tag{12.4}$$

such that no communication exists among processors?

The conditions on the solutions are at most one of α and β can be zero; similarly, at most one of α' and β' can be zero. Otherwise, the equations do not define parallel lines. A solution that satisfies these conditions is referred to as a *nontrivial solution* and the corresponding partition is called a *nontrivial partition.* Because i' and j' are given by Equations 12.1 and 12.2, we have:

$$\alpha'\,(a_{11}i + a_{12}j + a_{10}) + \beta'\,(a_{21}i + a_{22}j + a_{20}) = c'$$

which implies:

$$\left(\alpha' a_{11} + \beta' a_{21}\right) i + \left(\alpha' a_{12} + \beta' a_{22}\right) j = c' - \alpha' a_{10} - \beta' a_{20}$$

Because a family of lines in A is defined by $\alpha i + \beta j = c$, we have:

$$\alpha = \alpha' a_{11} + \beta' a_{21} \tag{12.5}$$

$$\beta = \alpha' a_{12} + \beta' a_{22} \tag{12.6}$$

$$c = c' - \alpha' a_{10} - \beta' a_{20} \tag{12.7}$$

A solution to the preceding system of equations would imply zero communication. In matrix notation, we have:

$$\begin{pmatrix} a_{11} & a_{21} & 0 \\ a_{12} & a_{22} & 0 \\ -a_{10} & -a_{20} & 1 \end{pmatrix} \begin{pmatrix} \alpha' \\ \beta' \\ c' \end{pmatrix} = \begin{pmatrix} \alpha \\ \beta \\ c \end{pmatrix}$$

The preceding set of equations decouples into:

$$\begin{pmatrix} a_{11} & a_{21} \\ a_{12} & a_{22} \end{pmatrix} \begin{pmatrix} \alpha' \\ \beta' \end{pmatrix} = \begin{pmatrix} \alpha \\ \beta \end{pmatrix}$$

and

$$-a_{10}\alpha' - a_{20}\beta' + c' = c$$

We illustrate the use of the preceding sufficient condition with Example12.6.

Example 12.6

```
for $i = 2$ to $N$
for $j = 2$ to $N$
$A[i, j] \leftarrow B[i-1, j] + B[i, j-1]$
```

for each element $A[i, j]$, we need two elements of B. Consider the element $B[i-1, j]$. For communication-free partitioning, the system:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} \alpha' \\ \beta' \\ c' \end{pmatrix} = \begin{pmatrix} \alpha \\ \beta \\ c \end{pmatrix}$$

must have a solution. Similarly, considering the element $B[i, j-1]$, a solution must exist for the following system as well:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} \alpha' \\ \beta' \\ c' \end{pmatrix} = \begin{pmatrix} \alpha \\ \beta \\ c \end{pmatrix}$$

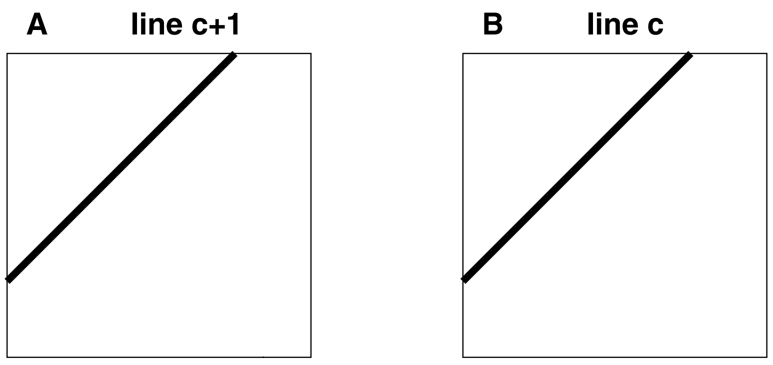


FIGURE 12.6 Partitions of arrays A and B for Example 12.6.

Given that there is a single allocation for A and B, the two systems of equations must admit a solution. This reduces to the following system:

$$\begin{aligned} \alpha &= \alpha' \\ \beta &= \beta' \\ c &= c' + \alpha' \\ c &= c' + \beta' \end{aligned}$$

The set of equations reduce to $\alpha = \alpha' = \beta = \beta'$ which has a solution say $\alpha = 1$. This implies that both A and B are partitioned by antidiagonals. Figure 12.6 shows the partitions of the arrays for zero communication. The relations between c and c' give the corresponding lines in A and B.

With a minor modification of Example 12.6, Example 12.7 shows:

Example 12.7

```
for $i = 2$ to $N$
for $j = 2$ to $N$
$A[i, j] \leftarrow B[i - 2, j] + B[i, j - 1]$
```

the reduced system of equations would be:

$$\begin{aligned} \alpha &= \alpha' \\ \beta &= \beta' \\ c &= c' + 2\alpha' \\ c &= c' + \beta' \end{aligned}$$

which has a solution $\alpha = \alpha' = 1$ and $\beta = \beta' = 2$. Figure 12.7 shows the lines in arrays A and B that incur zero communication.

The next example shows a nested loop in which arrays cannot be partitioned such that no communication exists.

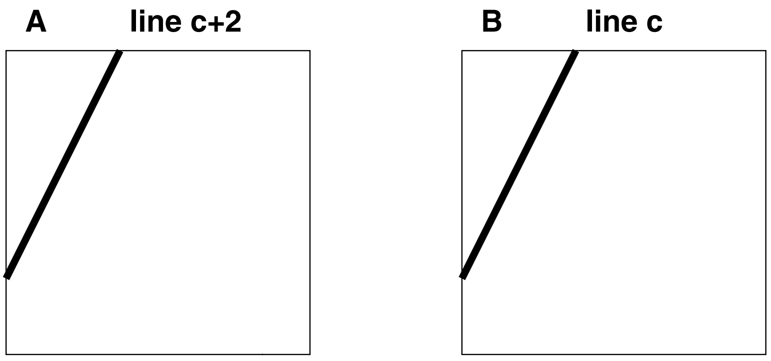


FIGURE 12.7 Lines in arrays A and B for Example 12.7.

Example 12.8

```
for $i = 2$ to $N$
  for $j = 2$ to $N$
    $A[i, j] \leftarrow B[i-2, j-1] + B[i-1, j-1] + B[i-1, j-2]$
```

The system of equations in this case is:

$$\alpha = \alpha'$$
$$\beta = \beta'$$
$$c = c' + \alpha' + \beta'$$
$$c = c' + 2\alpha' + \beta'$$
$$c = c' + \alpha' + 2\beta'$$

which reduces to

$$\alpha' + \beta' = 2\alpha' + \beta'$$
$$\alpha' + \beta' = \alpha' + 2\beta'$$

This has only one solution $\alpha' = \beta' = 0$, which is not a nontrivial solution. Thus, no communication-free partitioning of arrays A and B occurs.

The examples considered so far involve constant offsets for access of array elements and we had to find compatible partitions. The next case considered is one where we need to find mutually compatible partitions. Consider the nested loop in Example 12.9.

Example 12.9

```
for $i = 2$ to $N$
  $L_1$ : for $j = 1$ to $N$
    $A[i, j] \leftarrow B[i-1, j]$
  ...
  ...
  $L_2$ : for $j = 2$ to $N$
    $B[i, j] \leftarrow A[i, j-1]$
```

In this case, the accesses due to loop L_1 result in the system:

$$\alpha = \alpha'$$
$$\beta = \beta'$$
$$c = c' + \alpha'$$

and the accesses due to loop L_2 result in the system:

$$\alpha' = \alpha$$
$$\beta' = \beta$$
$$c' = c + \beta$$

Therefore, for communication-free partitioning, the preceding two systems of equations must admit a solution; thus, we get the reduced system:

$$\alpha' = \alpha$$
$$\beta' = \beta$$
$$\alpha' = -\beta$$

which has a solution $\alpha = \beta' = 1$ and $\alpha' = \beta = -1$. Figure 12.8 shows partitions of A and B into diagonals.

12.4.1 Constant Offsets in Reference

We discuss the important special case of array accesses with constant offsets which occur in codes for the solution of partial differential equations. Consider the following loop:

```
for $i = 1$ to $N$
for $j = 1$ to $N$
$A[i, j] \leftarrow B[i + q_1^i, j + q_1^j]$
        $+ B[i + q_2^i, j + q_2^j]$
        $+ \cdots + B[i + q_m^i, j + q_m^j]$
```

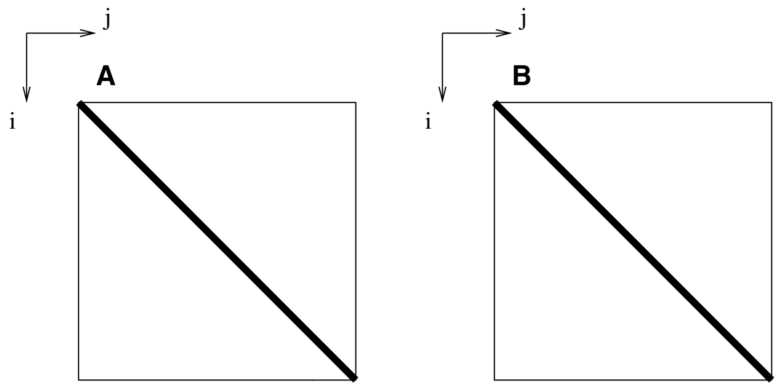


FIGURE 12.8 Mutually compatible partition of A and B for Example 12.9.

where q_k^i and q_k^j (for $1 \le k \le m$) are integer constants. The vectors $\vec{q_k} = (q_k^i, q_k^j)$ are referred to as *offset vectors.* There are m such offset vectors, one for each access pair $A[i, j]$ and $B[i + q_k^i, j + q_k^j]$. In such cases, the system of equations is (for each access indexed by k, where $1 \le k \le m$):

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -q_k^i & -q_k^j & 1 \end{pmatrix} \begin{pmatrix} \alpha' \\ \beta' \\ c' \end{pmatrix} = \begin{pmatrix} \alpha \\ \beta \\ c \end{pmatrix}$$

which reduces to the following constraints:

$$\alpha = \alpha'$$
$$\beta = \beta'$$
$$c = c' - q_k^i \alpha' - q_k^j \beta' \qquad 1 \le k \le m$$

Therefore, for a given collection of offset vectors, communication-free partitioning is possible, if:

$$c = c' - q_k^i \alpha' - q_k^j \beta' \qquad 1 \le k \le m$$

If we consider the offset vectors (q_k^i, q_k^j) as points in 2-dimensional space, then communication-free partitioning is possible, if the points (q_k^i, q_k^j) for $1 \le k \le m$ are collinear. In addition, if all $q_k^i = 0$, then no communication is required between rowwise partitions; similarly, if all $q_k^j = 0$, then partitioning the arrays into columns results in zero communication. For zero communication in nested loops involving K-dimensional arrays, this means that offset vectors treated as points in K-dimensional space must lie on a $K - 1$ dimensional hyperplane.

In all the preceding examples, there was one solution to the set of values for α, α', β, β'. In the next section, we show an example with an infinite number of solutions, and with loop transformations playing a role in the choice of a specific solution.

12.4.2 Partitioning for Linear References and Program Transformations

In this section, we discuss communication-free partitioning of arrays when references are not constant offsets but linear functions. Consider the loop in Example12.10.

Example 12.10

```
for $i = 1$ to $N$
for $j = 1$ to $N$
$A[i, j] \leftarrow B[j, i]$
```

Communication-free partitioning is possible if the system of equations:

$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \alpha' \\ \beta' \\ c' \end{pmatrix} = \begin{pmatrix} \alpha \\ \beta \\ c \end{pmatrix}$$

has a solution where no more than one of α and β is zero and no more than one of α' and β' is zero. The set reduces to:

$$\alpha = \beta'$$
$$\beta = \alpha'$$
$$c = c'$$

This set has an infinite number of solutions. We present four of them and discuss their relative merits.

The first two equations involve four variables, fixing two of which leads to values of the other two. For example, if we set $\alpha = 1$ and $\beta = 0$, then array A is partitioned into rows. From the set of equations, we get $\alpha' = \beta = 0$ and $\beta' = \alpha = 1$, which means array B is partitioned into columns; because $c = c'$, if we assign row k of array A and column k of array to the same processor, then no communication occurs. See Figure 12.3 for the partitions.

A second partition can be chosen by setting $\alpha = 0$ and $\beta = 1$. In this case, array A is partitioned into columns. Therefore, $\alpha' = \beta = 1$ and $\beta' = \alpha = 0$, which means B is partitioned into rows. See Figure 12.9(a) for this partition. If we set $\alpha = 1$ and $\beta = 1$, then array A is partitioned into antidiagonals. From the set of equations, we get $\alpha' = \beta = 1$ and $\beta' = \alpha = 1$, which means array B is also partitioned into antidiagonals. Figure 12.9(b) shows the third partition. A fourth partition can be chosen by setting $\alpha = 1$ and $\beta = -1$. In this case, array A is partitioned into diagonals. Therefore, $\alpha' = \beta = -1$ and $\beta' = \alpha = 1$, which means B is also partitioned into antidiagonals. In this case, the kth subdiagonal (below the diagonal) in A corresponds to the kth superdiagonal (above the diagonal) in array B. Figure 12.9(c) illustrates this partition.

From the point of loop transformations [5, 80], we can rewrite the loop to indicate which processor executes which portion of the loop iterations; partitions 1 and 2 are easy. Let us assume that the number of processors is p and the number of rows and columns in N and N is a multiple of p. In such a case, partition 1 (A is partitioned into rows) can be rewritten as:

`Processor` k `executes` $(1 \le k \le p)$:

```
for $i = k$ to $N$ by $p$
    for $j = 1$ to $N$
        $A[i, j] \leftarrow B[j, i]$
```

and all rows r of A such that $r \bmod p = k$ $(1 \le r \le N)$ are assigned to processor k; all columns r of B such that $r \bmod p = k$ $(1 \le r \le N)$ are also assigned to processor k.

In the case of partition 2 (A is partitioned into columns), the loop can be rewritten as:

```
Processor $k$ executes $(1 \le k \le p)$ :
for $i = 1$ to $N$
    for $j = k$ to $N$ by $p$
        $A[i, j] \leftarrow B[j, i]$
```

and all columns r of A such that $r \bmod p = k$ $(1 \le r \le N)$ and all rows r of B such that $r \bmod p = k$ $(1 \le r \le N)$ are also assigned to processor k. Because there are no data dependences anyway, the loops can be interchanged and written as:

`Processor` k executes $(1 \le k \le p)$:

```
for $j = k$ to $N$ by $p$
    for $i = 1$ to $N$
        $A[i, j] \leftarrow B[j, i]$
```

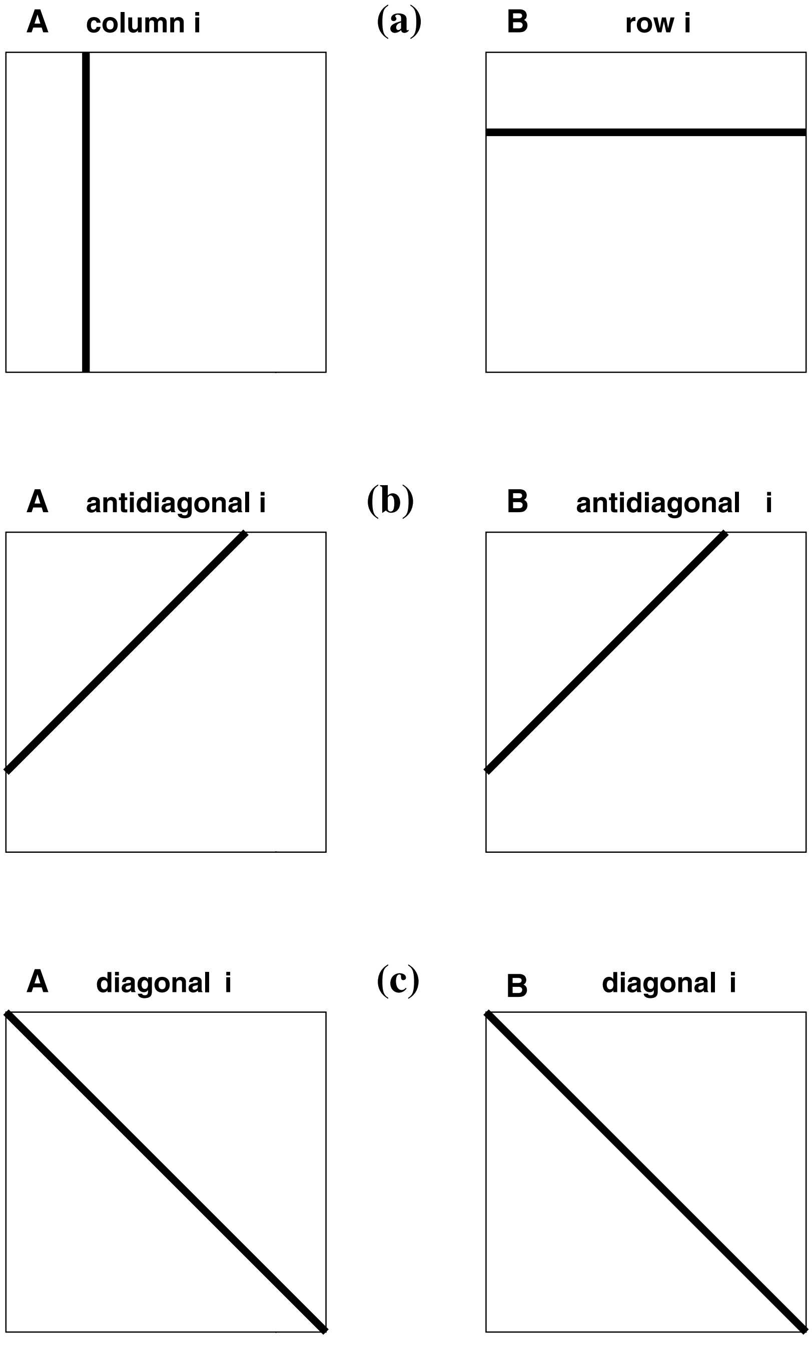


FIGURE 12.9 Decompositions of arrays A and B for Example 12.10.

Partitions 3 and 4 can result in complicated loop structures. In partition 3, $\alpha = 1$ and $\beta = 1$. The steps we perform to transform the loop use loop skewing and loop interchange transformations [80]. We perform the following steps:

1. If $\alpha = 1$ and $\beta = 0$, then distribute the iterations of the outer loop in a round-robin manner; in this case, processor k (in a system of p processors) executes all iterations (i, j) where $i = k, k+p, k+2p, \ldots, k+N-p$ and $j = 1, \ldots, N$. This is referred to as wrap distribution. If $\alpha = 0$ and $\beta = 1$, then apply loop interchange and wrap distribute the iterations of the interchanged outer loop. If not, apply the following transformation to the index set:

$$\begin{pmatrix} 1 & 0 \\ \alpha & \beta \end{pmatrix}$$

2. Apply loop interchanging so that the outer loop now can be stripmined. Because these loops do not involve flow or antidependences, loop interchanging is always legal. After the first transformation, the loop does not need to be rectangular. Therefore, the rules for interchange of trapezoidal loops in [80] are used for performing the loop interchange.

The resulting loop after transformation and loop interchange is the following:

Example 12.11

```
for $j = 2$ to $2N$
    for $i = \max(1, j - N)$ to $\min(N, j - 1)$
        $A[i, j - i] \leftarrow B[j - i, i]$
```

The load-balanced version of the loop is:
Processor k executes $(1 \le k \le p)$:

```
for $j = k + 1$ to $2N$ by $p$
    for $i = \max(1, j - N)$ to $\min(N, j - 1)$
        $A[i, j - i] \leftarrow B[j - i, i]$
```

The reason we distribute the outer loop iterations in a wraparound manner is that such a distribution results in load-balanced execution when $N \gg p$.

In partition 4, $\alpha = 1$ and $\beta = -1$. The resulting loop after transformation and loop interchange is the following:
`Processor` k `executes` $(1 \le k \le p)$:

```
for $j = k + 1 - N$ to $N - 1$ by $p$
    for $i = \max(1, 1 - j)$ to $\min(N, N - j)$
        $A[i, j + i] \leftarrow B[j + i, i]$
```

Next, we consider a more complicated example to illustrate partitioning of linear recurrences:

Example 12.12

```
for $i = 2$ to $N$
    for $j = 1$ to $N$
        $A[i, j] \leftarrow B[i + j, i] + B[i - 1, j]$
```

The access $B[i + j, i]$ results in the following system:

$$\begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \alpha' \\ \beta' \\ c' \end{pmatrix} = \begin{pmatrix} \alpha \\ \beta \\ c \end{pmatrix}$$

and the second access $B[i - 1, j]$ results in the system:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} \alpha' \\ \beta' \\ c' \end{pmatrix} = \begin{pmatrix} \alpha \\ \beta \\ c \end{pmatrix}$$

which together give rise to the following set of equations:

$$\begin{aligned} \alpha &= \alpha' + \beta' \\ \beta &= \alpha' \\ \alpha &= \alpha' \\ \beta &= \beta' \\ c &= c' + \alpha' \end{aligned}$$

which has only one solution which is $\alpha = \alpha' = \beta = \beta' = 0$. Thus communication-free partitioning has been shown to be impossible.

However, for the following loop, communication-free partitioning into columns is possible.

Example 12.13

```
for $i = 2$ to $N$
for $j = 1$ to $N$
$A[i, j] \leftarrow B[i + j, j] + B[i - 1, j]$
```

The accesses give the following set of equations:

$$\begin{aligned} \alpha &= \alpha' + \beta' \\ \alpha &= \alpha' \\ \beta &= \beta' \\ c &= c' + \alpha' \end{aligned}$$

In this case, we have a solution: $\alpha = 0$ and $\beta = 1$ giving $\alpha' = 0$ and $\beta' = 1$. Thus both A and B are partitioned into columns.

12.4.3 Generalized Linear References

In this subsection, we discuss the generalization of the problem formulation discussed earlier on in the current section.

Example 12.14

```
for $i = 1$ to $N$
for $j = 1$ to $N$
$B[i'', j''] \leftarrow A[i', j']$
```

where i', i'', j' and j'' are linear functions of i and j, that is:

$$i'' = f_l(i, j) = b_{11}i + b_{12}j + b_{10} \tag{12.8}$$

$$j'' = g_l(i, j) = b_{21}i + b_{22}j + b_{20} \tag{12.9}$$

$$i' = f(i, j) = a_{11}i + a_{12}j + a_{10} \tag{12.10}$$

$$j' = g(i, j) = a_{21}i + a_{22}j + a_{20} \tag{12.11}$$

Thus, the statement in the loop is:

$$B[b_{11}i + b_{12}j + b_{10}, b_{21}i + b_{22}j + b_{20}] \leftarrow A[a_{11}i + a_{12}j + a_{10}, a_{21}i + a_{22}j + a_{20}]$$

In this case, the family of lines in array B are given by:

$$\alpha i'' + \beta j'' = c$$

and lines in array A are given by:

$$\alpha' i' + \beta' j' = c'$$

Thus, the families of lines are:

$$\texttt{Array B:} \quad \alpha\,(b_{11}i + b_{12}j + b_{10}) + \beta\,(b_{21}i + b_{22}j + b_{20}) = c \tag{12.12}$$

$$\texttt{Array A:} \quad \alpha'\,(a_{11}i + a_{12}j + a_{10}) + \beta'\,(a_{21}i + a_{22}j + a_{20}) = c' \tag{12.13}$$

which is rewritten as:

$$\texttt{Array B:} \quad i\,(b_{11}\alpha + b_{21}\beta) + j\,(b_{12}\alpha + b_{22}\beta) = c - \alpha b_{10} - \beta b_{20} \tag{12.14}$$

$$\texttt{Array A:} \quad i\left(a_{11}\alpha' + a_{21}\beta'\right) + j\left(a_{12}\alpha' + a_{22}\beta'\right) = c' - \alpha' a_{10} - \beta' a_{20} \tag{12.15}$$

Therefore, for communication-free partitioning, we should find a solution for the following system of equations (with the constraint that at most one of α, β is zero and at most one of α', β' is zero):

$$\begin{pmatrix} a_{11} & a_{21} & 0 \\ a_{12} & a_{22} & 0 \\ -a_{10} & -a_{20} & 1 \end{pmatrix} \begin{pmatrix} \alpha' \\ \beta' \\ c' \end{pmatrix} = \begin{pmatrix} b_{11} & b_{21} & 0 \\ b_{12} & b_{22} & 0 \\ -b_{10} & -b_{20} & 1 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \\ c \end{pmatrix}$$

Consider the following example:

Example 12.15

```
for $i = 2$ to $N$
for $j = 1$ to $N$
$B[i + j, i] \leftarrow A[i - 1, j]$
```

The accesses result in the following system of equations:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} \alpha' \\ \beta' \\ c' \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \\ c \end{pmatrix}$$

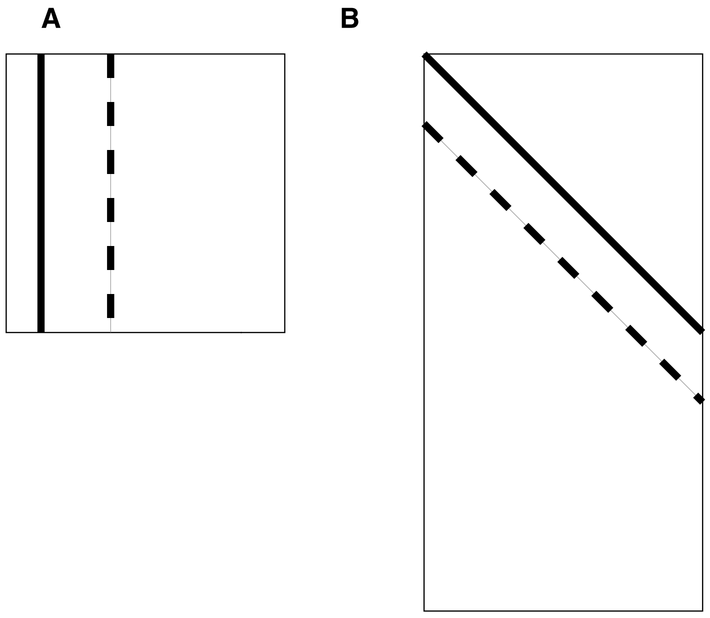


FIGURE 12.10 Partitions of arrays A and B for Example 12.15.

which leads to the following set of equations:

$$\alpha' = \alpha + \beta$$
$$\beta' = \alpha$$
$$c = c' + \alpha'$$

which has a solution $\alpha = 1, \beta = -1, \alpha' = 0, \beta' = 1$. See Figure 12.10 for the partitions.

Now for a more complicated example:

Example 12.16

```
for $i = 2$ to $N$
    for $j = 2$ to $N$
        $B[i + j - 3, i + 2] \leftarrow A[i - 1, j + 1]$
```

The accesses result in the following system of equations:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & -1 & 1 \end{pmatrix} \begin{pmatrix} \alpha' \\ \beta' \\ c' \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 0 \\ 3 & -2 & 1 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \\ c \end{pmatrix}$$

which leads to the following set of equations:

$$\alpha' = \alpha + \beta$$
$$\beta' = \alpha$$
$$\alpha' - \beta' + c' = 3\alpha - 2\beta + c$$

which has solutions where:

$$2 \begin{pmatrix}\alpha' \\ \beta'\end{pmatrix} = \begin{pmatrix}5 & -1 \\ -1 & 3\end{pmatrix} \begin{pmatrix}\alpha \\ \beta\end{pmatrix}$$

The system has the following solution: $\alpha = 1, \beta = 1, \alpha' = 2, \beta' = 1$. The loop after transformation is:

`Processor` k `executes` $(1 \leq k \leq p)$:

```
for $j = k + 4$ to $2N$ by $p$
    for $i = \max(2, j - N)$ to $\min(N, j - 2)$
        $B[j - 3, i + 2] \leftarrow A[i - 1, j - i + 1]$
```

The following subsection deals with a formulation of the problem for communication minimization, when communication-free partitions are not feasible.

12.4.4 Minimizing Communication

In this section, we present a formulation of the communication minimization problem that can be used when communication-free partitioning is impossible. We focus on two-dimensional arrays with constant offsets in accesses. The results generalize to higher dimensional arrays easily. We consider the following loop model:

```
for $i = 1$ to $N$
    for $j = 1$ to $N$
        $A[i, j] \leftarrow B[i + q_1^i, j + q_1^j] + B[i + q_2^i, j + q_2^j] + \cdots + B[i + q_m^i, j + q_m^j]$
```

The array accesses in the preceding loop give rise to the set of offset vectors, $\vec{q_1}, \vec{q_2}, \ldots, \vec{q_m}$. The $2 \times m$ matrix Q whose columns are the offset vectors q_i is referred to as the offset matrix. Because $A[i, j]$ is computed in iteration (i, j), a partition of the array A defines a partition of the iteration space and vice versa. For constant offsets, the shape of the partitions for the two arrays A and B are the same; the array boundaries depend on the offset vectors.

Given the offset vectors, the problem is to derive partitions such that the processors have equal amount of work and communication is minimized. We assume that there are N^2 iterations (N^2 elements of array A are computed) and the number of processors is p. We also assume that N^2 is a multiple of p. Thus, the workload for each processor is $\frac{N^2}{p}$.

The shapes of partitions considered are parallelograms, of which rectangles are a special case. A parallelogram is defined by two vectors each of which is normal to one side of the parallelogram. Let the normal vectors be $\vec{S_1} = (S_{11}, S_{12})$ and $\vec{S_2} = (S_{21}, S_{22})$. The matrix S refers to:

$$S = \begin{pmatrix}S_{11} & S_{12} \\ S_{21} & S_{22}\end{pmatrix}$$

If i and j are the array indices, $\vec{S_1}$ defines a family of lines given by $S_{11}i + S_{12}j = c_1$ for different values of c_1 and the vector $\vec{S_2}$ defines a family of lines given by $S_{21}i + S_{22}j = c_2$ for different values of c_2. S must be nonsingular to define parallelogram blocks that span the entire array. The matrix S defines a linear transformation applied to each point (i, j); the image of the point (i, j) is

$(S_{11}i + S_{12}j, S_{21}i + S_{22}j)$. We consider parallelograms defined by solutions to the following set of linear inequalities:

$$\begin{aligned} S_{11}i + S_{12}j &\geq c_1 \\ S_{11}i + S_{12}j &< c_1 + r_1 l \\ S_{21}i + S_{22}j &\geq c_2 \\ S_{21}i + S_{22}j &< c_2 + r_2 l \end{aligned}$$

where $r_1 l$ and $r_2 l$ are the lengths of the sides of the parallelograms.

The number of points in the discrete Cartesian space enclosed by this region (which must be the same as the workload for each processor, $\frac{N^2}{p}$) is $l^2 \frac{r_1 r_2}{|det(S)|}$ when $det(S) \neq 0$. The nonzero entries in the matrix $Q' = SQ$ represent inter-processor communication. Let $Q'(i)$ be the sum of the absolute values of the entries in the ith row of Q', that is:

$$Q'(i) = \sum_{j=1}^{m} |Q'_{i,j}|$$

The communication volume incurred is:

$$2l\left(Q'(1)\frac{r_1}{|det(S)|} + Q'(2)\frac{r_2}{|det(S)|}\right) \tag{12.16}$$

Thus, the problem of finding blocks that minimize interprocessor communication is that of finding the matrix S, the value l and the aspect ratios r_1 and r_2 such that the communication volume is minimized subject to the constraint that the processors have about the same amount of workload, that is:

$$l^2 \frac{r_1 r_2}{|det(S)|} = \frac{N^2}{p}$$

The elements of matrix S determine the shape of the partitions and the values of r_1, r_2, l determine the size of the partitions.

12.5 Finding Distributions and Loop Transformations Using a Matrix-Based Approach

This and the following sections present a technique for finding good distributions of arrays and suitable loop-restructuring transformations so that communication is minimized in the execution of nested loops on message-passing machines. For each possible distribution (by one or more dimensions), we derive the best unimodular loop transformation that results in block transfers of data. Unimodular matrices have a determinant with a value of ± 1 and they are used extensively in parallelizing compilers [12]. Unlike other work that focuses either on data layout or on program transformations, this section combines both array distributions and loop transformations resulting in good performance. The techniques described here are suitable for dense linear algebra codes.

On a DMM, local memory accesses are much faster than accesses to nonlocal data. When a number of nonlocal accesses are to be made between processors, it is preferable to send fewer but larger messages instead of several smaller messages more frequently (called *message vectorization* [73]). This is because the message setup cost is usually large. Even in shared memory machines, it is preferable to use block transfers. Interprocessor communication time is usually modeled as $t = \alpha + \beta * \gamma$, where α and β are machine dependent and γ is the length of the message. Usually $\alpha \gg \beta$ and thus it is desirable to communicate longer messages instead of short ones whenever possible.

Given a program segment, our aim is to determine the computation and data mapping onto processors. Parallelism can be exploited by transforming the loop nest suitably and then distributing the iterations of the transformed outermost loop onto the processors. The distribution of data onto processors may then result in communication and synchronization that counter the advantages obtained by parallelism. This section presents an algorithm that results in the optimal performance while simultaneously considering the conflicting goals of parallelism and data locality.

Although a programmer can manually write code to enhance data locality by specifying data distribution among processors, we present a technique where we can automatically derive data distribution given the program structure. We present a method by which the program is restructured such that when the outer loop iterations are mapped onto the processors, it results in the least communication. Wherever communication is unavoidable, we restructure the inner loops so that data can be transferred using block transfers (the message vectorization approach). Our approach relieves the programmer from having to specify the distribution of the arrays and optimize the communication among processors in case this communication is unavoidable.

This section is organized as follows: Subsection 12.5.1 talks about the need for automatic distribution; Subsection 12.5.2 introduces our first algorithm for automatic distribution and vectorization of messages; Subsection 12.5.3 is a step-by-step application of the algorithm to several examples; Subsection 12.5.4 shows the advantage of relaxing the owner-computes rule when our algorithm does not find a solution; Subsection 12.5.5 presents our extended algorithm and applies it to an example in a step-by-step fashion.

12.5.1 Automatic Distribution

We consider those cases where we allocate outer iterations to processors so that each outer loop iteration is done by a single processor. The data are then allocated so that a minimum of communication occurs and all communication is done through block transfers. This section deals with an algorithm to restructure the program to enhance data locality while still enabling parallelism. We construct the entries of a legal invertible transformation matrix so that there is a one-to-one mapping from the original iteration space to the transformed iteration space. This transformation when applied to the original loop structure does the following:

- Allow the outermost loop iteration to be distributed over the processors (i.e., an entire outermost iteration is mapped on to a single processor).
- Determine the data distribution (block or cyclic distribution of a single-array dimension).
- Allow blocks transfers to be moved out of the innermost loop so that all the necessary data are transferred to the respective local memories before the execution of the innermost loop.

12.5.1.1 Relevant Background

The transformation matrix is derived from the data reference matrix of the array references. Given a loop nest with indices $i_1, i_2, \ldots, i_n$ which is represented by a column vector $\vec{I}$, we define a *data reference matrix*, $A_{\mathcal{R}}$, for each array reference A (distinct or nondistinct) in a loop nest such that the array reference can be written in the form $A_{\mathcal{R}}\vec{I} + \vec{b}$ where $\vec{b}$ is the offset vector. In what follows we assume that the arrays are not replicated onto the available processors. Consider the following loop nest.

Example 12.17

```
DO $i = 1, N_1$
  DO $j = 1, N_2$
    DO $k = 1, N_3$
      $B[i, j - i] = B[i, j - i] + A[i, j + k]$
    ENDDO
  ENDDO
ENDDO
```

In the preceding example, the data reference matrix for array B is

$$B_{\mathcal{R}} = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \end{bmatrix}$$

and the data reference matrix for array A is

$$A_{\mathcal{R}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix}$$

Note that there are two data reference matrices for array B though they are identical. For each array, we use only the distinct data reference matrices.

12.5.1.2 Effect of a Transformation

On applying a transformation T to a loop with index I, the transformed loop index becomes $\vec{I'} = T\vec{I}$ and the transformed data reference matrix becomes $A'_{\mathcal{R}} = A_{\mathcal{R}}T^{-1}$. The columns of T^{-1} determine the array subscripts of the references in the transformed loop. The key aspect of the algorithm presented in this section is that the entries of the inverse of the transformation matrix are derived using the data reference matrices.

Li and Pingali [56] discuss the completion of partial transformations derived from the data access matrix of a loop nest; the rows of the data access matrix are subscript functions for various array accesses (excluding constant offsets). Their work assumes that all arrays are distributed by columns. In contrast, our work attempts to find the best distribution for various arrays (by rows, columns, or blocks) such that communication incurred is minimal; for each possible combination of distribution of arrays, we find the best compound loop transformation that results in least communication. Among all these possible distributions (and the associated loop restructuring), we find the one that incurs the smallest communication overhead. Whereas several researchers have addressed the issue of automatic alignment, none except Anderson and Lam [7] addresses the interaction of program transformations and data mapping.

12.5.1.3 Motivation and Assumptions

Consider Example12.17 that is similar to the one given by Li and Pingali [56]. Two references to array B (though not distinct) and one reference to array A exist. Li and Pingali [56] assume that all arrays are distributed by columns and derive a transformation matrix that matches column distribution. In this case, the loop can be distributed in such a way that no communication is incurred. Both the arrays can be distributed by rows (i.e., each processor can be assigned an entire row of array A and an entire row of array B). This makes the loop run without any communication. We notice that the first row in the data reference matrix for arrays A and B are the same, that is, [1 0 0]. This allows the first dimension of both the lhs and rhs arrays to be distributed (i.e., by rows) over the processors so that there is no communication. In the next section, we derive an algorithm to construct a transformation matrix, which determines the distribution of data.

We restrict our analysis to affine array references in loop nests whose upper and lower bounds are affine. We assume that the iterations of the outermost loop are distributed among processors. To exploit data locality and reduce communication among processors, we further look at transformations that facilitate block transfers so that the data elements that are referenced are brought to local memory in large chunks; this allows us to amortize the high message start-up costs over large messages. We assume that the data can be distributed along any one dimension of the array (wrapped or blocked) and that the loop index variable appearing in the subscript expression of the distributed dimension of our base array and any array that is identically distributed is that corresponding to the outermost loop. The results can be generalized where data are distributed along multiple dimensions and block transfers are set up in outer iterations. Again we assume that the arrays used in the iterations of the loop nest are not replicated onto the available processors.

12.5.1.4 Criteria for Choosing the Entries in the Transformation Matrix

Let the array indices of the original loop be $i_1, i_2, \ldots, i_n$. Let the array indices of the transformed loop be $j_1, j_2, \ldots, j_n$. We look for transformations such that the lhs array has the outermost loop index as the only element in any one of the dimensions of the array, e.g., $C[*, *, \ldots, j_1, \ldots, *]$ where j_1 is in the rth dimension and "$*$" indicates a term independent of j_1. The lhs array can then be distributed along dimension r. This means that the data reference matrix $C'_{\mathcal{R}}$ of the transformed array reference C has at least one row in which the first entry is nonzero and the rest are zero (i.e., there is a row r in $C'_{\mathcal{R}}T^{-1} = [\alpha, 0, 0, \ldots, 0]$). For all arrays that appear on the rhs:

- If a row in all the data reference matrices of an array is identical to a row in the reference matrix in the lhs array, then that array can be distributed in the same way as the lhs array. No communication is due to that array, because the arrays are always mapped onto the same processor. If all the references of all the arrays have a row in the data reference matrix identical to that of the lhs array, then the entire loop can be distributed along that dimension and there is no communication.
- If the preceding condition does not hold, choose the entries in T^{-1} such that the following conditions hold:

  1. Some dimension of the rhs reference consists only of the transformed innermost loop index, e.g., $A[*, \ldots, j_n, \ldots, *]$.
  2. All the other dimensions are independent of the innermost loop index (i.e., "$*$" indicates a term independent of j_n).

  This means the transformed reference matrix must have only one nonzero in some row r, and that nonzero must occur in column n. If this condition is satisfied, then dimension r of the rhs array is not a distributed dimension; thus, we can move communication arising from that rhs reference outside the innermost loop. This allows a block transfer to the local memory before the execution of the innermost loop. This means that a row in the transformed data reference matrix $A'_{\mathcal{R}}$ has a row with all entries zero except in the last column, which is nonzero. Also, the last column of the $A'_{\mathcal{R}}$ has all remaining entries as zero.
- If communication could be moved out of the innermost loop, the previous step can be applied repeatedly starting with the deepest loop outside the innermost and working outward; this process can either stop at some level of the outside that communication cannot be moved or when no more loops in the loop nest are to be considered.

The transformation should also satisfy the condition that the determinant is ± 1 and must preserve the dependences in the program.

12.5.2 The Algorithm

Consider the following loop where n is the loop nesting level and d is the dimension of the arrays:

```
DO $i_1 = 1, N_1$
  ...
    DO $i_n = 1, N_n$
      $L[C]\vec{I} + \vec{B}^l = R[A]\vec{I} + \vec{B}^r$
    ENDDO
  ...
ENDDO
```

where:

$$C = \begin{bmatrix} c_{11} & \dots & c_{1n} \\ \ddots & & \\ c_{d1} & \dots & c_{dn} \end{bmatrix} \text{ and } A = \begin{bmatrix} a_{11} & \dots & a_{1n} \\ \ddots & & \\ a_{d1} & \dots & a_{dn} \end{bmatrix}$$

are the access or reference matrices for the lhs array L and rhs array R, respectively:

$$\vec{I} = \begin{bmatrix} i_1 \\ \vdots \\ i_n \end{bmatrix}$$

is the iteration vector, and:

$$\vec{B}^l = \begin{bmatrix} b_1^l \\ \vdots \\ b_d^l \end{bmatrix} \text{ and } \vec{B}^r = \begin{bmatrix} b_1^r \\ \vdots \\ b_d^r \end{bmatrix}$$

are the constant offset vectors for the lhs and rhs arrays, respectively. Let the inverse of the transformation matrix be:

$$Q = T^{-1} = \begin{bmatrix} q_{11} & \dots & q_{1n} \\ \ddots & & \\ q_{n1} & \dots & q_{nn} \end{bmatrix}$$

The algorithm is shown in Table 12.1. We use the notation $\vec{A}[i, :]$ to refer to the ith row of a matrix A, and $\vec{A}[:, j]$ to refer to the jth column of a matrix A.

12.5.3 Examples of Automatic Distribution

We illustrate the use of the algorithm through several examples in this section. The reader is referred to the work by Ramanujam and Narayan [66] for a detailed discussion of the algorithm. In the following discussion, we refer to the matrix T^{-1} as the matrix Q.

TABLE 12.1 Algorithm for Data Distribution and Loop Transformations

Step 0. If a row in the reference matrix of all the arrays is the same, then no communication is involved. The data can be distributed along the respective dimension and all the data for the computation will be in local memory (initialize $i \leftarrow 1$).

Step 1. Distribute lhs array along dimension i (i.e., set $\vec{c}_i.[T^{-1}] = [1\ 0\ \ldots\ 0]$, where $\vec{c}_i$ represents row i of the lhs array C).

Step 2. Choose a rhs array that does not have a row in the reference matrix the same as that of an lhs array. For each row j in turn, set: $\vec{a}^p_j.[T^{-1}] = [0\ 0\ \ldots\ 0\ 1]$ for a reference to that array and $\vec{a}^p_{k \neq j} \cdot \vec{q}_n = 0$, where $\vec{a}^p_j$ represents row j in the data reference matrix for the pth rhs array A, and $\vec{q}_n$ is the nth column of T^{-1}.

If a valid T^{-1} is found, check the determinant of T^{-1}. If nonzero block transfers are possible for that rhs array, (break) go to step 3.

If there are no valid T^{-1} or the determinant of T^{-1} is zero, block transfers are not possible for dimension j on that array with the given distribution of the lhs array; therefore, increment j and go to step 2.

Step 3. Repeat step 2 for all the reference matrices of a particular array to check the results for that particular value of j.

Step 4. Repeat step 2 for all distinct arrays on rhs (increment p).

Step 5. Check the number of arrays where block transfers are possible.

Step 6. Repeat step 1 to step 4 for the lhs array distributed along each of the other dimensions in turn (increment i).

Step 7. Compare the number of arrays that can have block transfers and distribute the lhs array along the dimension that yields maximum number of block transfers for the arrays on the rhs.

Example 12.18
Matrix Multiplication

```
DO $i = 1, N$
  DO $j = 1, N$
    DO $k = 1, N$
      $C[i, j] = C[i, j] + A[i, k] * B[k, j]$
    ENDDO
  ENDDO
ENDDO
```

The reference matrices of the arrays are:

$$C_{\mathcal{R}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}, A_{\mathcal{R}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \text{and } B_{\mathcal{R}} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}.$$

Step 1. C is distributed along first dimension. Set:

$$\vec{C}_{\mathcal{R}}[1, :] \cdot \vec{Q}[:, 1] = 1$$
$$\vec{C}_{\mathcal{R}}[1, :] \cdot \vec{Q}[:, 2] = 0$$
$$\vec{C}_{\mathcal{R}}[1, :] \cdot \vec{Q}[:, 3] = 0$$

Therefore, we have, $q_{11} = 1, q_{12} = 0$ and $q_{13} = 0$.

Step 1a. Derive distribution of array A. Because row 1 of A is the same as that of C (i.e., $\vec{C}_{\mathcal{R}}[1, :] = \vec{A}_{\mathcal{R}}[1, :]$), distribute A and C identically.

Step 2.1. Derive distribution for array B. Check if you can find a matrix, $B_{\mathcal{R}}Q$, of the form:

$$B_{\mathcal{R}}Q = \begin{bmatrix} 0 & 0 & 1 \\ ? & ? & 0 \end{bmatrix}$$

where ? denotes entries we do not care about. Set:

$$\vec{B}_{\mathcal{R}}[1,:] \cdot \vec{Q}[:,1] = 0$$
$$\vec{B}_{\mathcal{R}}[1,:] \cdot \vec{Q}[:,2] = 0$$
$$\vec{B}_{\mathcal{R}}[1,:] \cdot \vec{Q}[:,3] = 1$$

Therefore, we have $q_{31} = 0$, $q_{32} = 0$ and $q_{33} = 1$. In addition, set $\vec{B}_{\mathcal{R}}[2,:] \cdot \vec{Q}[:,3] = 0$. This implies $q_{23} = 0$. Therefore, the first dimension of B is not distributed. Finally we have:

$$T^{-1} = Q = \begin{bmatrix} 1 & 0 & 0 \\ q_{21} & q_{22} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

For a unimodular transformation, $q_{22} = \pm 1$. Note that the dependence vector is [0 0 1], and therefore, no constraints are on q_{21}. This results in the identity matrix as the transformation matrix, and thus nothing needs to be done. Distribute A and C by rows, and B by columns. The code shown next gives the best performance we can get in terms of parallelism and locality. Note that the communication occurs outside the innermost loop. In this way a coarser grain in the communication pattern is achieved by vectorizing the messages. The loop follows:

```
DO $u = 1, N$
    DO $v = 1, N$
        send $B[*, u]$
        receive $B[*, v]$
        DO $w = 1, N$
            $C[u, v] = C[u, v] + A[u, w] * B[w, v]$
        ENDDO
    ENDDO
ENDDO
```

We go ahead and complete the algorithm by looking at distributing the lhs array in the next dimension.

Step 1.1. C is distributed along the second dimension. Set:

$$\vec{C}_{\mathcal{R}}[2,:] \cdot \vec{Q}[:,1] = 1$$
$$\vec{C}_{\mathcal{R}}[2,:] \cdot \vec{Q}[:,2] = 0$$
$$\vec{C}_{\mathcal{R}}[2,:] \cdot \vec{Q}[:,3] = 0$$

Therefore, we have $q_{21} = 1$, $q_{22} = 0$ and $q_{23} = 0$.

Step 1.1a. Derive distribution for array B. Because the second row of $B_{\mathcal{R}}$ is the same as the second row of $C_{\mathcal{R}}$ distribute B the same as C.

Step 2.2. Derive distribution for array A. Check whether you can find a matrix, $A_{\mathcal{R}}Q$, of the form:

$$A_{\mathcal{R}}Q = \begin{bmatrix} 0 & 0 & 1 \\ ? & ? & 0 \end{bmatrix}$$

where ? denotes entries we do not care about. Set:

$$\begin{aligned} \vec{A}_{\mathcal{R}}[1,:] \cdot \vec{Q}[:,1] &= 0 \\ \vec{A}_{\mathcal{R}}[1,:] \cdot \vec{Q}[:,2] &= 0 \\ \vec{A}_{\mathcal{R}}[1,:] \cdot \vec{Q}[:,3] &= 1 \end{aligned}$$

Therefore we have $q_{11} = 0$, $q_{12} = 0$ and $q_{13} = 1$; and $\vec{A}_{\mathcal{R}}[2,:] \cdot \vec{Q}[:,3] = 0 \Longrightarrow q_{33} = 0$. Finally, we have:

$$T^{-1} = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ q_{31} & q_{32} & 0 \end{bmatrix}$$

For a unimodular transformation, $q_{32} = \pm 1$. Therefore:

$$T^{-1} = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \text{ and } T = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

Distribute arrays A, B and C by columns. The transformed loop is given as follows:

```
DO $u = 1, N$
    DO $v = 1, N$
        send $A[*, u]$
        receive $A[*, v]$
        DO $w = 1, N$
            $C[w, u] = C[w, v] + A[w, v] * B[v, u]$
        ENDDO
    ENDDO
ENDDO
```

We see that the performance of the loop is similar in both cases. Therefore, array C can be distributed either by columns with the preceding transformation, or by rows with no transformation for the same performance with respect to communication. Again notice that the communication is carried outside the innermost loop.

Consider the symmetric rank 2K (SYR2K) code, from the basic linear algebra subroutines (BLAS) [57] example shown as follows:

Example 12.19 SYR2K

```
DO $i = 1, N$
  DO $j = i, \min(i + 2b - 2, N)$
    DO $k = \max(i - b + 1, j - b + 1, 1), \min(i + b - 1, j + b - 1, N)$
      $C[i, j - i + 1] = C[i, j - i + 1] + A[k, i - k + b] * B[k, j - k + b]$
                        $+ A[k, j - k + b] * B[k, i - k + b]$
    ENDDO
  ENDDO
ENDDO
```

The reference matrices for the arrays are:

$$C_{\mathcal{R}} = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \end{bmatrix}, A^1_{\mathcal{R}} = B^2_{\mathcal{R}} = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & -1 \end{bmatrix}, \text{and } A^2_{\mathcal{R}} = B^1_{\mathcal{R}} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & -1 \end{bmatrix}$$

Step 1. C row distributed. Set:

$$\vec{C}_{\mathcal{R}}[1, :] \cdot \vec{Q}[:, 1] = 1$$
$$\vec{C}_{\mathcal{R}}[1, :] \cdot \vec{Q}[:, 2] = 0$$
$$\vec{C}_{\mathcal{R}}[1, :] \cdot \vec{Q}[:, 3] = 0$$

Therefore we have, $q_{11} = 1$, $q_{12} = 0$ and $q_{13} = 0$. None of the other references matrices have any row common with $C_{\mathcal{R}}$.

Step 1.1. Derive distribution of A for the first reference; check whether the first dimension of A can not be distributed. Check whether you can find a matrix, $A^1_{\mathcal{R}}Q$, of the form:

$$A^1_{\mathcal{R}}Q = \begin{bmatrix} 0 & 0 & 1 \\ ? & ? & 0 \end{bmatrix}$$

where ? denotes entries we do not care about. Set:

$$\vec{A}^1_{\mathcal{R}}[1, :] \cdot \vec{Q}[:, 1] = 0$$
$$\vec{A}^1_{\mathcal{R}}[1, :] \cdot \vec{Q}[:, 2] = 0$$
$$\vec{A}^1_{\mathcal{R}}[1, :] \cdot \vec{Q}[:, 3] = 1$$

Therefore, $q_{31} = 0$, $q_{32} = 0$ and $q_{33} = 1$. In addition, $\vec{A}^1_{\mathcal{R}}[2, :] \cdot \vec{Q}[:, 3] = 0$ implies $q_{13} - q_{33} = 0$, which is impossible. Therefore, the first dimension of A has to be distributed.

Step 1.2. Derive distribution of A using first reference; check whether a second dimension of A cannot be distributed. Check whether you can find a matrix, $A^1_{\mathcal{R}}Q$, of the form:

$$A^1_{\mathcal{R}}Q = \begin{bmatrix} ? & ? & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

where ? denotes entries we do not care about. Set:

$$\vec{A}^1_{\mathcal{R}}[2,:] \cdot \vec{Q}[:,1] = 0$$
$$\vec{A}^1_{\mathcal{R}}[2,:] \cdot \vec{Q}[:,2] = 0$$
$$\vec{A}^1_{\mathcal{R}}[2,:] \cdot \vec{Q}[:,3] = 1$$

and $\vec{A}^1_{\mathcal{R}}[1,:] \cdot \vec{Q}[:,3] = 0$. Therefore, $q_{11} - q_{31} = 0 \Longrightarrow q_{13} = 1$; $q_{12} - q_{32} = 0 \Longrightarrow q_{32} = 0$; and $q_{13} - q_{33} = 1 \Longrightarrow q_{33} = -1$, which is impossible because $q_{33} = 1$. Thus, the second dimension of A also has to be distributed. Based on an analysis of the first reference of A, every dimension of A must be distributed. A similar result follows from an analysis of the second reference to A as well. Because the reference matrix for array A and B are the same, no block transfers for B can take place as well.

Step 2.0. C column is distributed. Set:

$$\vec{C}_{\mathcal{R}}[2,:] \cdot \vec{Q}[:,1] = 1$$
$$\vec{C}_{\mathcal{R}}[2,:] \cdot \vec{Q}[:,2] = 0$$
$$\vec{C}_{\mathcal{R}}[2,:] \cdot \vec{Q}[:,3] = 0$$

Therefore:

$$-q_{11} + q_{21} = 1 \Longrightarrow q_{11} = q_{21} - 1$$
$$-q_{12} + q_{22} = 0 \Longrightarrow q_{12} = q_{22}$$

and:

$$-q_{13} + q_{23} = 0 \Longrightarrow q_{13} = q_{23}$$

Step 2.1a. Derive distribution of A; check whether the first dimension of A cannot be distributed. Set:

$$\vec{A}^1_{\mathcal{R}}[1,:] \cdot \vec{Q}[:,1] = 0$$
$$\vec{A}^1_{\mathcal{R}}[1,:] \cdot \vec{Q}[:,2] = 0$$
$$\vec{A}^1_{\mathcal{R}}[1,:] \cdot \vec{Q}[:,3] = 1$$

Therefore, $q_{31} = 0$, $q_{32} = 0$ and $q_{33} = 1$, and:

$$\vec{A}^1_{\mathcal{R}}[2,:] \cdot \vec{Q}[:,3] = 0 \Longrightarrow q_{13} - q_{33} = 0 \Longrightarrow q_{13} = 1 \text{ and } q_{23} = 1$$

This means that under a column distribution of array C, the first reference to array A (i.e., $A^1_{\mathcal{R}}$) allows A not to be distributed along its first dimension. We now check whether the same result can be obtained with the second reference to array A (i.e., $A^2_{\mathcal{R}}$).

Step 2.1b. For second reference of A, check whether the second reference allows the first dimension of A not to be distributed. Set:

$$\vec{A}^2_{\mathcal{R}}[1,:] \cdot \vec{Q}[:,1] = 0$$
$$\vec{A}^2_{\mathcal{R}}[1,:] \cdot \vec{Q}[:,2] = 0$$
$$\vec{A}^2_{\mathcal{R}}[1,:] \cdot \vec{Q}[:,3] = 1$$

Therefore, $q_{31} = 0$, $q_{32} = 0$ and $q_{33} = 1$:

$$\vec{A}^2_{\mathcal{R}}[2,:] \cdot \vec{Q}[:,3] = 0 \Longrightarrow q_{23} - q_{33} = 0$$

and $q_{23} = 1$.

Thus, both references to A allow A not to be distributed by its first dimension. Thus, A can be column distributed (by its second dimension). Because B has identical array reference matrices those of A, array B can also be distributed by columns. Recall, that we started out with a column distribution of C. Thus, we have the inverse of the transformation matrix as:

$$T^{-1} = \begin{bmatrix} q_{11} & q_{12} & 1 \\ q_{21} & q_{22} & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

The only constraint on the unknown elements is that the resulting matrix be legal and unimodular.

Thus, we choose the unknown values such that T is a legal unimodular transformation. A possible T^{-1} is shown as follows:

$$T^{-1} = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \Longrightarrow T = \begin{bmatrix} -1 & 1 & 0 \\ 1 & 0 & -1 \\ 0 & 0 & 1 \end{bmatrix}$$

The transformed reference matrices are as follows:

$$C'_{\mathcal{R}} = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \end{bmatrix}, A'^{,1}_{R} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

and

$$A'^{,2}_{\mathcal{R}} = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}$$

By using the preceding algorithm we distribute arrays A, B and C by columns. In this way we have communication arising from A and B. Because we are using the owner-computes rule, the accesses to C are all local. We can thus move the communication outside the innermost loop. The transformed code with block transfers is as shown as follows:

```
DO $u = \max(0, 2 - 2b), \min(N - 1, 2b - 2)$
    DO $v = \max(1 - N, 1 - b), \min(N - 1, b - 1 - u)$
        send $A[*, u], B[*, u]$
        receive $A[*, v + b], A[*, u + v + b], B[*, u + v + b], B[*, v + b]$
        DO $w = \max(1, 1 - v), \min(N - u - v, N)$
            $C[v + w, u + 1] = C[v + w, u + 1] + A[w, v + b] * B[w, u + v + b]$
                        $+ A[w, u + v + b] * B[w, v + b]$
        ENDDO
    ENDDO
ENDDO
```

12.5.4 Relaxing the Owner-Computes Rule

Thus far we have relied on the use of the owner-computes rule and have assumed the processor that owns the lhs element of the assignment statement is the one that performs the computation. There are cases, though, in which using the owner-computes rule cannot allow block transfers. When this happens, we can try the algorithm by relaxing the owner-computes rule.

If we apply the method presented earlier to the code shown in Example 12.20, we find that whether array C is distributed by rows or by columns both arrays A and B must be distributed in all their dimensions and thus no block transfers are possible. The code shown in Example 12.20 is a variation of the code we have already seen in Example 12.19. Consider the following code:

Example 12.20

```
DO $i = 1, N$
    DO $j = 1, N$
        DO $k = 1, N$
            $C[i, j] = C[i, j] + A[k, i - k + b] * B[k, j - k + b]$
                $+ A[k, j - k + b] * B[k, i - k + b]$
        ENDDO
    ENDDO
ENDDO
```

By relaxing the owner-computes rule and modifying the algorithm accordingly we find that block transfers are indeed possible. We could distribute arrays A and B by rows and array C by columns and obtain the following code:

```
DO $u = 1, N$
    DO $v = 1, N$
        DO $w = 1, N$
            $tmp[w] = tmp[w] + A[u, v - u + b] * B[u, w - u + b]$
                $+ A[u, w - u + b] * B[u, v - u + b]$
        ENDDO
        send $tmp[*]$
        receive $C[*, u]$
    ENDDO
ENDDO
```

where tmp is a temporary column vector used to store the column of C that is computed locally. This same column storage is used each time the processor needs to compute a column of C. Another alternative is to distribute arrays A, B and C by rows instead and use the code shown next:

```
DO $u = 1, N$
    DO $v = 1, N$
        DO $w = 1, N$
            $tmp[w] = tmp[w] + A[u, w - u + b] * B[u, v - u + b]$
                $+ A[u, v - u + b] * B[u, w - u + b]$
            ENDDO
            send $tmp[*]$
            receive $C[u, *]$
        ENDDO
    ENDDO
```

where tmp is a temporary row vector used to store the row of C that is computed locally. This same row storage is used each time the processor needs to compute a row of C.

We notice that when the algorithm presented previously could not find a solution that would allow block transfers we could then, by relaxing the owner-computes rule, allow block transfers by allowing some other processor to perform the computation.

12.5.5 Generalized Algorithm

To explain the algorithm in Table 12.2 we use it to obtain the solution for the following problem. This is the code from Example 12.19:

```
DO $i = 1, N$
  DO $j = 1, N$
    DO $k = 1, N$
      $C[i, j] = C[i, j] + A[k, i - k + b] * B[k, j - k + b]$
                $+ A[k, j - k + b] * B[k, i - k + b]$
    ENDDO
  ENDDO
ENDDO
```

Notice that the accesses to the two-dimensional arrays A, B and C are such that although C is accessed along its second dimension, it is the first dimension of arrays A and B that is accessed. In other words, this is an example of communication along distinct axes. Note that the alignment phase

TABLE 12.2 Generalized Algorithm for Data Distribution and Loop Transformations

Step 0 through Step 7. These steps are the same as in Table 12.1.

Step 8. If no block transfers are possible, then initialize i to 1.

Step 9. Choose an rhs array and distribute it along dimension i. This array is now the base array.

Step 10. Choose an array that does not have a row in the reference matrix the same as that of the base array. For each row j in turn, set: $\vec{b}_j^p \cdot [T^{-1}] = [0\ 0\ \ldots\ 0\ 1]$ for a reference to that array and $\vec{b}_{k \neq j}^p \cdot \vec{q}_n = 0$.

If a valid T^{-1} is found, check the determinant of T^{-1}. If nonzero block transfers are possible for that array, (break) go to step 11.

If there are no valid T^{-1} or the determinant of T^{-1} is zero, block transfers are not possible for dimension j on that array with the given distribution of the base array; therefore, increment j and go to step 10.

Step 11. Repeat step 10 for all the reference matrices of a particular array to check the results for that particular value of j.

Step 12. Repeat step 10 for all distinct arrays if necessary (increment p).

Step 13. If no block transfers are possible, then increment i and repeat step 10.

Step 14. If block transfers are possible, then stop. Otherwise, initialize i to 1, repeat step 10 for a new rhs base array and stop when a solution is found or there are no more rhs arrays to be chosen as base arrays.

is not able to eliminate the interprocessor communication in this case because the index variables that are used for the accesses along the dimensions are different. In other words, the first dimension of A and B is indexed with a variable distinct to the variable used for the accesses along the second dimension of array C.

We identify the reference matrices shown as follows:

$$C_{\mathcal{R}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}, \; A^1_{\mathcal{R}} = B^2_{\mathcal{R}} = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & -1 \end{bmatrix}, \; A^2_{\mathcal{R}} = B^1_{\mathcal{R}} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & -1 \end{bmatrix}$$

The steps resulting from applying the algorithm to the preceding problem are presented in what follows:

1. C distributed along its first dimension. Set:

$$\vec{C}_{\mathcal{R}}[1,:] \cdot \vec{Q}[:,1] = 1$$

$$\vec{C}_{\mathcal{R}}[1,:] \cdot \vec{Q}[:,2] = 0$$

$$\vec{C}_{\mathcal{R}}[1,:] \cdot \vec{Q}[:,3] = 0$$

from which we obtain, $q_{11} = 1, \; q_{12} = 0$ and $q_{13} = 0$. Note that no rows are in any of the other reference matrices that are the same as any of the rows of $C_{\mathcal{R}}$. Otherwise, we could determine at this point which of the remaining arrays could be distributed using the same distribution that we have for C.

a. Derive distribution for the first dimension of array A using the first reference matrix of A (i.e., $A^1_{\mathcal{R}}$ by checking whether the first dimension of A cannot be distributed). In other words, check whether we can find a matrix of the form:

$$A^1_{\mathcal{R}} Q = \begin{bmatrix} 0 & 0 & 1 \\ ? & ? & 0 \end{bmatrix}$$

Set:

$$\vec{A}^1_{\mathcal{R}}[1,:] \cdot \vec{Q}[:,1] = 0$$

$$\vec{A}^1_{\mathcal{R}}[1,:] \cdot \vec{Q}[:,2] = 0$$

$$\vec{A}^1_{\mathcal{R}}[1,:] \cdot \vec{Q}[:,3] = 1$$

to obtain $q_{31} = 0, q_{32} = 0$ and $q_{33} = 1$. To satisfy the requirement that the innermost loop index variable must not appear in the second dimension, also set $\vec{A}^2_{\mathcal{R}}[1,:] \cdot \vec{Q}[:,3] = 0$ from which we obtain $q_{13} = q_{33} = 0$ that is a contradiction to the preceding finding stating $q_{33} = 1$. Therefore, the first dimension of A must be distributed and thus we cannot perform block transfers for A along its first dimension. Because $B^2_{\mathcal{R}} = A^1_{\mathcal{R}}$ this means that we also cannot perform block transfers for B along its first dimension. This is all assuming that C is distributed along its first dimension.

b. Derive distribution for the second dimension of array A using the first reference matrix of A (i.e., $A^1_{\mathcal{R}}$ by checking whether the second dimension of A cannot be distributed). In other words, check whether we can find a matrix of the form:

$$A^1_{\mathcal{R}} Q = \begin{bmatrix} ? & ? & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Set:

$$\vec{A}^1_{\mathcal{R}}[2,:] \cdot \vec{Q}[:,1] = 0$$

$$\vec{A}^1_{\mathcal{R}}[2,:] \cdot \vec{Q}[:,2] = 0$$

$$\vec{A}^1_{\mathcal{R}}[2,:] \cdot \vec{Q}[:,3] = 1$$

Therefore, $q_{11} = q_{31} = 0, q_{12} = q_{32} = 0$ and $q_{13} = q_{33} = 1$. Now set $\vec{A}^1_{\mathcal{R}}[1,:] \cdot \vec{Q}[:,3] = 0$, that is, $q_{33} = 0$, which is again a contradiction to the preceding finding stating $q_{33} = 1$. Thus, the second dimension of A must be distributed and we cannot perform block transfer for A along its second dimension. Because $B^2_{\mathcal{R}} = A^1_{\mathcal{R}}$, this means that we also cannot perform block transfers for B along its second dimension. Remember that this analysis has been made assuming that the lhs array C is distributed along its first dimension.

2. C is distributed along its second dimension. Set:

$$\vec{C}_{\mathcal{R}}[2,:] \cdot \vec{Q}[:,1] = 1$$

$$\vec{C}_{\mathcal{R}}[2,:] \cdot \vec{Q}[:,2] = 0$$

$$\vec{C}_{\mathcal{R}}[2,:] \cdot \vec{Q}[:,3] = 0$$

Therefore $q_{21} = 1$, $q_{22} = 0$ and $q_{23} = 0$. Again no rows occur in any of the other reference matrices that are the same to any of the rows of $C_{\mathcal{R}}$.

a. Derive distribution for the first dimension of array A using the first reference matrix of A (i.e., $A^1_{\mathcal{R}}$) by checking if the first dimension of A cannot be distributed. In other words, check whether we can find a matrix of the form:

$$A^1_{\mathcal{R}} Q = \begin{bmatrix} 0 & 0 & 1 \\ ? & ? & 0 \end{bmatrix}$$

Set:

$$\vec{A}^1_{\mathcal{R}}[1,:] \cdot \vec{Q}[:,1] = 0$$

$$\vec{A}^1_{\mathcal{R}}[1,:] \cdot \vec{Q}[:,2] = 0$$

$$\vec{A}^1_{\mathcal{R}}[1,:] \cdot \vec{Q}[:,3] = 1$$

to obtain $q_{31} = 0$, $q_{32} = 0$ and $q_{33} = 1$. Also set $\vec{A}^2_{\mathcal{R}}[1,:] \cdot \vec{Q}[:,3] = 0$, which yields $q_{13} = q_{33} = 0$, a contradiction. Therefore, the first dimension of A must be distributed. This means that we cannot perform block transfers along the first dimension of A. Note that $B^2_{\mathcal{R}} = A^1_{\mathcal{R}}$ and thus we cannot perform block transfers along the first dimension of B if C is distributed along its second dimension.

b. Derive distribution for the second dimension of array A using the first reference matrix of A (i.e., $A^1_{\mathcal{R}}$ by checking whether the second dimension of A cannot be distributed). In other words, check whether we can find a matrix of the form:

$$A^1_{\mathcal{R}} Q = \begin{bmatrix} ? & ? & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Set:

$$\vec{A}^1_{\mathcal{R}}[2,:] \cdot \vec{Q}[:,1] = 0$$
$$\vec{A}^1_{\mathcal{R}}[2,:] \cdot \vec{Q}[:,2] = 0$$
$$\vec{A}^1_{\mathcal{R}}[2,:] \cdot \vec{Q}[:,3] = 1$$

Therefore, $q_{11} = q_{31} = 0, q_{12} = q_{32} = 0$ and $q_{13} = q_{33}+1$. Now set $\vec{A}^1_{\mathcal{R}}[1,:]\cdot\vec{Q}[:,3] = 0$, which yields $q_{33} = 0$. Thus, $q_{13} = 1$. This results in:

$$Q = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

which is not unimodular. Therefore, the second dimension of A must also be distributed. This means that we cannot perform block transfers along the second dimension of A and, as before, because $B^2_{\mathcal{R}} = A^1_{\mathcal{R}}$, we also cannot perform block transfers along the second dimension of B. Therefore, using the owner-computes rule does not allow block transfers for either array A or array B.

3. At this time we relax the owner-computes rule and allow the owner of a rhs array to be the one performing the computation. A is distributed along its first dimension. Set:

$$\vec{A}^1_{\mathcal{R}}[1,:] \cdot \vec{Q}[:,1] = 1$$
$$\vec{A}^1_{\mathcal{R}}[1,:] \cdot \vec{Q}[:,2] = 0$$
$$\vec{A}^1_{\mathcal{R}}[1,:] \cdot \vec{Q}[:,3] = 0$$

from which we obtain, $q_{31} = 1$, $q_{32} = 0$ and $q_{33} = 0$. Note that the first row of $A^2_{\mathcal{R}}$, $B^1_{\mathcal{R}}$ and $B^2_{\mathcal{R}}$ is identical to the first row of $A^1_{\mathcal{R}}$. Therefore, both arrays A and B can be distributed by rows.

a. Derive distribution for the first dimension of array C to check whether it cannot be distributed. Check whether we can find a matrix of the form:

$$C_{\mathcal{R}}Q = \begin{bmatrix} 0 & 0 & 1 \\ ? & ? & 0 \end{bmatrix}$$

Set:

$$\vec{C}_{\mathcal{R}}[1,:] \cdot \vec{Q}[:,1] = 0$$
$$\vec{C}_{\mathcal{R}}[1,:] \cdot \vec{Q}[:,2] = 0$$
$$\vec{C}_{\mathcal{R}}[1,:] \cdot \vec{Q}[:,3] = 1$$

which results in $q_{11} = 0$, $q_{12} = 0$ and $q_{13} = 1$. Now set:

$$\vec{C}_{\mathcal{R}}[2,:] \cdot \vec{Q}[:,3] = 0$$

which yields $q_{23} = 0$. This means that:

$$Q = \begin{bmatrix} 0 & 0 & 1 \\ q_{21} & q_{22} & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

which is unimodular if we choose $q_{22} = \pm 1$, that is:

$$Q = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

Therefore, by distributing A, B by rows and C by columns and using the preceding transformation, we can transform the code to:

```
DO $u = 1, N$
  DO $v = 1, N$
    DO $w = 1, N$
      $tmp[w] = tmp[w] + A[u, w - u + b] * B[u, v - u + b]$
              $+ A[u, v - u + b] * B[u, w - u + b]$
    ENDDO
    send $tmp[*]$
    receive $C[*, u]$
  ENDDO
ENDDO
```

Using tiling to obtain higher granularity communication — It would be advantageous to continue to increase the granularity of the communication between processors. One way to accomplish this is by tiling one or more dimensions of the iteration space. Tiling is a well-known technique used to assign blocks of iterations, instead of just one at a time, to the available processors [6, 7, 65, 81]. Some of the key issues involved in tiling include the choice of tile sizes. A detailed discussion of these is beyond the scope of this chapter.

Comparison with the work of Li and Pingali — Li and Pingali [57] used specified data distributions and developed a systematic loop transformation strategy, identified by them as *access normalization*, which restructures loop nests to exploit locality and block transfers whenever possible. Li and Pingali [56] discussed the completion of partial transformations derived from the data access matrix of a loop nest; the rows of the data access matrix are subscript functions for various array accesses (excluding constant offsets). Their work assumes that all arrays are distributed by columns.

12.6 Brief Review of Other Work on Data Mapping

Research on problems related to memory optimizations goes back to studies of the organization of data for paged memory systems [1]. Balasundaram and others [17] have developed interactive parallelization tools for multicomputers that provide the user with feedback on the interplay between data decomposition and task partitioning on the performance of programs. Gallivan et al. [28] discussed problems associated with automatically restructuring data so that it can be moved to and from local memories in the case of shared memory machines with complex memory hierarchies. They present a series of theorems that enable one to describe the structure of disjoint sublattices accessed by different processors, use this information to make “correct” copies of data in local memories and write the data back to the shared address space when the modifications are complete. Gannon, Jalby and Gallivan [29] discussed program transformations for effective complex-memory management for a CEDAR-like architecture with a three-level memory. Gupta and Banerjee [34] present a constraint-based system to automatically select data decompositions for loop nests in a program. Hudak and Abraham [41] discussed the generation of rectangular and hexagonal partitions of arrays accessed in sequentially iterated parallel loops. Knobe, Lukas and Steele [47] discussed techniques for automatic layout of arrays in a compiler targeted to SIMD architectures such as the Connection Machine computer system. Li and Chen [54] (and Chen, Choo and Li [24]) have addressed the problem of index domain alignment, which is that of finding a set of suitable alignment functions mapping the index domains of the arrays into a common index domain to minimize the communication cost incurred due to data movement. The class of alignment functions that they consider primarily are permutations and embeddings. The kind of alignment functions that we deal with are more general than these. Mace [60] proved that the problem of finding optimal data storage patterns for parallel processing (the shapes problem) is NP-complete, even when limited to 1- and 2-dimensional arrays; in addition, efficient algorithms are derived for the shapes problem for programs modeled by a directed acyclic graph (DAG) that is derived by series–parallel combinations of treelike subgraphs. Wang and Gannon [75] presented a heuristic state–space search method for optimizing programs for memory hierarchies.

In addition, several researchers have developed compilers that take a sequential program augmented with annotations that specify data distribution, and generate the necessary communication. Koelbel, Mehrotra and van Rosendale [48, 49] address the problem of automatic process partitioning of programs written in a functional language called BLAZE given a user-specified data partition. A group led by Kennedy at Rice University [18] is studying similar techniques for compiling a version of FORTRAN for local memory machines, which includes annotations for specifying data decomposition. They show how some existing transformations could be used to improve performance. Rogers and Pingali [68] present a method that given a sequential program and its data partition performs task partitions to enhance locality of references. Zima, Bast and Gerndt [87] have developed SUPERB, an interactive system for semiautomatic transformation of FORTRAN programs into parallel programs for the SUPRENUM machine, a loosely coupled hierarchical multiprocessor.

Next, we discuss in greater detail the works of Huang and Sadayappan [40], Gilbert and Schreiber [32], and Gupta and Banerjee [34]. After this we discuss other work on data mapping.

Huang and Sadayappan [40] focus on partitions of iterations and data arrays that eliminate data communication and consider partitions of iteration and data spaces along sets of hyperplanes. Because data elements are not to be accessed by different processors, even read-only data cannot be shared. All iterations belonging to an iteration hyperplane and all the data belonging to a data hyperplane are assigned to one processor, thus, the owner-computes rule is implicit. A processor executes iterations from the iteration hyperplanes that are assigned to it and in so doing it accesses data from its assigned data hyperplanes. Their article presents no way of dealing with cases when communication-free partitioning, while maintaining parallelism, is not possible. It begins by presenting solutions for a single hyperplane partitioning for each iteration and data space and moves on to multiple (double) hyperplanes per space at which time they propose a heuristic. Huang and Sadayappan [40] derive necessary and sufficient conditions for communication-free hyperplane partitioning of both data and computation for fully parallel loop nests in the absence of flow and antidependences. Flow and antidependences are treated elsewhere and a list of articles that treat this subject is given later in this work. For communication-free single hyperplane partitioning of the iteration and data spaces the following must hold for an access function in the form of $A^i_{j,k}(I) + a^i_{j,k}$, which accesses the kth reference to the jth data array in the ith nested loop, where I is used to denote the iteration vector, H and G are row vectors containing the iteration and data hyperplane coefficients (which are rational numbers), respectively, and α is nonzero:

1. $G_{j1}A^i_{j1,k_1} = G_{j2}A^i_{j2,k_2}$
2. $G_j a^i_{j,k_1} = G_j a^i_{j,k_2}$

3. $H_i = \alpha^i_j G_j A^i_{j,1}$
4. $\alpha^{i1}_{j1}\alpha^{i2}_{j2} = \alpha^{i1}_{j2}\alpha^{i2}_{j1}$
5. $\alpha^{i1}_{j1} G_{j1}(a^{i1}_{j1,1} - a^{i2}_{j1,1}) = \alpha^{i1}_{j2} G_{j2}(a^{i1}_{j2,1} - a^{i2}_{j2,1})$

An example that captures the essence of their work for the case of multiple arrays and multiple references, with single hyperplane partitioning, is shown next for the same loop used previously, that is:

```
**DO** $i = lb_i, ub_i$
    **DO** $j = lb_j, ub_j$
        $A[i, j] = f(A[i, j], B[i - 1, j], B[i, j - 1])$
    **ENDDO**
**ENDDO**
```

Here we find:

$$A_1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, a_1 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, A_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, a_2 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$B_1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, b_1 = \begin{bmatrix} -1 \\ 0 \end{bmatrix}, B_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, b_2 = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$$

from which, by applying the conditions stated previously, we get:

$$G_A \begin{bmatrix} A_1 - A_2 & a_1 - a_2 \end{bmatrix} = \begin{bmatrix} g_{A1} & g_{A2} \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}$$

and:

$$G_A \begin{bmatrix} B_1 - B_2 & b_1 - b_2 \end{bmatrix} = \begin{bmatrix} g_{B1} & g_{B2} \end{bmatrix} \begin{bmatrix} 0 & 0 & -1 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}$$

From the last equation we find that $g_{B1} = g_{B2}$. The other set of equations is found from:

$$\begin{bmatrix} G_A & G_B \end{bmatrix} \begin{bmatrix} A_1 \\ -B_1 \end{bmatrix} = \begin{bmatrix} g_{A1} & g_{A2} & g_{B1} & g_{B2} \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ -1 & 0 \\ 0 & -1 \end{bmatrix} = \begin{bmatrix} 0 & 0 \end{bmatrix}$$

which yields $g_{A1} = g_{B1}$ and $g_{A2} = g_{B2}$. Therefore, we can choose:

$$G_A = G_B = \begin{bmatrix} 1 & 1 \end{bmatrix}$$

which is the same result we obtained using the method in Ramanujam and Sadayappan [64]. Additionally:

$$H = \alpha_A G_A A_1 = \begin{bmatrix} 1 & 1 \end{bmatrix}$$

for $\alpha_A = 1$.

The work in Huang and Sadayappan [40] does not assume anything about the architecture of the machine, and implicitly assumes the owner-computes rule. As mentioned earlier, no attempt is made to deal with the problem when communication is unavoidable. The alignment obtained and the access functions allowed are more general than what is allowed in the current HPF standard [36].

Gilbert and Schreiber [32] propose a method that considers only one expression at a time. The minimum cost of computing an arbitrary expression is found for architectures with robustness (e.g., hypercubes, linear arrays and meshes) on which realistic metrics could be used. As Gilbert and Schreiber [32] explain, a given metric describes the cost of moving an array from one position to another within a machine. For example, the l_1 (Manhattan), l_2 (Euclidean), l_∞ and Hamming metrics are realistic for a 1-dimensional processor array, a grid of processors with connections to their nearest neighbors, a grid of processors with connections to their nearest neighbors and their diagonal neighbors, and for hypercubes, respectively. In the l_1 or Manhattan metric the distance d from a point x to a point y on a k-dimensional space is given by:

$$d(x, y) = \sum_i |x_i - y_i|$$
$$1 \le i \le k$$

whereas in the l_∞ metric we have:

$$d(x, y) = \max_i |x_i - y_i|$$
$$1 \le i \le k$$

The cost of the expression is evaluated by embedding its rooted binary tree onto the architecture and then finding the minimum cost of evaluating it using an specific metric. Subexpressions needed to evaluate an expression are in turn evaluated where doing so is cheapest (i.e., at the closest processors among a set of processors at which the evaluation of the subexpression is possible, to the processor that can evaluate the expression). The authors do not assume the owner-computes rule. As an example of what is presented by Gilbert and Schreiber [32] we have in Figure 12.11(a) four arrays to be combined in the expression $(w \oplus x) \otimes (y \odot z)$, where $\oplus$, $\otimes$ and $\odot$ are array operators. Each point in Figure 12.11 is a processor and so we could think of a grid of processors as the architecture that is used. In Figure 12.11(b) we have the result of applying this method to the preceding expression. Region A in Figure 12.11(b) represents the set of processors that should evaluate $w \oplus x$ (i.e., the set of processors for which the cost of evaluating $w \oplus x$ is minima). Similarly, region B represents the set of processors that should evaluate subexpression $y \odot z$, and region C represents the set of processors that should evaluate the final expression (i.e., the root). Assume that processor p in region C is chosen to evaluate the final expression among all processors that can evaluate it. Then the processor in A that is closest to processor p is chosen to evaluate $w \oplus x$, and the processor in B that is closest to processor p is chosen to evaluate $y \odot z$. They both send their partial results to p, which then evaluates the final expression.

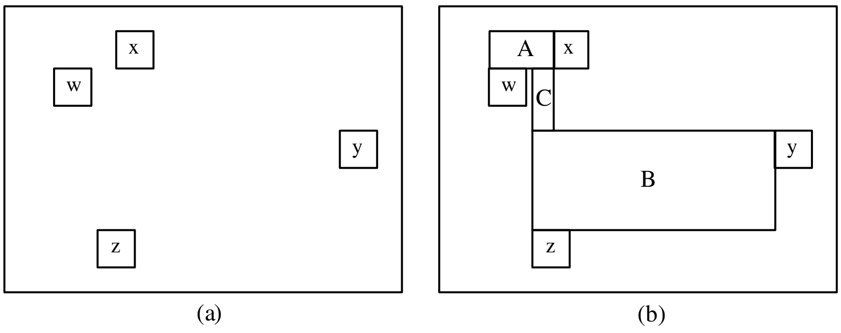


FIGURE 12.11 Example from Gilbert and Schreiber.

Gilbert and Schreiber [32] use an approach where the processors at which the expression under consideration, as well as its subexpressions, should be evaluated to minimize cost are found. The article deals with neither data nor computation decompositions. The work pertaining to a computation is performed by several processors. The work by Gilbert and Schreiber [32] is different to our work in that we consider all statements within a loop nest, instead of just one statement at a time. They relax the owner-computes rule and use the l_1-metric, which is in this context robust and realistic for a grid of processors with nearest neighbor connections. We are concerned with partitioning of both data and computation, vectorization of messages and mapping transformations to machine communication primitives. We also relax the owner-computes rule, but we assume that the work performed during one iteration is performed by only one processor.

Gupta and Banerjee [34] present a method restricted to partitioning of arrays (i.e., no computation partitioning). In their method Gupta and Banerjee select important segments of code to determine distribution of various arrays based on some constraints. Quality measures are used to choose among contradicting constraints. These quality measures may require user intervention. The compiler tries to combine constraints for each array in a consistent manner to minimize overall execution time and the entire program is considered. Small arrays are assumed to be replicated on all processors. The distribution of arrays is by rows, columns or blocks. This work uses heuristic algorithms to determine the alignment of dimensions (i.e., component alignment of various arrays because the problem has been shown to be NP-complete). The owner-computes rule is assumed and issues concerning the best way to communicate messages among processors are dealt with, as in the aggregate communication work introduced by Tseng [73]. Communication costs are determined by Gupta and Banerjee [34] after identifying the pairs of dimensions that should be aligned. Consideration is given to when it would be best to replicate a dimension instead of distribute it.

The algorithm builds the component affinity graph (CAG) developed by Li and Chen [54], as shown in Figure 12.12, and decides to align the first dimension of each of the arrays and also the second dimension because it would be too costly to do otherwise. That is, the cheapest way to partition the node set into $D = 2$ disjoint subsets is by grouping A_1 and B_1 into one subset, and A_2 and B_2 into another subset, where D is the dimensionality of the arrays. In this way the total weight of the edges going from one subset to the other is zero. The cost of choosing a cyclic distribution should make it favorable for the algorithm to choose a contiguous distribution for both dimensions. The alignment done is in terms of which dimensions should be aligned but it does not calculate how to best align them. The nodes of the CAG represent array dimensions. An edge is added between two nodes for every constraint in the alignment of two dimensions. The weight of the edge is equal to the quality measure of the constraint.

The work by Gupta and Banerjee [34] uses the owner-computes rule, requires user intervention and does not attempt to compute alignments beyond alignment of dimensions. In our work we address both data and computation alignment, relaxing the owner-computes rule. We address cases of axis

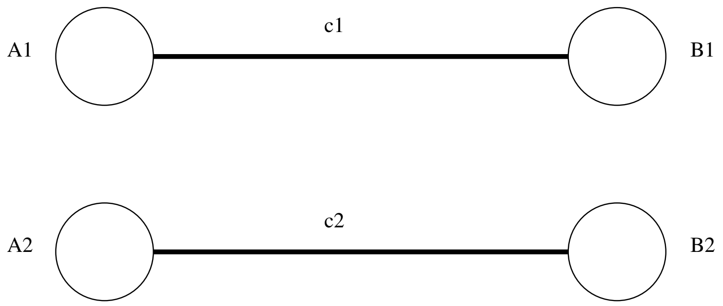


FIGURE 12.12 Component affinity graph (CAG) partitioned by classes of dimensions.

alignment, stride and reversal alignments and offset alignment. We do agree that small arrays, such as scalars, should be replicated; also, the communication should be optimized by moving it outside the innermost loop whenever possible.

Bau et al. [14] use elementary matrix methods to determine communication-free alignment of code and data. They also deal with the problem of replicating read-only data to eliminate communication. They incorporate data dependences in their proposed solution to the problem, but the owner-computes rule is assumed. Replication of data is also incorporated into their proposed solution.

Amarasinghe et al. [6] show how to find partitions for *doall* and *doacross* parallelism, and to minimize communication across loop nests they use a greedy algorithm that tries to avoid the largest amounts of potential communication. They give examples of how to obtain parallelism by incurring some communication when this is the only way to run in parallel.

Chatterjee et al. [19] and [20] provide an algorithm that obtains alignments that are more general than the owner-computes rule by decomposing alignment functions into several components. Chatterjee et al. [19] investigate the problem of evaluating FORTRAN 90 style array expressions on massively parallel DMMs. They present algorithms based on dynamic programming. A number of other researchers have also made contributions to this problem. Kim and Wolfe [46] show how to find and operate on the communication pattern matrix from user-aligned references. Our approach generates the alignment of data and computation and frees the user from this task. Li and Pingali [57] start with user specified data distributions and develop a systematic loop transformation strategy identified by them as access normalization, which restructures loop nests to exploit locality and block transfers whenever possible. Although we are also interested in maintaining locality, our approach and theirs are different. We develop the data and computation distributions based on our findings; the user does not have to specify them.

O'Boyle [61] proposed an automatic data partition algorithm based on the analysis of four distinct factors. We concur with him in his view that automatic data partitioning is possible and that it must be considered in the context of the whole compilation process instead of left to the programmer. He does not consider partitioning of computation along with that of data and he is not concerned with finding the alignment that minimizes communication, as we are in our work. Wakatani and Wolfe [74] address the problem of minimizing communication overhead but from a different context than ours. They are concerned with the communication arising from the redistribution of an array and proposed a technique called *strip mining redistribution*. They are not concerned with automatically generating the alignments, as we are, to free the programmer from this task and achieve minimum communication while preserving parallelism.

Chatterjee, Gilbert and Schreiber [22] and Sheffer et al. [71] deal with determining both static and dynamic distributions. They use the alignment–distribution graph (ADG) with nodes that represent program operations, the ports in the nodes represent array objects manipulated by the program and the edges connect array definitions to their respective uses. The ADG is a directed edge-weighed graph although it is used as an undirected graph. Communication occurs when the alignment or distribution at the end points of an edge is different. The completion time of a program is modeled as the sum of the cost over all the nodes (which accounts for computation and realignment) plus the sum over all the edges of the redistribution time (which takes into account the cost per data item of all-to-all personalized communication, the total data volume and the discrete distance between distributions).

The main effort of Ayguadé et al. [8] is directed toward intraprocedural data mappings. Candidate distributions are used to build a search space from which to determine, based on profitability analyses, the points at which to realign or redistribute the arrays to improve the performance by reducing the total data movement. The CAG of Li and Chen [54] is used to determine the best local distribution for a particular phase of the code. All the arrays in a phase are distributed identically. Control flow information is used for phase-sequencing identification. An intraprocedural remapping algorithm is provided.

Garcia, Ayguadé and Labarta [30] present an approach to automatically perform static distribution using a constraint based model on the communication-parallelism graph (CPG). The CPG contains edges representing both communication and parallelization constraints. The constraints are formulated and solved using a linear 0-1 integer programming model and solver. They obtain solution for one-dimensional array distributions (i.e., only one dimension of the arrays is distributed), and use an iterative approach for the multidimensional problem.

Kremer [53] proves the dynamic remapping problem NP-complete. Kremer et al. [52] and Kremer [51] consider the profitability of dynamic remapping and use an interactive tool for automatic data layout, respectively. Kennedy and Kremer [44, 45] deal with dynamic remapping in FORTRAN D [73] and HPF [36]. The work by Kennedy and Kremer proposes a way to solve the NP-complete interdimensional alignment problem [53] using a state-of-the-art, general-purpose integer programming solver [45]. Thus, Kennedy and Kremer [45] formulate the interdimensional alignment problem as a 0-1 integer programming problem. The same is done by Bixby, Kennedy and Kremer et al. [15].

Palermo and Banerjee [63] deal with dynamic partitioning by building the communication graph. In this graph the nodes correspond to statements in the program and the edges are flow dependences between the statements. The weight on these edges reflects communication. Maximal cuts are used to remove the largest communication constraints and recursively divide the graph or subgraphs until chunks of code (phases) that should share the same partitioning schemes are grouped together. Thus, remapping may be inserted between phases and not within a particular phase to reduce communication between phases.

In addition, relevant background in the areas of dependence analysis and program transformations is needed for understanding the material in Section 12.5. For the general theory of dependence analysis and vectorization, the reader is referred to Allen and Kennedy [3, 5]; Allen, Callahan and Kennedy [4]; Banerjee [10–12]; Banerjee et al. [13]; Blume and Eigenmann [16]; Cytron [25]; Goff, Kennedy and Tseng [33]; Irigoin and Triolet [42]; Maydan, Hennessy and Lan [58]; Padua and Wolfe [62]; Ramanujam and Sadayappan [65]; Wolf and Lam [76]; Wolfe [77–80, 82]; Wolfe and Banerjee [83]; Wolfe and Tseng [84]; and Zima and Chapman [85]. For cache and locality issues see Anderson and Lam [7]; Fang and Lu [26]; Gallivan, Jalby and Gannon [28]; and Gannon, Jalby and Gallivan [29]. In the articles by Chatterjee et al. [20, 21] and by Stichnoth [72] the reader can find an introduction to the issues related with the assignment of array elements to the local memory of processors and how these are accessed. Alignment is discussed in detail by Chatterjee et al. [20, 21] and Chatterjee, Gilbert and Schreiber [22]. HPF and related issues are covered in [36], Hiranandani, Kennedy and Tseng et al. [37, 38], and Hiranandani et al. [39]. Communication-free compiling is the main topic of Fang and Lu [26], Huang and Sadayappan [40] and Ramanujam and Sadayappan [64], whereas data flow analysis is treated in the book by Aho, Sethi and Ullman [2] and the article by Maydan, Amarasinghe and Lam [59]. Finally, compiling for DMM is the topic of Bal, Steiner and Tanenbaum [9]; Gupta et al. [35]; Hiranandani, Kennedy and Tseng [37, 38]; Tseng [73]; and Zima and Chapman [86].

12.7 Summary

In DMM, interprocessor communication is more time consuming than instruction execution. If insufficient attention is paid to the data allocation problem, then so much time may be spent in interprocessor communication that much of the benefit of parallelism is lost. It is therefore worthwhile for a compiler to analyze patterns of data usage to determine allocation for minimizing interprocessor communication. This chapter presents a detailed discussion of two techniques, the first for communication-free partitioning of arrays (Section12.4) and the second for deriving data distributions and associated loop transformations to minimize communication overhead in message-passing computers (Section12.5).

In Section12.4, we present a formulation of the problem of determining whether communication-free array partitions (decompositions) exist and present machine-independent sufficient conditions for the same for a class of parallel loops without flow or antidependences, where array references are affine functions of loop index variables. In addition, where communication-free decomposition is not possible, we have discussed a mathematical formulation that aids in minimizing communication.

Section12.5 develops an algorithm that derives the terms in the transformation matrix, which gives the best locality and minimum communication on DMM. We used the concept of data reference matrices for individual array references. By using this concept as the starting point, we systematically derive the best set of transformation matrices that give both good locality while enabling parallelism. Unlike [56], where a padding matrix is used along with an arbitrary set of rows in the basis matrix, we generate a transformation matrix systematically. The key idea in this method is to move communication out of the innermost loop so that messages can be vectorized, reducing the amount of communication by an order of magnitude. An algorithm is provided and several detailed examples are used to show the effectiveness of this systematic approach. This algorithm begins by assuming the owner-computes rule and relaxes it if no block transfers solution is achieved. The algorithm also gives an optimal distribution of arrays on to the processors such that block transfers are enabled to reduce interprocessor communication. Here, distribution of data only along one dimension is considered. However, complex distributions with more than one distributed dimension can be derived using a simple extension of this algorithm.

Data mapping in concert with program transformations remains a challenging problem for a wide variety of machine architectures, ranging from message-passing machines at one end to nonuniform memory access shared-memory machines at the other. The problem is also important for embedded systems that have limited amounts of memory. Progress in this area is important for the effective exploitation of a broad spectrum of machine architectures.

Acknowledgment

We gratefully acknowledge the support of the U.S. National Science Foundation through an NSF Young Investigator Award 9457768; and through NSF grants 9210422, 0073800 and 0121706.

References

[1] W. Abu-Sufah, D. Kuck and D. Lawrie, On the performance enhancement of paging systems through program analysis and transformations, *IEEE Trans. Comput,* C-30(5), 341–356, May 1981.

[2] A.V. Aho, R. Sethi and J.D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA, 1986.

[3] J. Allen and K. Kennedy, Automatic Loop Interchange, in Proceedings 1984 SIGPLAN Symposium on Compiler Construction, 19, 233–246, June 1984.

[4] J.R. Allen, D. Callahan and K. Kennedy, Automatic Decomposition of Scientific Programs for Parallel Execution, in Proceedings of the 14th Annual ACM Symposium on the Principles of Programming Languages, Munich, Germany, January 1987.

[5] J.R. Allen and K. Kennedy, Automatic translation of Fortran programs to vector form, *ACM Trans. on Programming Languages Syst.*, 9(4), October 1987, 491–542.

[6] S.P. Amarasinghe, J.M. Anderson, M.S. Lam and A.W. Lim, An Overview of a Compiler for Scalable Parallel Machines, in Proceedings of the 6th Annual Workshop on Languages and Compilers for Parallel Computing, Portland, OR, August 1993.

[7] J.M. Anderson and M.S. Lam, Global Optimizations for Parallelism and Locality on Scalable Parallel Machines, in Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation, Albuquerque, NM, June 1993, pp. 112–125.

[8] E. Ayguadé, J. Garcia, M. Gironès, M.L. Grande and J. Labarta, Data redistribution in an automatic data distribution tool, in *Proceedings of the 8th Annual Workshop on Languages and Compilers for Parallel Computing, Lecture Notes in Computer Science*, Springer-Verlag, 1995.

[9] H.E. Bal, J.G. Steiner and A.S. Tanenbaum, Programming languages for distributed computing systems, *ACM Comput. Surv.*, 21(3), September 1989.

[10] U. Banerjee, An introduction to a formal theory of dependence analysis, *J. Supercomputing*, 2(2), 133–149, October 1988.

[11] U. Banerjee, A theory of loop permutations, in *Languages and Compilers for Parallel Processing*, Research Monographs in Parallel and Distributed Computing, D. Gelernter, A. Nicolau and D. Padua, Eds., Pitman, London, 1990, pp. 54–74.

[12] U. Banerjee, Unimodular transformation of double loops, in *Advances in Languages and Compilers for Parallel Processing*, A. Nicolau, D. Gelernter, T. Gross and D. Padua, Eds., Pitman, London, 1991, pp. 192–219.

[13] U. Banerjee, R. Eigenmann, A. Nicolau and D. Padua, Automatic program parallelization. *Proc. of the IEEE*, 81(2), 211–243, February 1993.

[14] D. Bau, I. Kodukula, V. Kotlyar, K. Pingali and P. Stodghill, Solving alignment using elementary linear algebra, in *Languages and Compilers for Parallel Computing: Seventh International Workshop*, K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau and D. Padua, Eds., Springer-Verlag, 1994.

[15] R. Bixby, K. Kennedy and U. Kremer, Automatic Data Layout Using 0-1 Integer Programming, in Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, Montreal, Canada, August 1994, pp. 111–122.

[16] W. Blume and R. Eigenmann, Performance analysis of parallelizing compilers on the perfect benchmarks programs, *IEEE Trans. Parallel Distributed Syst.*, 3(6), 643–656, November 1992.

[17] V. Balasundaram, G. Fox, K. Kennedy and U. Kremer, An Interactive Environment for data partitioning and Distribution, Proceedings 5th Distributed Memory Computing Conference (DMCC5), Charleston, SC, April 1990, pp. 1160–1170.

[18] D. Callahan and K. Kennedy, Compiling programs for distributed-memory multiprocessors, *J. Supercomputing,* 2, 151–169, October 1988.

[19] S. Chatterjee, J.R. Gilbert, R. Schreiber and S.-H. Teng, Optimal evaluation of array expressions on massively parallel machines, Technical report TR 92.17, Research Institute for Advanced Computer Science, NASA Ames Research Center, Moffett Field, CA, September 1992; also available as Xerox PARC Technical report CSL-92-11, *ACM Trans. Programming Languages Syst.* [submitted].

[20] S. Chatterjee, J.R. Gilbert, R. Schreiber and S.-H. Teng, Automatic Array Alignment in Data-Parallel Programs, in *Proceedings of the 20th Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, Charleston, SC, January 1993, pp. 16–28; also available as RIACS Technical report 92.18 and Xerox PARC Technical report CSL-92-13.

[21] S. Chatterjee, J.R. Gilbert, F.J.E. Long, R. Schreiber and S.-H. Teng, Generating Local Addresses and Communication Sets for Data-Parallel Programs, in Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, CA, May 1993, pp. 149–158; also available as RIACS Technical report 93.03.

[22] S. Chatterjee, J.R. Gilbert and R. Schreiber, Mobile and Replicated Alignment of arrays in Data-Parallel Programs, in *Proceedings of Supercomputing '93*, Portland, OR, November 1993 [in press].

[23] S. Chatterjee, J.R. Gilbert, R. Schreiber and T.J. Sheffler, Array Distribution in Data-Parallel Programs, in *Proceedings of the 7th Annual Workshop on Languages and Compilers for Parallel Computing*, August 1994.

[24] M. Chen, Y. Choo and J. Li, Compiling parallel programs by optimizing performance, *J. Supercomputing,* 2, 171–207, October 1988.

[25] R. Cytron, Doacross: Beyond Vectorization for Multiprocessors, in Proceedings of the 1986 International Conference on Parallel Processing, August 1986, pp. 836–844.

[26] J.Z. Fang and M. Lu, An iteration partition approach for cache or local memory thrashing on parallel processing, *IEEE Trans. Comput.*, 42(5), May 1993.

[27] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon and D. Walker, *Solving Problems on Concurrent Processors — Vol. 1: General Techniques and Regular Problems,* Prentice Hall, Englewood Cliffs, NJ, 1988.

[28] K.Gallivan, W. Jalby and D. Gannon, On the Problem of Optimizing Data Transfers for Complex Memory Systems, in Proceedings 1988 ACM International Conference on Supercomputing, St. Malo, France, June 1988, pp. 238–253.

[29] D. Gannon, W. Jalby and K. Gallivan, Strategies for cache and local memory management by global program transformations, *J. Parallel Distributed Comput.*, 5(5), 587–616, October 1988.

[30] J. Garcia, E. Ayguadé and J. Labarta, A Novel Approach Towards Automatic Data Distribution, in Proceedings of Supercomputing '95, San Diego, CA, December 1995.

[31] M. Gerndt, Array Distribution in SUPERB, in Proceedings 1989 ACM International Conference on Supercomputing, Athens, Greece, June 1989, pp. 164–174.

[32] J.R. Gilbert and R. Schreiber, Optimal expression evaluation for data parallel architectures, *J. Parallel and Distributed Comput.*, 13(1), 58–64, September 1991.

[33] G. Goff, K. Kennedy and C. Tseng, Practical Dependence Testing, in Proceedings of the SIGPLAN '91 Conference on Program Language Design and Implementation, Toronto, Canada, June 1991.

[34] M. Gupta and P. Banerjee, Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. *IEEE Trans. Parallel and Distributed Syst.*, 3(2), 179–193, March 1992.

[35] S.K.S. Gupta, S.D. Kaushik, S. Mufti, S. Sharma, C. -H. Huang and P. Sadayappan. On compiling array expressions for efficient execution on distributed-memory machines, in *Proceedings of the 1993 International Conference on Parallel Processing*, A.N. Choudhary and P.B. Berra, Eds., CRC Press, Boca Raton, FL, August 1993, Vol. 2, pp. 301–305.

[36] High Performance Fortran Forum, High Performance Fortran language specification version 1.0. Draft, Center for Research on Parallel Computation, Rice University, Hoaston, TX, December 1993.

[37] S. Hiranandani, K. Kennedy and C. Tseng, Compiler Optimizations for Fortran D on MIMD Distributed-Memory Machines, in Proceedings of Supercomputing '91, Albuquerque, NM, November 1991.

[38] S. Hiranandani, K. Kennedy and C. Tseng, Compiler support for machine-independent parallel programming in Fortran D, in *Languages, Compilers, and Run-Time Environments for Distributed Memory Machines*, J. Saltz and P. Mehrotra, Eds., North-Holland, Amsterdam, 1992.

[39] S. Hiranandani, K. Kennedy, J. Mellor-Crummey and A. Sethi, Advanced compilation techniques for Fortran D, Technical report Rice CRPC-TR-93-338, Center for Research on Parallel Computation, Rice University, Houston, TX, October 1993.

[40] C.-H. Huang and P. Sadayappan, Communication-free hyperplane partitioning of nested loops, *J. Parallel Distributed Comput.*, 19(2), 90–102, October 1993.

[41] D. Hudak and S. Abraham, Compiler Techniques for Data Partitioning of Sequentially Iterated Parallel Loops, Proceedings ACM International Conference on Supercomputing, June 1990, pp. 187–200.

[42] F. Irigoin and R. Triolet, Supernode Partitioning, in Proceedings 15th Annual ACM Symposium Principles of Programming Languages, San Diego, CA, January 1988, pp. 319–329.

[43] A. Karp, Programming for parallelism, *IEEE Comput.,* 20(5), 43–57, May 1987.

[44] K. Kennedy and U. Kremer, Initial Framework for Automatic Data Layout in FORTRAN D: A short update on a Case Study. Technical report Rice CRPC-TR93324-S, Center for Research on Parallel Computation, Rice University, Houston, TX, July 1993.

[45] K. Kennedy and U. Kremer, Automatic Data Layout for High Performance FORTRAN, Technical report Rice CRPC-TR94498-S, Center for Research on Parallel Computation, Rice University, Houston, TX, December 1994.

[46] I. Kim and M. Wolfe, Communication Analysis for Multicomputer Compilers, in Proceedings of Parallel Architectures and Compilation Techniques (PACT 94), August, 1994.

[47] K. Knobe, J. Lukas and G. Steele, Data optimization: allocation of arrays to reduce communication on SIMD machines, *J. Parallel Distributed Comput.,* 8(2), 102–118, February 1990.

[48] C. Koelbel, P. Mehrotra and J. van Rosendale, Semi-automatic process partitioning for parallel computation, *Int. J. Parallel Programming,* 16(5), 365–382, 1987.

[49] C. Koelbel, P. Mehrotra and J. van Rosendale, Supporting Shared Data Structures on Distributed Memory Machines, Proceedings Principles and Practice of Parallel Programming, Seattle, WA, March 1990, pp. 177–186.

[50] C. Koelbel, Compiling Programs for Non-Shared Memory Machines, Ph.D. thesis, CSD-TR-1037, Purdue University, West Lafayette, IN, November 1990.

[51] U. Kremer, Automatic Data Layout for Distributed-Memory Machines, Technical report Rice CRPC-TR93299-S, Center for Research on Parallel Computation, Rice University, Houston, TX, February 1993.

[52] U. Kremer, J. Mellor-Crummey, K. Kennedy and A. Carle, Automatic Data Layout for distributed-Memory Machines in the d Programming Environment, Technical report Rice CRPC-TR93298-S, Center for Research on Parallel Computation, Rice University, Houston, TX, February 1993.

[53] U. Kremer, NP-Completeness of Dynamic Remapping. Technical Report Rice CRPC-TR93330-S, Center for Research on Parallel Computation, Rice University, Houston, TX, August 1993.

[54] J. Li and M. Chen, Index Domain Alignment: Minimizing Cost of Cross-Referencing between Distributed Arrays, in Frontiers '90: The 3rd Symposium on the Frontiers of Massively Parallel Computation, College Park, MD, October 1990.

[55] J. Li and M. Chen, Generating Explicit Communication from Shared-Memory Program References, in Proceedings of Supercomputing '90, New York, NY, November 1990.

[56] W. Li and K. Pingali, A singular loop transformation framework based on non-singular matrices, in *Proc. 5th Workshop on Languages and Compilers for Parallel Computing*, Springer-Verlag, August 1992.

[57] W. Li and K. Pingali, Access Normalization: Loop Restructuring for Numa Compilers, in Proceedings 5th International Conference on Architectural Support for Programming Languages and Operating Systems, Boston, MA, October 1992, pp. 285–295.

[58] D.E. Maydan, J.L. Hennessy and M.S. Lam, Efficient and Exact Data Dependence Analysis, in Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, June 1991, pp. 1–14.

[59] D.E. Maydan, S.P. Amarasinghe and M.S. Lam, Array Data Flow Analysis and Its Use in Array Privatization, in Proceedings 20th Annual ACM Symposium on Principles of Programming Languages, January 1993.

[60] M. Mace, *Memory Storage Patterns in Parallel Processing,* Kluwer Academic, Boston, MA, 1987.

[61] M. O'Boyle, A data algorithm for distributed memory compilation, in *Proceedings of the 6th International PARLE Conference*, Athens, Greece, July 1994, Springer-Verlag,

[62] D.A. Padua and M. Wolfe, Advanced compiler optimizations for supercomputers, *Commun. ACM*, 29(12), 1184–1201, December 1986.

[63] D.J. Palermo and P. Banerjee, Automatic selection of dynamic data partitioning schemes for distributed-memory multicomputers, in *Proceedings of the 8th Workshop on Languages and*

Compilers for Parallel Computing, Lecture Notes in Computer Science, Columbus, OH, August 1995, Springer-Verlag,

[64] J. Ramanujam and P. Sadayappan, Compile-time techniques for data distribution in distributed memory machines, *IEEE Trans. Parallel Distributed Syst.*, 2(4), 472–482, October 1991.

[65] J. Ramanujam and P. Sadayappan, Tiling Multidimensional Iteration Spaces for Nonshared Memory Machines, in Proceedings of Supercomputing '91, Albuquerque, NM, November 1991.

[66] J. Ramanujam and A. Narayan, Automatic Distribution for HPF-Like Languages, Technical report TR-94-07, Department of Electrical and Computer Engineering, Louisiana State University, Baton Rouge, LA, January 1994.

[67] J. Ramanujam, Compile-Time Techniques for Parallel Execution of Loops on Distributed Memory Multiprocessors, Ph.D. thesis, Department of Computer and Information Science, Ohio State University, Columbus, OH, September 1990.

[68] A. Rogers and K. Pingali, Process Decomposition through Locality of Reference, Proceedings ACM SIGPLAN 89 Conference on Programming Language Design and Implementation, Portland, OR, June 1989, pp. 69–80.

[69] A. Rogers, Compiling for Locality of Reference, Ph.D. thesis, Cornell University, August 1990.

[70] M. Rosing and R. Weaver, Mapping Data to Processors in Distributed Memory Computations, Proceedings 5th Distributed Memory Computing Conference (DMCC5), Charleston, SC, April 1990, pp. 884–893.

[71] T.J. Sheffer, R. Schreiber, J.R. Gilbert and B. Pugh, Efficient distribution analysis via graph contraction, in *Proceedings of the 8th Workshop on Languages and Compilers for Parallel Computing, Lecture Notes in Computer Science*, Columbus, OH, August 1995, Springer-Verlag,

[72] J.M. Stichnoth, Efficient Compilation of Array Statements for Private Memory Multicomputers, Technical report CMU-CS-93-109, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, February 1993.

[73] C.-W. Tseng, An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines, Ph.D. thesis, Department of Computer Science, Rice University, Houston, TX, January 1993; available as technical report CRPC-TR93291.

[74] A. Wakatani and M. Wolfe, A new approach to array redistribution: Strip mining redistribution, in Proceedings of the Sixth International PARLE Conference, Athens, Greece, July 1994, Springer-Verlag.

[75] K. Wang and D. Gannon, Applying AI Techniques to Program Optimization for Parallel Computers, in *Parallel Processing for Supercomputers and Artificial Intelligence,* K. Hwang and D. DeGroot, Eds., McGraw-Hill, New York, 1989, pp. 441–485.

[76] M.E. Wolf and M.S. Lam, An algorithmic approach to compound loop transformations, in A. Nicolau, D. Gelernter, T. Gross and D. Padua, Eds., *Proceedings of the Third Workshop on Languages and Compilers for Parallel Computing*, MIT Press, Irvine, California, 1990, pp. 243–259; available as Advances in Languages and Compilers for Parallel Computing.

[77] M. Wolfe, Advanced Loop Interchange, in Proceedings 1986 International Conference on Parallel Processing, St. Charles, IL, August 1986, pp. 536–543.

[78] M. Wolfe, Loop skewing: the wavefront method revisited, *Int. J. Parallel Programming*, 15(4), 279–294, 1986.

[79] M. Wolfe, Multiprocessor synchronization for concurrent loops, *IEEE Software*, 34–42, January 1988.

[80] M. Wolfe, *Optimizing Supercompilers for Supercomputers,* Pitman Publishing, London and the MIT Press, Cambridge, MA, 1989.

[81] M. Wolfe, More Iteration Space Tiling, in Proceedings Supercomputing 89, Reno, NV, November 1989, pp. 655–664.

[82] M. Wolfe, *High Performance Compilers for Parallel Computing*, Addison-Wesley, Redwood City, CA, 1996.

[83] M. Wolfe and U. Banerjee, Data dependence and its application to parallel processing. *Int. J. Parallel Programming*, 16(2), 137–178, 1987.

[84] M.J. Wolfe and C. Tseng, The power test for data dependence, *IEEE Trans. Parallel and Distributed Syst.*, 3(5), 591–601, September 1992.

[85] H. Zima and B. Chapman, *Supercompilers for Parallel and Vector Computers*, Frontier Series, Addison-Wesley, Reading, MA, 1990.

[86] H. Zima and B. Chapman, Compiling for distributed-memory systems, *Proc. of the IEEE*, 81(2), 264–287, February 1993.

[87] H. Zima, H. Bast and H. Gerndt, SUPERB: A tool for semi-automatic MIMD-SIMD parallelization, *Parallel Comput.,* 6, 1–18, 1988.

13

Register Allocation

K. Gopinath

Indian Institute of Science

13.1 Introduction

An important part of code generation is to decide which values in a program reside in a register (register allocation) and in what register (register assignment). The values may include temporary variables introduced in the compilation process. The two aspects are often taken together and loosely referred to as register allocation. Some algorithms [2, 17] have an explicit separate phase for assignment whereas some combine both [49]; many combine both along with a separate assignment phase also [18, 52]. The fundamental problem to be solved is either the optimal reuse of a limited number of registers or the minimization of traffic to and from memory.

If the number of registers are not enough to hold all the values (the typical case), the problem of register allocation also includes determining the duration (or the segment of the live range) for which the variable resides in the register so allocated. Many current designs typically assume an unlimited number of virtual registers that get mapped to a finite number of physical registers during the process of register allocation. The virtual registers hold all the values of interest in the program that can be allocated to registers.

A register allocator thus has to decide the set of candidates to be allocated (virtual registers) and their associated live ranges. It next has to assign the registers to the candidates using some optimality criterion (e.g., minimize spilling). Finally, it has to rewrite the code reflecting the allocations.

The first part can include renumbering, coalescing, splitting and rematerialization.

- *Renumbering virtual registers*. At the abstract code level, we need to distinguish between unrelated variables that happen to share the same abstract storage location (e.g., on the stack) or are used by a programmer (e.g., use of a counter i in different loops). The reuse of the stack locations could be the result of storage optimization (as in machines with register windows or in cache locality improvement, etc.). If the reuse is not detected, the live range in a specific register becomes longer than logically necessary and increases register pressure. Separating out the live range so that each value range is a candidate for register allocation is helpful. Approaches such as renumbering [17], webs [82] or static single assignment (SSA)[1] have been used.
- *Splitting live ranges*. Instead of spilling a live range entirely, if it cannot be allocated a register because of conflicts, it may be better to split the live range so that some part of it can reside in a register, with move instructions connecting the pieces. The splitting reduces conflicts and can make allocation possible. The important issues are what live ranges to split and where to split.
- *Coalescing live ranges*. To avoid overhead, a register allocator should coalesce two temporaries that are related by a move instruction if this can be done without increasing the number of spills.
- *Rematerialization*. Sometimes it is cheaper to recompute a value (typically constants) than to keep it in a register. In such cases, registers can be freed earlier with a decrease in register pressure.

[1]In SSA, only one textual definition of a variable exists (there may be multiple redefinitions as this definition may be in a loop).

The second part of the register allocation problem is NP hard and various models have been used (graph coloring, bin packing, integer programming, other graph models based on cliques or feedback vertex set), though graph coloring is the most popular.

Some registers are often preallocated. For example, we may need some dedicated registers such as frame pointer (FP) or stack pointer (SP) though the need for an FP may be eliminated if *alloca* is not supported or used.[2] Without *alloca*, the offset from the base of the activation record of every variable and temporary is known at compile time.

To make some registers available for an instruction, spilling may be needed. However, spilling itself may need extra registers, especially in reduced instruction-set computing (RISC) architectures that do not support access to memory in instructions except through LD/STs and if the addressing of the spill location requires computation. Hence, some scratch registers may have to be preallocated. Two designs for register allocation are possible: either all the virtual registers have memory as the "home location" and some of them are promoted to registers by register allocation, or a register is the home location of a virtual register and demoted or spilled if the register is needed by a more important virtual register. Even though these two approaches seem symmetrical, there can be a difference in the number of scratch registers that may have to be preallocated. In the first case, an instruction at a particular point in the register allocation process may have multiple memory references that need to be promoted to register before the instruction can be logically executed. If all registers have been used by this point, current and future memory references can be handled only by a store (spill) and a load, and the quality of allocation can degrade sharply.[3] A better way is to leave some registers as scratch; the worst-case need for scratch registers is the maximum number of registers used in any instruction in a given instruction set. However, such a design can enhance portability because the code can be a high-level intermediate code with many specific details of the architecture or implementation not known; the scratch registers can help in emulating the abstract instructions. In the second approach, because all loads and stores from memory to virtual registers are already in the code, there may be no need to keep scratch registers if all "hidden" computations (such as address calculation, including the cases when certain sizes of offsets cannot be used directly in some instructions) are also exposed into the code.[4] Although this can give better allocations, it can be costly in terms of compile time. Every time a variable is spilled, it not only changes the code to be optimized (similar to the first case where a memory resident variable has to be loaded into a scratch register) but also changes the register conflicts.[5] Hence, the allocation exercise has to be repeated on the new code until no further spilling occurs. The first case is pessimistic (because some registers have to be set aside) with the advantage of having more portability whereas the second is optimistic with the disadvantage of all hidden computations having to be exposed and hence less portable.

When only linear code is considered (i.e., no branches), an interference graph constructed from the interval graph[6] is a good representation. First, the interval graph of the liveness of variables can be computed and then the interferences (nonempty intersection of intervals) can be computed.

[2]*allocates* temporary space in the stack frame of the caller.

[3]*gcc* [39] uses this strategy but does "postload" analysis to improve code (see Section 13.9.1). The problem becomes acute for very long instruction word (VLIW)/(EPIC) processors, because each spill instruction may have to be scheduled in a cycle of its own. For such architectures, Berson et al. [13] report that spilling is poorer than splitting live ranges when there are excessive register and functional unit demands.

[4]Note that although instruction selection phase handles the issue of what instruction to use given certain offsets, a phase-ordering issue exists because the offset of a spilled value is only known at the end of register allocation.

[5]See Section 13.5.1.1, especially the last part of the discussion on splitting, for further clarification.

[6]Interval graphs are defined as in graph theory, not as the intervals of data flow analysis.

If the chromatic number of the resulting graph is less than the registers available, the code can be register allocated without any spills (i.e., unloading of a variable into memory). Interval graphs can be colored optimally in polynomial time (see Section 13.3.1.2).

In the presence of branching, an interference graph is still a good representation but interval graphs are no longer the case. We can use simpler but inaccurate representations, use more complex models such as using predication to still get nonbranching code or use specific models such as cyclic interval graphs for nested loops [53]. The former technique is widely used. For example, consider a control flow graph (CFG) with many basic blocks. Instead of modeling the liveness of variables at instruction level along with branching information, one can model liveness at the basic block level and ignore the CFG structure so that in effect all basic blocks are now at a flat level. Although such a representation loses accuracy, it may be simpler and faster to solve. Another possibility is to use predication to convert the CFG to a linear set of instructions through *if-conversion* [59]. This also loses some accuracy because the optimal splicing of instructions from different branches for linearized predicated instructions is nontrivial, from a register allocation perspective.

It has been shown that register allocation is an NP-complete problem for general graphs when the corresponding decision problem is modeled as follows: *Is an undirected graph k-colorable?* The (three-conjunctive normal form (3-CNF)) satisfiability problem can be reduced to this problem [3]. This formulation does not consider the cost models for stores or loads. If the problem of register allocation is formulated as the minimization of memory traffic between registers and memory, the problem is also hard, as has been shown by Farach and Liberatore [35].

However, the issue is minimizing the spills instead of maximizing the number of live ranges not spilled. Although the two problems are equivalent in the case of finding optimal solutions, they are not in the context of heuristic solutions or approximate solutions. Hence, the model used (graph coloring, bin packing, integer programming, other graph models based on feedback vertex set or cliques) can have a direct bearing on the quality of the code produced, though graph coloring is the most popular model.

Another important aspect is the heuristic time and space complexity of the algorithms for register allocation. Because programs can be quite large with hundreds to thousands of variables (more accurately, value ranges) in a function and the number of interference edges can be in the range of millions (e.g., the combined interference graphs of procedures and functions in *gcc* of mid-1990s have approximately a total of 4.6 million edges), decomposition strategies are critical. Traditionally, these have been handled at the function and procedure level. Hierarchical [22, 73] and profile-based models (region-based approaches [52, 54]) have become increasingly popular. Graph-theoretical decomposition models have also been examined [86, 94]. These models can also reduce the need for large space requirements, with the vertical model of region-based compilation (see Section 13.7) the most aggressive. By using profile information, the region-based approaches may have already decided, by register allocation time, on some aspects of renumbering, coalescing and splitting due to the decomposition of the program into regions and the corresponding decomposition of live ranges. If this decomposition is not appropriate, then the register allocator might have to reorganize the regions for better performance.

The heuristic complexity that seems to be acceptable from a practical viewpoint is at most $n \log n$ (empirical results for Chaitin's register allocator with instruction-level liveness granularity, hence larger graphs) or n^2 (empirical results for Chow and Hennessy's register allocation with basic block level liveness granularity, hence smaller graphs) [16]. Here n is the object size of the program or routine in bytes. With the introduction of dynamic compilation, even faster approaches are becoming necessary. For SPEC92 benchmarks, although the *gcc* compiler (2.5.7) takes an average of 0.2 sec per function, it takes 0.04 sec for register allocation [49] (i.e., 25% of the total time).

13.1.1 Importance and Future Trend

It has been often called one of the most important — if not the most important — optimization [57, 82] in compilers. With older complex instruction set computers (CISCs) there have been two difficulties: limited number of registers and nonuniform registers (i.e., implicit use of certain registers for some operations, etc.). Register allocation is critical in getting performance out of CISC machines [2].

In RISC and VLIW/EPIC[7] designs, designed to exploit available instruction parallelism through compiler optimizations, two important optimizations in the back end that finally determine the quality of code are instruction scheduling and register allocation. Instruction scheduling exposes the instruction level parallelism in the code, whereas register allocation assigns registers to frequently accessed variables. Many of the architectural details are exposed to enable compilers to use the resources effectively. The back end optimizations are done on assembly code or a low-level instruction fetch (IF) of the code because all the address computations must be made explicit and all machine details have to be incorporated for performance. It is now crucial to do a good job of combining instruction scheduling and register allocation. However, strong interactions occur between these optimizations. For example, if register allocation is done before instruction scheduling, false dependences (anti- and output dependences) may be introduced due to register reuse, which limits the scheduler's reordering possibilities. However, instruction scheduling typically increases the register requirements (register pressure) and may result in spill code. Such spill code may not be necessary if register renaming is done in hardware at runtime (quite common in current superscalar microprocessors) but this is too late because the compiler cannot know at compile time and has to spill.

With RISC designs, the number of registers has been at least 32 and in current VLIW/EPIC style designs, as many as 128. Because of the abundance of these registers, the importance of register allocation is widely perceived to have been reduced vs. other optimizations such as instruction scheduling. Hence, many compilers for such machines first attempt instruction scheduling, then register allocation and finally again attempt instruction scheduling to handle code corresponding to spills [9].

However, aggressive compiler techniques such as loop unrolling, promoting of subscripted array variables into registers (especially in loops) and interprocedural optimizations create heavy register pressure and it is still quite important to do a good job of register allocation.

For just-in-time (JIT) compilation for Javalike systems, register allocation is also critical and many different register allocators that optimize speed of compilation or execution efficiency may be needed in the same system. For example, the Jalapeno quick compiler [58] compiles each method as it executes for the first time and balances compile time and runtime costs by applying some effective optimizations, with register allocation the most important.

13.1.1.1 Patents on Register Allocation

Some critical algorithms have been patented in the United States such as the use of graph coloring for register allocation (held by IBM, Chaitin et al., U.S. Patent 4,571,678, 1986) and a dependent patent on optimistic graph coloring (held by Rice University, Briggs et al., U.S. Patent 5,249,295, 1993). Similarly, the hierarchical graph coloring method (by Callahan and Koblenz, U.S. Patent 5,530,866, 1996), the linear scan and bin packing allocators (by Burmeister et al., U.S. Patent 5,339,428, 1994), an interprocedural register allocator (held by HP, Odnert et al., U.S. Patent 5,555,417, 1996) are also covered. However, the Chow and Hennessy priority-based coloring method is not patented. Approximately 22 patents granted seem to be exclusively concerned with register allocation as of December 2001.

[7]EPIC is the basis of a new VLIW-like architecture that Hewlett-Packard (HP) and Intel have designed [70].

It is believed that *gcc* from the Free Software Foundation avoids the IBM patent by spilling a real register during the reload phase instead of a symbolic register during simplification [71]; we discuss this further in a case study.

13.1.2 Related Topics: Storage Optimization, Paging, Caching and Register Renaming

One related area is storage optimization that minimizes storage for a computation. They were first explored as early as 1961 [11] and seem to have been the inspiration for the later work on the coloring approach. Fabri [60] developed a live range splitting technique for minimizing storage that has been more successfully used in register allocation [18]. Later work on storage optimization has concentrated on update in place optimizations or copy elimination, especially important for functional languages or memory-constrained embedded devices. Minimizing storage through targeting [41] and single threading [87] has close analogs (coalescing, e.g., [38]) in register allocation. Storage optimization has an impact on register allocation: if storage locations are reused, any spill locations (home locations) also get reused and results in better cache performance.

Other related ideas are paging (from operating systems [OSs]) and caching and register renaming (from computer architecture). Paging, caching and register allocation all exploit memory hierarchy but are different mechanisms. In register allocation, the user or compiler manages the hierarchy whereas it is automatic in case of caches and paging. Because a register is expected to be reused, instruction level parallelism may be compromised due to increase in anti- and output dependences with register reuse. Register renaming in some architectural designs is a way of eliminating these false dependences automatically through the use of extra hidden registers in the microarchitecture that are not present at the instruction level.

Differences between paging, caching and register allocation — In paging and caching, one assumes that the memory ref streams are not a given and the job of managing the hierarchy has to be done on-line as best as possible. In register allocation, the future references to variables are known to some extent because the full text of the code may be potentially available. Even if the full text of the code is available, the exact code paths may not be known. In realistic situations, one cannot even assume that the full text of the code is available. For example, if a program uses dynamic linking (the norm nowadays because the C library is dynamically linked), the exact code executed is not known until runtime. If static linking is chosen, for good register allocation, it might be appropriate to attempt it at link time because more information is available. However, the problem is not so much the question of having the full text of the code or not: it is the ability to do the register allocation analysis in an economical way so that quadratic effects (using heuristics) or worse (register allocation is NP-complete) do not make the register allocation have unacceptable times.

In the paging or caching problem, typically only one page or cache reference is generated in one step (assuming an uniprocessor).[8] In local register allocation, multiple references can be generated in one step (e.g., ADD R1, R2, R3). Multiple simultaneous references arise also in VLIW architectures.

13.1.3 Brief History

The first significant exercise in register allocation occurred as early as 1954 [99] in the context of the FORTRAN compilation system. Even though the architecture had only two registers, there was surprisingly a substantial effort in doing a good job of register allocation. For example [99]:

[8]We ignore older machines such as VAXes that could refer to multiple memory addresses in one instruction.

The degree of optimization . . . achieved was really not equalled again in subsequent compilers until the mid-60s when the work of Fran Allen and John Cocke began to be used in fairly sophisticated optimizing compilers. . . . The index register allocation procedure . . . was at least as optimal as any procedure developed during the next 15 years, and it probably is in fact optimal.

For straight-line code, the replacement policy in the allocator was the same as that used in Belady's MIN algorithm, which Belady proved to be optimal [15].

Ershov in the former Soviet Union developed the algorithm for optimal register allocation on expression trees. The FORTRAN H compilation system for the IBM 360 (ca. 1967) is another milestone. It used dominators to identify loops in CFGs of basic blocks [75] and used data flow analysis to compute accurate live-in and live-out information to compute costs and benefits of allocating a register to a variable (see Section 13.6.5.3).

The Bliss-11 compiler has been a groundbreaking optimizing compiler, including in register allocation that is viewed as a bin packing problem. Chaitin et al. [45] studied coloring as a systematic technique in 1979 and this technique became important with RISC architectures. Chow and Hennessy [18] introduced priority coloring in 1984.

Hierarchical methods have been studied since 1990. Knobe and Meltzer [72] (and later Knobe and Zadeck [73]) first gave a model (control tree based register allocation), which is also similar to an another model (hierarchical graph coloring with tiling) developed around the same time by Callahan and Koblenz [22] but without some problematic aspects of the first.

Another hierarchical model is based on regions. Region-based approaches have been proposed since the 1990s. One significant experimental compiler has been the ELCOR research compiler from HP Labs that has systematically explored the region-based approach. This back end is available as a part of the publicly available Trimaran compilation system [98].

13.1.4 Outline of Chapter

In Section 13.2, we discuss some of the background on the subject. In Section 13.3, we cover various theoretical models for the register allocation problem. In Section 13.4, we describe some of the techniques used in local register allocation. In Section 13.5, we discuss some of the techniques used in global register allocation but give some emphasis to the graph coloring approach because it is widely used. We also give a high-level description of an hierarchical algorithm based on graph coloring. The books [1, 82] also give up-to-date treatments and can be fruitfully studied, though they differ in specifics. They also have detailed examples of the coloring algorithm. We also examine other approaches such as those based on bin packing, integer linear programming and interprocedural register allocation; and end the section with region-based approaches.

In Section 13.6, we descibe as a case study a region based register allocator that is present in Trimaran, a publicly available compilation system [98]. We first give an overview of the approach, the performance problems with a basic design and then enhancements that are required to make it competitive. Though the compilation system is designed for VLIW/EPIC models, we restrict our attention to the classical register allocation problem (we do not discuss, e.g., scheduling-aware register allocation or cache miss or memory hierarchy optimizations necessary for VLIW/EPIC).

In Section 13.7, we cover phase-ordering issues with respect to register allocation and instruction scheduling and present a framework for phase ordering these optimizations. Instead of the horizontal model of compilation (where instruction scheduling is done on all units of compilation before register allocation and then vice versa), we would like to design a vertical model where instruction scheduling, register allocation and instruction scheduling again are done on each unit of compilation before another unit is considered. This can help in reusing information across the phases as well as incrementalizing some aspects of the optimizations, resulting in space and also time savings. If the unit is as small as an operation, we get a fully combined instruction scheduling and register allocation. However, because such a design is quite difficult (register allocation and instruction scheduling become very tightly coupled), we present a design with an unit that is a reasonably sized region (such as a basic block or hyperblock). In such a design, instruction scheduling and register allocation are decoupled to some extent. However, it is important still for register allocation to influence the scheduling and vice versa. This is carried out by first using a profile-insensitive list scheduler, then incorporating register pressure to make it sensitive to register allocation costs and finally making it profile sensitive. This is discussed in Section 13.8.

Finally, we end with two more case studies: one on *gcc* [39], a freely available compiler from Free Software Foundation, and a brief discussion on Jalapeno, a JIT compiler. An appendix gives some details about the Trimaran system.

13.2 Background

13.2.1 Types of Register Allocation

Various types of register allocation are available but the most frequently used classification depends on the scope of the allocation such as trees, directed acyclic graphs (DAGs) (basic blocks), loops, function, region based (including hierarchical), interprocedural and link level:

- *Local allocation.* This is only within a basic block. In the past, allocation within traces, superblocks and hyperblocks may also have been considered local but we consider them as region based (see later) given the generality of grouping multiple basic blocks. Local allocation is important for long basic blocks that usually result in scientific computations with heavy loop unrolling.
- *Global allocation.* This is within a procedure or function. This is needed to support separate compilation. For most programs, local allocation is just not sufficient due to the extensive branching. The local and global allocations are the most widely used scopes for register allocation.
- *Instruction level allocation.* Instruction level allocation may be needed when integrated register allocation and instruction scheduling is attempted. In many architectures with large numbers of registers, instruction scheduling is more important; hence when a cycle-level scheduler is used, registers are also modeled as a resource. Before scheduling an instruction, a check of the available registers is done along with the performing of any spill actions.
- *Interprocedural allocation.* This is across procedures but the full-blown version is usually not supported due to its complexity. The simpler and standard form that is usually followed is the callee and caller models of saving across calls (see later). One can consider this as a special case of region-based allocation (that is discussed next) with the region of whole procedures or functions. We consider it separately, however.
- *Region-based allocation.* This attempts to group significant basic blocks, even across procedure calls if appropriate, so that register allocation among these basic blocks is effective. The first two attempts have been with respect to traces [28] and loops [23, 30, 93]. Another approach has been in Lisplike languages where handling recursion efficiently is important: here leaf procedures are handled differently.

  In the Java type of environments where byte codes are interpreted at runtime, JIT may be used. This profiles the execution and compiles only some portions of the code. The register allocation is likely to be only of a particular region. We discuss region-based allocation in some detail.
- *Link level allocation.* This is at object time before the final loading phase. Given that separate compilation is common, it is necessary to do the simpler form of interprocedural allocation

(using caller and callee models) across procedures whose code is available only at link time. This requires rewriting of instructions with new register allocations.
- *Runtime allocation*. This is needed in dynamic compilation such as JIT for Java (as discussed earlier).

Also other aspects and models are:

- Whether speed of allocation or quality is important. For JIT applications, speed is often more important than quality. For certain other cases, quality is absolutely critical (e.g., real-time or embedded applications with severe constraints such as time or memory) and approaches based on exhaustive enumeration, integer programming or other approaches similar to superoptimizers might be used when human coding is problematic.
- Whether register allocation is on an intermediate form (e.g., UCODE, ELCOR) or an assembly code. When an intermediate form is used, one instruction can map to multiple lower level instructions and setting aside scratch registers to avoid deadlock during allocation process is critical. Another major problem is the phase-ordering problem with instruction selection and register allocation. For example, instruction selection depends on whether an operand is in memory or register whereas register allocation depends on the exact sequence of instructions.
- What machine models are used — CISC [2], RISC [17] or vector/VLIW/EPIC [52, 55, 68–70] — whether predicated instructions exist. Vector register allocation is usually handled by the programmer but register allocation for software pipelined loops is now also done in the compiler (as in ELCOR). More specifically, what register models are available? Many possibilities exist: small and large register sets, multiple register files, regular vs. irregular registers, type of stack layout and calling conventions, etc. Also, whether a module for machine descriptions exists (*gcc*, ELCOR) that has to be consulted during allocation is a consideration. This is especially important for VLIW/EPIC types of architectures that usually have multiple functional units and multiple register files.
- Whether the target language has specific features. For example, FORTRAN and C have different aliasing models; object-oriented languages such as C++ and Java may require special support (a region in these languages may be the constructor followed by the specific method call, etc.); languages like Lisp that use recursion heavily, functional languages may require coalescing or targeting [41] because a value defined is not changed or changed in a highly constrained manner (as in SSA or single assignment).
- Whether subscripted array variables are allocated or only scalars. This is quite important in scientific computation where subscripted array variables are often extensively used in tightly nested loops.

Another aspect is the interaction with other optimizations: whether stand-alone or combined with other optimizations (e.g., instruction scheduling, copy propagation, instruction selection in code generators) and how phase dependencies are handled. We have already discussed the latter for instruction scheduling (Section 13.1.1) whereas for instruction selection one strategy is to perform instruction selection twice — once to estimate the register needs and then again after register assignments. We study interaction with instruction scheduling in some detail.

13.2.2 Interaction with Pointers and Call Conventions

Many languages allow a pointer be taken of a variable. Such variables cannot wholly reside in registers. Even if they are allocated both a register and a memory location, consistency issues can arise. Any update to a register has to be delivered to the memory location before an access through memory. These requirements become extremely severe with aliasing. Hence, if a pointer is taken of a variable, it is usually not allocated to a register. If good quality alias information is available, then one can consider register promotion [25] (especially for languages such as C) that determines which scalar variables can be safely kept in registers and rewrite the code to reflect those facts.

Call conventions — Many architectures have calling conventions where certain registers are denoted as caller-save and others as callee-save. The former are suitable if the calling routine has only a few registers live after the call and with small callee procedures. The latter are useful for large procedures because they require work on entry to the procedure. In addition, the callee might have a better knowledge of the code and may wish to insert the stores at the right place in the callee (the subject of the shrink wrapping optimization). With interprocedural register allocation, callee registers are more useful because information about register usage in the caller and callee procedures are known.

13.3 Theoretical Results

13.3.1 Restricted Acyclic Graphs

13.3.1.1 Trees

There is an optimal algorithm (Sethi–Ullman [SU] algorithm) for trees under many cost models (Section 13.4.1.1). However, for even simpler graphs such as chains, the problem becomes NP-complete [91] if instruction scheduling is included. With restriction of delay slot delays to one, the problem when combined with instruction scheduling is tractable as in the delayed load scheduling (DLS) algorithm [88].

13.3.1.2 Graphs Resulting from Basic Blocks

The resulting interference graph is an interval graph [79] for which also optimal (polynomial) algorithms exist. The order of the largest clique in the graph ($C(G)$) is the same as the chromatic number ($K(G)$) in an interval graph (and also for bipartite graphs), and a greedy strategy that assigns the smallest indexed color already not used is optimal [103]. To see this, consider an interval that has been colored with a color with index p. Because this is the smallest indexed color already not used, there is a clique with intervals colored with indices from 1 to $p - 1$. Hence, $C(G) \geq p \geq K(G)$. Because $K(G) \geq C(G)$ for all graphs, the coloring is optimal. More elaborate cost-based approaches, however, are NP complete (Section 13.3.2.2). See also Section 13.4.2.

13.3.2 Unrestricted Acyclic Graphs

The problem is NP complete for general graphs [3]. The optimality result for interval graphs in Section 13.3.1.2 does not carry over with the introduction of branches because a live range is now a set of intervals instead of an interval, and maximal cliques do not necessarily develop.

Various models have been used for handling unrestricted acyclic graphs: graph coloring through simplification, bin packing, integer programming, other graph models based on cliques or feedback vertex set. Though all these problems are NP-complete, they have different “approximability.” For example, whereas a good approximate solution for graph coloring is not possible (see later), bin packing has good constant factor approximations both in off-line (the situation mostly obtained in the compiler case) and on-line cases. In the off-line case, if m bins are optimal, it may need at most $11 \cdot m/9 + 4$ bins but for the on-line case it might need $\lceil 17 \cdot m/10 \rceil$ bins. If worst-case guarantees are needed for the approximation algorithms, some models (e.g., based on bin packing used in some compilers or feedback vertex set advocated in CRISP [81]) may be better than widely popular models such as graph coloring.

13.3.2.1 Coloring Models

A (vertex) coloring of an undirected graph is an assignment of a label to each vertex such that the labels on the pair of vertices incident to any edge are different. A minimum coloring of a graph is a coloring that uses as few different labels as possible, with the chromatic number its number of colors.

More than a century and quarter ago Kempe [63] developed the coloring approach based on simplification:[9] a graph G having a node X with degree less than k is k colorable if and only if the reduced graph G' formed by removing X with all its adjacent edges is k colorable.

Clique and coloring problems are closely related: the size of the maximum clique is a lower bound on the minimum number of labels needed to color a graph. The maximum clique problem finds as large a set of pairwise incompatible items as possible. The minimum coloring problem is to group the items into as few groups as possible, subject to the constraint that no incompatible items end up in the same group.

Both the problems, finding the maximum clique or minimum coloring, are formally NP-hard for general graphs [44]. It is therefore unlikely that it is possible to find a fast (i.e., polynomial time) algorithm to solve these problems exactly. In addition, based on the results of [5, 7, 14, 33, 78], it seems unlikely that it is even possible to find an approximate solution to these problems quickly: Bellare, Goldreich and Sudan [14] show that, assuming $P \neq NP$, for any $\epsilon \geq 1/4$ (for clique) or $1/7$ (for coloring), no polynomial time approximation algorithm can find a solution that is guaranteed to be within a ratio of N^{ϵ} of optimal.[10] They also show a value of $\epsilon \geq 1/3$ (for clique) and $1/5$ (for coloring) assuming[11] *co-RP*$\neq$*NP*. Because N, the number of nodes in the interference graph, is often more than 1000 for whole procedures, this ratio is often more than 2:3 or 2:4. Hence, if coloring is used, decomposition of the graph so that the number of nodes in the graph is below 500 (or even 100) is beneficial.

Wigderson's approximation algorithm [102] for graph coloring runs in polynomial time but can use as many as $2kN^{1-1/(N-1)}$ colors where k is the chromatic number.

13.3.2.2 Cost-Based Models

If the problem of register allocation is formulated as the minimization of memory traffic between registers and memory, the problem is also hard as has been shown recently [35]. More specifically, consider local register allocation [35] that assigns registers to variables in basic blocks, which are maximal branch-free sequences of instructions.[12] An optimum local allocation schedules the loading of values from memory into registers and the storing from registers into memory. The main difficulty of local register allocation stems from the trade-off between the cost of loads and the cost of stores. The cost of an allocation breaks down into the cost for loading and the cost for storing. The cost for loading is proportional to the number of times a register is spilled while alive. However, the cost for storing is a fixed cost that is paid once or not at all. If a cost is charged for storing, that cost does not depend on the number of times a register is actually spilled. An optimum allocation is hard to find because of the fixed costs due to stores.

[9]However, in this work, he wrongly claimed to have solved the four-color problem.

[10]We use N as the number of vertices in the interference graph all through this chapter.

[11]*RP* is randomized polynomial time. A randomized algorithm accepts strings in a language L in *RP* in polynomial time with a probability of success (say) of $\geq 1/2$ without accepting any string not in *L. co-RP* is the complement of *RP*.

[12]The local register allocation problem as considered is general enough to model off-line paging with write backs and weighted caching.

13.3.2.3 Combined Register Allocation and Instruction Scheduling Problem

This model from [81] is quite instructive, though its primary focus is on integrating instruction scheduling and register allocation. Because instruction scheduling is an NP-complete problem for a basic block whereas a register allocation problem can be optimally solved in polynomial time (Section 13.3.1.2), the combined problem is likely to be NP hard. This has been shown to be the case even if the edge latencies are all 0 and only one register exists [81].

We first introduce the important concepts and notation from that paper. Let $V = \{v_1, \ldots, v_m\}$ be the set of instructions in a basic block. Each instruction is labeled with an execution time $t(v_i)$. $DG = (V, E)$ is the data-dependence graph where each edge $(v_i, v_j) \in E$ is labeled with an interinstruction latency $l(v_i, v_j) \geq 0$ — this means that instruction v_j must start at least $l(v_i, v_j)$ cycles after the completion time of instruction v_i. A schedule, Σ, specifies the start time $\sigma(v_i) \geq 0$ for each instruction v_i. The completion time of schedule Σ is simply $T(\Sigma) = max_i\{\sigma(v_i) + t(v_i)\}$, the completion time of the last instruction to complete in Σ.

For a given schedule Σ, we define a value range of virtual register r to be a triple $(r, \sigma(v_i), \sigma(v_j))$, such that:

- Instructions v_i and v_j are in the basic block.
- For $\sigma(v_i) < \sigma(v_j)$, instruction v_i is scheduled before instruction v_j in Σ.
- Instruction v_i is either a producer or a consumer of virtual register r.
- Instruction v_j is a consumer of virtual register r.
- There is no intervening instruction v_k such that $\sigma(v_i) < \sigma(v_k) < \sigma(v_j)$ and instruction v_k is a consumer of virtual register r.

A value range is different from a live range, which extends from the first definition to the last use. Value ranges represent the finest granularity of splitting live ranges. Let $VAL(\Sigma)$ be the set of value ranges over all virtual registers in schedule Σ. The bandwidth of $VAL(\Sigma)$ at any time τ, $BW(\Sigma, VAL(\Sigma), \tau)$ is the number of value ranges in $VAL(\Sigma)$ that start at some time $< \tau$ and end at some time $\geq \tau$ in schedule Σ. Now, let R be the number of physical registers available. Register spills are required if the bandwidth exceeds R at any time τ (i.e., if $BW(\Sigma, VAL(\Sigma), \tau) > R$ for any time τ. The spill choices made by a solution to CRISP are reflected in $SVAL(\Sigma) \subseteq VAL(\Sigma)$, the set of spilled value ranges, and $AVAL(\Sigma) = VAL(\Sigma) - SVAL(\Sigma)$, the set of active (nonspilled) value ranges. Spilling a value range (r, τ_1, τ_2) has the effect of forcing the value contained in virtual register r to reside in memory during the time interval (τ_1, τ_2), thereby avoiding the need for a physical register to hold that value during that time interval. The overhead of spilling a single value range consists of a store instruction inserted at time τ_1 and a load instruction inserted at time τ_2. For architectures that charge a single cycle overhead for each instruction, the total cost of all spilled value ranges equal twice the number of spilled value ranges. The combined cost function for CRISP is then given by the following problem statement:

PROBLEM 13.1. *For CRISP find a resource-feasible schedule Σ and spilled value range set $SVAL(\Sigma)$ so as to minimize the combined cost function $C(\Sigma) = T(\Sigma) + 2|SVAL(\Sigma)|$, the sum of the completion time of the schedule and twice the number of spilled value ranges. $C(\Sigma)$ represents the overall execution time of the basic block if we assume that each spill increases the execution time by two cycles.*

There is a polynomial time reduction from a feedback vertex set[13] to a restricted version of CRISP termed RCRISP (which has all edge latencies zero and only one register), and hence RCRISP

[13]Is there a set of k vertices for which removal (along with incident edges) results in an acyclic graph?

is NP-hard [81]. However, there exists a constant-factor approximation to CRISP with a small approximation ratio in practice ($1 + c$ where c is the average value of the number of operands that can be in registers across all instructions).

It is not known whether the preceding approximation results can be extended to unrestricted acyclic graphs. If that is the case, the CRISP approach may be better than the graph coloring approach because the problem of approximating to within a ratio of N^{ϵ} the maximum R-colorable vertex-induced subgraph of a given interference graph G (unrestricted acyclic graph) is NP-hard. Also, the problem of approximating to within a ratio of N^{ϵ} the minimum number of vertices to delete from an interference graph G to leave an R-colorable subgraph is also NP-hard.

13.3.3 Cyclic Graphs

Register allocation for variables in nested loops are related to the class of circular arc graph coloring problems [46, 64]. A graph G is called a circular arc graph if its vertices can be placed in a one-to-one correspondence with a set of circular arcs of a circle in such a way that two vertices of G are joined by an edge if and only if the corresponding two arcs intersect one another. If the intervals are periodic, one can fit them into one circle. Theoretically, the problem of determining a k coloring for a circular arc graph with n arcs has a complexity of $O(nk!k \log k)$ [46].

13.4 Local Register Allocation

13.4.1 Register Allocation in Basic Blocks

A local (basic block) register allocator does not consider the liveness of a variable across block boundaries; all live variables that reside in registers are stored at the end of each block. Because the size of most basic blocks is short, such a register allocator can introduce considerable "spill code" at each block boundary.

Expressions in a basic block can be represented as DAGs where each leaf node is labeled by a unique variable name or constant, and interior nodes are labeled by an operator symbol having one or more nodes for the operation as children.

When the expression DAG for a basic block is a tree, the SU algorithm [3] generates an optimal solution for register allocation.

13.4.1.1 Sethi–Ullman Numbering

The first part labels each node of the tree with an integer (the Sethi–Ullman number) that denotes the fewest number of registers required to evaluate the tree without spilling. The labeling can be done by visiting nodes bottom-up so that a node is not visited until all its children are labeled (postorder traversal). Given two labeled children nodes, the parent node label is set to the label of the child node requiring more registers. If the register requirement of both children is the same, either node can be evaluated first with the parent needing one more register than its children. The label is defined as:

$$label(m) = \begin{cases} \max(l_1, l_2) & \text{if } l_1 \neq l_2 \\ l_1 + 1 & \text{if } l_1 = l_2 \end{cases}$$

The second part is to generate the code during the tree traversal. The order of tree traversal is decided by the label of each node: first evaluate the node requiring more registers. When the label of a node is bigger than the number of physical register R, the spill code has to be introduced. Figure 13.1 shows an example of SU numbering where we need three registers.

This algorithm assumes that all registers are equivalent, the trees are binary and the value of the result fits into one register. Appel and Supowit [6] have generalized the SU algorithm to remove

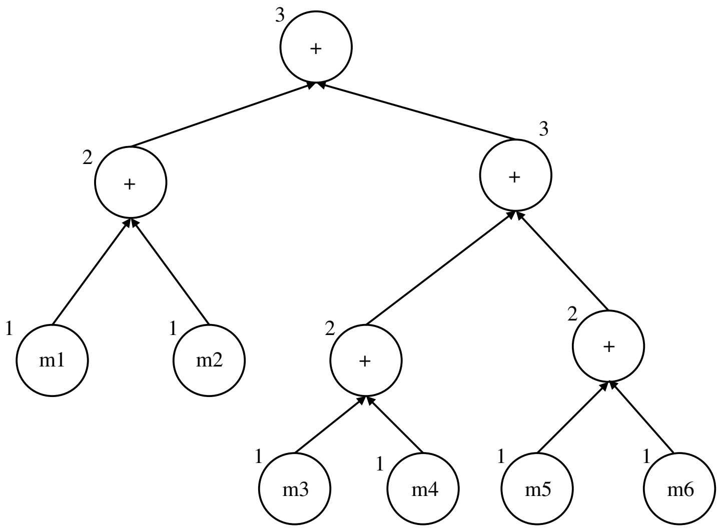


FIGURE 13.1 Sethi–Ullman numbering.

the last two assumptions. Given any tree, assume that a registers are required and that b registers are required to hold the result. Then any ordering that is nondecreasing in $a - b$ of the k children of the root minimizes the number of registers needed.

13.4.2 Minimization of Loads and Stores in Long Basic Blocks

In this approach [51], suitable for long basic blocks, a "shortest path" through a weighted DAG is found. At each instruction where no free registers are available, the register to spill can either be the next farthest to be read or written and which can be either in clean, dirty or dead (no future reference or next access is a write) state. Given a cost model for a load or store, depending on whether the spillee is in a clean, dirty or dead state, each allocation results in a different configuration. The problem now is that of searching through this weighted DAG for the minimal cost path. Because the number of configurations grows exponentially with the number of virtual registers VR and number of registers $R\left(=\binom{|VR|-1}{R-1}\cdot 2^R\right)$, some way is needed for pruning this DAG. The following are some of them: replace a dead register, among the clean registers replace the farthest [15], etc. However, these are not enough and further heuristics are necessary. For example, if there are two candidates for spilling (a dirty x and a clean y) and y is referenced sooner than x, then replacing y might be cheaper (because no store of y is required). By using some weights for multiplying the distances, they arrive at an heuristic weighted cost algorithm that performs within 10% of optimal and better than other algorithms such as Belady MIN or clean first (spill the most distant clean variable; if none, spill the most distant dirty one).

13.4.3 Register Allocation in Traces: Multiflow Compiler

The Multiflow compiler performs its register allocation on a trace [61]. Briefly, a trace is a linear sequence of basic blocks without internal loops. The Multiflow compiler combines register allocation and instruction scheduling. After constructing a sequence of basic blocks, register allocation is performed along with instruction scheduling. Trace scheduling gives priority to blocks in the highest frequency trace; thus, instruction scheduling and register allocation in these blocks do not have constraints from other blocks. A variable is bound to a register only when the instruction scheduler sees an actual reference to that variable. The live ranges extend outward from the highest frequency traces, and compiler generated compensation code is pushed outward to low frequency traces. Register spilling may occur inside a trace if not enough registers are available.

When scheduling a trace, the instruction scheduler must decide from where to read the variables that are live on entry to the trace and where to write the variables that are live on exit from the trace. For the first trace, no binding decisions have been made and the instruction scheduler is free to reference these variables in the locations (memory or register) that result in a good schedule. Most subsequent traces are entered from, or branch to, other traces where code has been scheduled and binding decisions have already been made. When a trace is entered from a register allocated trace, it must read its upward-exposed variables from locations where last written. When a trace branches to a register allocated trace, it must write downward-exposed variables into locations that are read downstream. This process is effected through a data structure called value location mapping (VLM). A VLM maps a set of variables into a set of locations. VLMs are created by the instruction scheduler after the instruction scheduler has generated code for a trace.

Some variables are live from the top to the bottom of a trace but are not actually referenced in the trace itself. The Multiflow compiler delays binding registers to these variables and affects them only when the variable is referred explicitly in other traces. The Multiflow scheduler uses the bindings already resolved as preferences in resolving the delayed bindings. The delayed binding carries the information necessary to determine whether these preferences can be satisfied.

13.5 Global Register Allocation

Some simple heuristic solutions exist for the global register allocation problem. For example, *lcc* [34] allocates registers to the variables with the highest estimated usage counts [31], spills a variable in a register whose next use is most distant by allocating memory on the stack and allocates temporary registers within an expression by doing a tree traversal. Another simple heuristic is the linear scan.

13.5.1 Register Allocation via Graph Coloring

Chaitin [17, 45] was the first systematic attempt to apply graph coloring to register allocation. An interference graph is constructed from the program. Each node in the interference graph represents a live range of a program data value that is a candidate to reside in a register. Informally, a live range is a collection of operations (or basic blocks in some implementations) where a particular definition of a variable is live. Two nodes in the graph are connected if the two data values corresponding to those nodes interfere with each other in such a way that they cannot reside in the same register. In addition, some machine dependences can also be modeled by including specific interferences. For example, if the result of an instruction cannot reside in a register due to nonregular registers, an interference edge can easily model it.

In coloring the interference graph, the number of colors used corresponds to the number of registers available for use. Chaitin's approach to register allocation attempts to map a fixed set of colors to all the nodes of the interference graph. If not all the nodes can be colored with the available registers, a live range is spilled (i.e., assigned a location in memory), and all references effected through LOAD/STORE instructions (spill code). The live range corresponding to the spilled variable can be deleted from the interference graph. This deletion reduces the chromatic number and we can repeat this process until the graph is colorable. However, this change to the code requires recomputing the interference graph again, an expensive operation. Spilling replaces a global node of higher degree to several local nodes of lower degree.

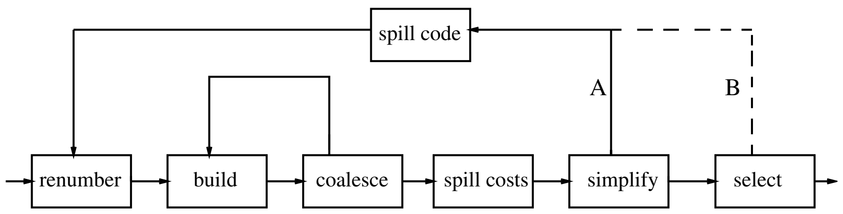


FIGURE 13.2 Steps in graph coloring: (A) the Chaitin framework; (B) the Briggs framework.

An important improvement to the basic algorithm is the idea that the live range of a temporary should be split into smaller pieces, with move instructions connecting the pieces. This relaxes the interference constraints, making the graph more likely to be *k* colorable. The graph-coloring register allocator should also coalesce two temporaries that are related by a move instruction if this can be done without increasing the number of spills.

The steps[14] in the algorithm are as follows (see Figure 13.2):

- *Renumber* (Section 13.5.2.2). This step creates a new live range for each definition point and, at each use point, unions together the live ranges that reach the use (disjoint set union-find problem). SSA can also be used; it is termed web analysis in [82]. In some simpler implementations, live ranges are discovered by finding connected groups of *def-use* chains. A single def-use chain connects the definition of a virtual register to all its uses.
- *Build interference graph* (Section 13.5.2.1). If the allocation of two live ranges to the same register changes the meaning of the program, they interfere. An interference graph specifies these relations between live ranges and this can be implemented either as a matrix or as a linked list. This phase is the most expensive; hence, some designs repeat other steps of the algorithm in an attempt to minimize the number of repetitions of this step.
- *Coalesce* (Section 13.5.2.3). Two live ranges are combined if the initial definition of one is a copy from the other and they do not otherwise interfere. Combining the two live ranges eliminates the copy instruction. Due to the change in the graph, build and coalesce steps have to be iterated until no changes occur.
- *Spill costs*. This step estimates for each live range the runtime cost of any instructions that would be added if the item were spilled. Spill cost is estimated by computing the number of loads and stores that would be required to spill the live range, with each operation weighted by $c \cdot 10^d$ (or use power of 8 on binary machines), where c is the operation cost on the target architecture and d is the instruction loop-nesting depth.
- *Simplification* (Section 13.5.2.5). This step creates an empty stack and repeats the following two steps for each live range l in graph G:

  1. If l has interfering nodes with a degree less than k, remove l from the graph G and put l on a stack for coloring
  2. Otherwise, choose a node l to spill using a heuristic such as (Chaitin's) choosing the node with the smallest ratio of spill cost divided by current degree. Remove l and all of its edges from the graph. Mark l to be spilled.

- *Coloring* (Section 13.5.2.6). Assign colors to the nodes in the stacked last-in, first-out (LIFO) order. Every live range on the stack is guaranteed to have a distinct color.

[14]We discuss some details of many of these steps in Section 13.5.2; we also provide the links here.

- *Spill code*. Spilling a live range can be done by inserting a store instruction after every definition and a load code before every use. This approach is refined by skipping the first instruction in sequential definitions and skipping second load instruction in sequential uses.

Appel [1] has a somewhat different structure with emphasis on removing move operations through coalescing.

13.5.1.1 Refinements to Chaitin's Approach

13.5.1.1.1 *Optimistic Coloring.*

Given an interference graph, Chaitin's algorithm proceeds by successively removing unconstrained vertices from the graph. This simplification approach may miss legal coloring opportunities. Consider a complete bipartite graph with M nodes on each side. Only two colors are needed but Chaitin's algorithm can spill if less than M registers are available as every node is constrained. For example, the interference graph in Figure 13.3 is colorable with 2 colors, but Chaitin's approach does not find an optimal solution for this case.

Optimistic coloring by Briggs et al. [11, 16] improves simplification by attempting to assign colors to live ranges that would have been spilled by Chaitin's algorithm. Optimistic coloring delays spill decisions until the register assignment phase. Unlike Chaitin's approach where spill candidate l is chosen when the interference graph is simplified and spilled, spill candidates get placed on the stack with all the other nodes in optimistic coloring. Only when select discovers that no color is available is the live range actually spilled. A spill is not necessary if not all neighbors have been colored with different colors. However, Lueh [76] reports that optimistic coloring is not effective in practice.

13.5.1.1.2 *Splitting.*

Another refinement fits live range splitting into Chaitin's coloring framework by splitting live ranges prior to coloring. The motivation is to reduce the degree of the interference graph and to allow the spilling of only those live range segments that span program regions of high register pressure. Aggressive live range splitting, as reported by Briggs, uses the SSA representation of a program to determine split points. A live range is split at a ϕ node when the incoming values to the ϕ node result from distinct assignments. The approach also splits all live ranges that span a loop by splitting these live ranges immediately before and after the loop. These approaches to splitting live ranges before

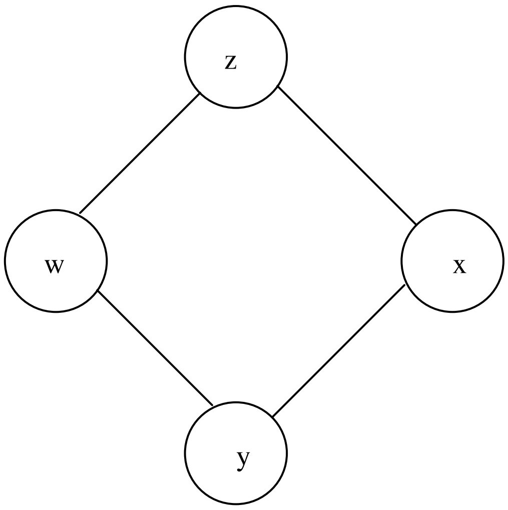


FIGURE 13.3 Chaitin requires three registers for this graph but optimistic coloring requires only two registers.

coloring have drawbacks of splitting live ranges unnecessarily. Conservative coalescing is proposed to increase the chance that the same color is given to partner live ranges. The code is traversed in any convenient order and at each copy instruction; this instruction may be deleted if source and target live ranges do not interfere and they do not make the resulting node constrained. Another conservative strategy [38] coalesces two nodes a and b if, for every neighbor t of a, either t interferes with b or t is unconstrained.

Bergner et al. [12] describe a heuristic called interference region spilling that reduces the amount of spill code necessary for spilled live ranges. Instead of spilling a live range everywhere, their method chooses a color for the live range and only spills it in areas where that color is unavailable. The allocator picks a color for the spilled live range by estimating the costs that would be incurred for each color; it selects the color with the smallest estimated cost.

Cooper and Simpson [29] introduce another method that compares splitting vs. spilling costs. Spilling a live range l_i does not break the interference with any live range l_j that is live at either a definition or a use of l_i. However, if l_i and l_j interfere but l_j is not live at a definition or a use of l_i, then l_i contains l_j and it may be beneficial to split l_i around l_j. A containment graph with live ranges as nodes is used to detect this situation: an edge from l_j to l_i in the graph indicates that l_i is live at a definition or use of l_j. l_i can be split across l_j if and only if l_i and l_j interfere and no edge exists from l_i to l_j in the containment graph. If the estimated cost of splitting is less than the cost of spilling everywhere, splitting is better than spilling. Experiments [29] show that in most cases it performs better than either the Briggs or Bergner approaches.

13.5.1.1.3 *Rematerialization.*

Rematerialization reduces the live range of a variable and hence promotes coloring. Briggs uses SSA in his implementation of rematerialization [10]. By examining the defining instruction for each value, he recognizes never-killed values and propagates this information through the SSA graph. The sparse simple constant algorithm by Wegman and Zadeck [104] is used to propagate never-killed information.

13.5.1.1.4 *Multiple Spill Heuristics.*

Because much of the time is taken in building the graph with graph coloring itself taking much less time, it is advantageous to repeat graph coloring multiple times with different costing models on the same interference graph and choose the coloring with the least cost. The spilling heuristic of Chaitin-style coloring [45] and optimistic coloring [16] use $\frac{S(lr)}{degree(lr)}$ as the priority where $degree(lr)$ is the degree lr in the interference graph. Even though the size of the live unit is a good heuristic for normalization, a large size live range is not necessarily bad.

Later work by Bernstein et al. [9] explored other spill choice functions. They present three alternative functions:

$$P(lr) = \frac{S(lr)}{degree(lr)^2} \tag{13.1}$$

$$P(lr) = \frac{S(lr)}{degree(lr)area(lr)} \tag{13.2}$$

$$P(lr) = \frac{S(lr)}{degree(lr)^2 area(lr)} \tag{13.3}$$

In the preceding equations, $area_n$ represents an attempt to quantify the impact lr has on live ranges throughout the routine:

$$area(lr) = \sum_{i \in Q(lr)} (5^{depth_i} \cdot width_i) \tag{13.4}$$

where $Q(lr)$ is a set of instructions in live range lr (i.e., instructions where a variable is live), $depth_i$ is the number of loops containing the instruction i and $width_i$ is the number of live ranges live across the instruction i. Experiments conducted by Bernstein et al. [9] show that no single spill priority function is better than others. They propose the use of a best of three technique where *simplify* is repeated three times, each time with a different spill metric, and choose the version giving the lowest spill cost.

13.5.1.1.5 *Hierarchical Allocation.*

Callahan and Koblenz [22] partition the register allocation of a function by defining a hierarchical tiling (similar to control trees [82], with each tile similar to a node in the control tree) based on the CFG. In the first phase, the tiles are colored individually in a bottom-up fashion and each variable is assigned a physical register (e.g., to satisfy linkage conventions), or a pseudo register or spilled to memory. A pseudo register is provisionally a good candidate for allocating a register but it is not possible to determine at this stage whether it can actually be allocated a register or spilled in the second phase (due to the presence of pass through live ranges that have no references in this tile). In a second, downward pass the pseudo registers are mapped to physical registers. In this pass, variables that have been allocated registers in the parent tile but not referenced in this tile are now included by making them interfere with every other variable in the conflict graph constructed in the first pass. These variables are also preferenced to use the registers that have been allocated to them in the parent tile. Spilling uses the optimistic coloring approach. The optimal location of the spill code is also addressed: it is either at the topmost possible tile or lower down if it can be pushed down due to conditional statements (see Section 13.5.2.4).

Though this approach is widely known and possibly implemented in many commercial compilers, no results are available on the effectiveness of this approach in the literature. We discuss in the next section (Section 13.5.2) a version of this approach using control trees.

13.5.1.1.6 *Graph Decomposition through Clique Separators.*

Gupta, Soffa and Ombres [94] use Tarjan's result on colorability of a graph by decomposition into subgraphs using clique separators, which states that if each subgraph can be colored using at most k colors, then the entire graph can be colored in k colors by combining the coloring of subgraphs. Each subgraph includes the clique separator. A renaming of registers for the clique separator nodes might be needed when merged with other subgraphs. In Tarjan's work, the entire interference graph is constructed and later clique separators are identified. In this work, clique separators are identified by examining code and interference graphs and then one subgraph is constructed at a time for space efficiency. A program is partitioned by selecting traces (paths) through the CFG and finding clique separators for each trace. If a variable is live on multiple traces, renaming may be necessary at branch or join points. The maximum number of cliques, chosen as separators, in which a live range can occur is fixed to a small constant (13.2). The time complexity of register allocation with m-clique is $O(N^2 \div m)$, but the overhead of determining the clique separators is larger than the benefit in register allocation time and results in longer overall time.

13.5.2 Efficient Data Structures and Algorithms for Graph Coloring

Register allocation requires careful attention to data structuring and algorithmic issues. For example, when constructing the interference graph in graph coloring, adjacency matrix is good but during the actual coloring phase adjacency lists are better.

The cost of constructing and manipulating the interference graph dominates the overall cost of allocation. On a test suite of relatively small programs [21], the cost is as as much as 65% with the graphs having vertices (N) from 2 to 5,936 and edges (E) from 1 to 723,605. N and

E are sometimes an order of magnitude larger on some graphs (especially, computer-generated procedures).

Chaitin [17, 45] suggests representing the interference graph using both an edge list for each node and a triangular bit matrix ($O(N \cdot N)$). The edge list allows for efficient examination of a node's neighbors, whereas the bit matrix ensures a constant time membership test. George and Appel [38] recommend using a hash table in place of the bit matrix because E/N for them is approximately 16 (sparse graphs). However, Cooper, Harvey and Torczon [21] report that this ratio is quite variable and that actually changes during coloring process itself. They conclude that below some threshold size, the split[15] bit matrix method is generally smaller and faster than either hashing or the original Chaitin method. Above the threshold, the compiler should use closed hashing with the universal hash function.

13.5.2.1 Interference Graph Construction

This depends on the exact definition of interference, the unit of liveness, whether predication exists and possibly also the model of scheduling.

A simple definition of interference is that two variables interfere if they are live at the same time. Assume a lower triangular adjacency matrix for the interference graph with the appropriate normalization (i.e., the first index is smaller than the second or swapped to get this effect).

```
at each instruction point I
  for each virtual register VR1 live at I
    for each virtual register VR2 live at I
      if VR1<>VR2
        set interferes(VR1,VR2) to true
        set interferes(VR2,VR1) to true
```

The complexity is $O(|I| \cdot |VR|^2)$ (can be crudely approximated as cubic in $|I|$). However, there is a better notion for interference: Consider:

```
   if (cond)
     then A=...
     else B=...
X:
   if (cond)
     then ...=A
     else ...=B
```

At X, both A and B are live but do not interfere. Hence, a better definition is the following: two names interfere if one of them is live at the definition point of the other [45]. The Chaitin definition is more accurate because use of an undefined variable is handled better.

The new algorithm for computing interferences is as follows:

```
at each instruction point I
  for each virtual register VR1 live at I
    for each virtual register VR2 modified at I
      set interferes(VR1,VR2) to true
      set interferes(VR2,VR1) to true
```

[15]That is, integer and real variables have different bit matrices.

Because the number of operands modified in an instruction is typically not more than one or is constant (especially true in RISCs), the complexity is $O(|I| \cdot |VR|)$ (can be approximated as quadratic in $|I|$).

An alternative definition [18] is: first, a live range is the intersection of the set of program graph nodes in which a variable is live and the set of nodes in which it is reaching; second, two live ranges interfere if they have a common node.

If liveness granularity is at the level of a block (BB/SB/HB), the algorithm for computing interferences is as follows:

```
compute the virtual registers live on exit from each block
for each block B
  for each instruction I in reverse order in B
    for each virtual register VR1 live at I
      for each virtual register VR2 modified at I
        set interferes(VR1,VR2) to true
        set interferes(VR2,VR1) to true
    update virtual registers live at the predecessor instruction
      by subtracting definitions in I and adding uses in I
```

It is sometimes necessary to look at intrainstruction level liveness. Consider a VLIW machine that has two types of scheduling [70]: Equal (EQ) scheduling requires that an operation take a specific duration while less than equal (LTE) requires less than or equal to some period; thus, LTE scheduling can cope with variations in time due to caches, etc. but with likely lower performance. Consider $a = a + b$, which is equivalent to $a1 = a2 + b$. In the LTE/NOT-EQ model, $a1$ and $a2$ interfere but not in the EQ model. If register allocation is to be independent of the specific scheduling model used, we need to define liveness that is based on operations instead of edges. In an edge-based liveness model, the liveness of $a2$ extends on the edge into this operation whereas liveness of $a1$ begins on the edge out of this operation; hence they do not interfere. Hence, to be independent of both styles of scheduling, we may have to model liveness at the level of operations at the cost of some accuracy.

In addition, the live ranges can be constructed recursively from sublive ranges, which can go all the way down to an operation (see Section 13.6.1 for some details). Predication is also another factor. Liveness analysis becomes predicated [62] as well as spill code. We discuss some aspects of this later (Section 13.6.1).

13.5.2.2 Renumbering

To compute the disjoint lifetimes of a virtual register, the union-find algorithm is useful. Each such disjoint lifetime can be independently allocated.

```
let D = set of all definitions of VR
for each def d in D, makeset(d)
for each use u of VR
  RD = set of defs reaching u
  select any one def d1 in RD
  S = find({d1})
  for each def d in RD - {d1}
    S = union(S, find(d))
```

The various sets at the end of the algorithm represent the new candidates for register allocation. If SSA representation is used, this is not necessary.

13.5.2.3 Coalescing

This section and the next have been adapted from an unpublished manuscript on register allocation [24, 73].[16]

It is legal to coalesce A and B if, except for the copy instruction between A and B, there are no interferences between A and B. It may not be possible to coalesce all such pairs because one pair may introduce interferences for another pair. It is also desirable to coalesce the pair that gives maximum benefit.

The following pseudo code first initializes the data structures (primarily a priority queue for each pair of virtual registers that could be potentially coalesced) and then attempts to coalesce them. Backtracking is effected if some interferences result due to coalescing of other candidates. The priority queue has four-tuple entries with the two candidates for coalescing as the first component and prioritized by the cost which is the second component.

Initialization:

```
for each instruction I
  if I is a copy instruction "dest=src"
   VR1 = find(dest)
   VR2 = find(src)
   if VR1 <> VR2
    update(PQ, [(VR1,VR2), cost, liveVRsTargetingDest,
      liveVRsTargetingSrc])
    // upd a priority queue PQ with entry E [(VR1, VR2),
    // cost (the savings if VR1 and VR2 were to be coalesced),
    // potential coalescible VRs live at copy instrs that
         target "dest",
    // potential coalescible VRs live at copy instrs that
         target "src"]
    update VR1's and VR2's potential coalescible VRs by
         adding E  to each
```

[16]An interesting informal description (taken verbatim from the abstract of [73]) is as follows:

Register allocation is like trying to fit blocks in a box. There are two distinct problems: first, the volume of the blocks may be too large for the box, and second, the shapes of the blocks may make the box difficult to pack. No matter how good your packing technique is, you cannot succeed if the volume of blocks is too large. On the other hand, that the volume of blocks is acceptable does not guarantee that the box can be packed. In the analogy to register allocation, each live range associated with a candidate (object to be allocated) is a block. Other algorithms view blocks as having uniform density. Our blocks have holes, regions in which the candidate's live range is reference-free. We prune portions of the program where the number of live candidates is greater than the number of registers (the register pressure is too high) by storing the value in memory on entry to the region and reloading it on exit. It is always possible to prune enough that the volume of the candidates does not exceed the volume of the box. In theory, once the volume problem is solved, packing is easy if one has a chainsaw: when something doesn't fit, cut it in two and try again. In practice, the problem is difficult to solve well because splitting has a cost: **move** *instructions must be inserted between the split regions. It is important to select not only the proper candidate to split but also the proper location of the split to facilitate packing the remaining blocks and also to minimize the costs of the* **move** *instructions. Both the pruning and the splitting use the control tree to take into account the program structure in determining costs. Because the algorithm uses the control tree to guide the pruning and splitting decisions, the algorithm is called control-tree register allocation algorithm*

.

```
else VR1 = find(dest)
  for each VR live at I
    VR2 = find(VR)
    set interferes(VR1,VR2) to true
    set interferes(VR2,VR1) to true
```

Coalesce:

```
for each entry E == [(VR1,VR2),*,*,*] in PQ
  if interferes(VR1, VR2) backtrack(E)
while (PQ is not empty)
  E = extract-min(PQ)
  VRu = union(VR1,VR2)
  interferes(VRu,*) setunion= interferes(VR1,*) setunion
    interferes(VR2,*)
  interferes(*,VRu) setunion= interferes(*,VR1) setunion
    interferes(*,VR2)

  update VRu's potential coalescible entries by adding
    those of VR1 and VR2
  for each potential coalescible entry E of VRu
    if E in PQ
      if E.VR1==E.VR2 then delete(PQ, E)
      else if interferes(E.VR1,E.VR2) backtrack(E)
           else if (VR1<>E.VR1) or (VR2<>E.VR2)
                                delete(PQ, E)
                                update(PQ, [(VR1,VR2), E.cost,
                                  E.Live1, E.Live2])
                else
    else delete(PQ, E)
```

Let E be [(VR1, VR2), cost, live1, live2]:

update(PQ, E)

```
if VR1 > VR2, swap (VR1, live1) and (VR2, live2)
if (VR1, VR2) in PQ, update this entry by adding to
  the corresp fields
  [cost, live1, live2]
else insert entry [(VR1, VR2), cost, live1, live2]
  into PQ return entry
```

backtrack(E)

```
vr1 = find(VR1)
vr2 = find(VR2)
for each L in live1
  vr = find(L)
  set interferes(vr, vr1) to true
  set interferes(vr1, vr) to true
for each L in live2
```

```
    vr = find(L)
    set interferes(vr, vr2) to true
    set interferes(vr2, vr) to true
  delete entry E from PQ

  for each entry E in either vr1's or vr2's potential
    coalescible VRs
    if E in PQ
      if interferes(find(E.VR1), find(E.VR2)) backtrack(E)
```

13.5.2.4 Splitting or Pruning for Colorability

Pruning involves determining the set of live ranges to be split and determining the right split points for the selected live ranges. Once a live range is split, compensation code needs to be inserted to store/load to/from memory for the uses encountered in the second part of the live range. Due to the cost of compensation code, pruning decisions need to be cost based, especially taking execution frequency into account. The following pruning technique is based on the hierarchical technique proposed by Callahan and Koblenz [22]. In this algorithm, sections of a live range with no references are identified: these potential candidate regions can be spilled to memory. The maximal length live range gap is referred to as a wedge in [24]. Nonoverlapping and maximal wedges are identified using the control tree.[17] The choice of wedges to prune is a function of the runtime cost of compensation code that would be added in the pruned region and the area of the program that would benefit from the pruning decision.

The initialization computes the excess register pressure for each control node and the estimate of the size of the lifetime of a candidate to be pruned in a control node. The pruning starts at the top of the control tree so that larger wedges are pruned first: this minimizes the compensation code and maximizes the area of the program that sees a reduction in the register pressure. At each node in the control tree the most desirable wedge is picked and removed; the various structures that keep track of pruning (e.g., excess register pressure) are then updated. If necessary, other wedges in priority order are removed next to reduce the excess register pressure. If this does not help, the children of the node are then examined for pruning.

Sometimes it is important to choose a wedge down the control tree instead of the top, for example, with an if-then-else statement (but not push it inside loops), to reduce the cost of the compensation code.[18]

Initialization

initPrune(Node N)

```
  if N a leaf of the control tree
    Livesegment(N) = Live(N)
    ExcessPressure(N) = |Live(N)| - |registers|
    for each live range L in Live(N)
      LiveSize(L,N) = 1
  else
    for each child M of control tree node N, initPrune(M)
```

[17]This is obtained by interval analysis (of data flow analysis) with minimal intervals [82].

[18]Note the similarity to shrink wrapping.

```
Livesegment(N) = set union, over each child node M of N,
  Livesegment(M)
Referenced(N) = set union, over each child node M of N,
  Referenced(M)
ExcessPressure(N) = max, over each child node M of N,
  ExcessPressure(M)
for each live range L in Livesegment(N),
  LiveSize(L,N) = sum, over each child node M of N,
  LiveSize(L,M)
for each live range L in Referenced(N)
for each child M of N
    if L not in Referenced(M) and L in Livesegment(M),
      newWedge(L,M)
```

newWedge(L,M)

```
if L is a wedge top that is preferable to push lower
  into control tree
  for each child MM of M, newWedge(L,MM)
else add L to Wedges(M)
```

Pruning

prune(N)

```
order Wedges
while (ExcessPressure(N)>0 and |Wedges(N)|>0)
  select max W from Wedges(N)
  insert ld/st as needed
  updatePressure(W,N)
for each child M of N
  if ExcessPressure(M)>0, prune(M)
```

updatePressure(W,N)

```
if N a leaf in the control tree
  delete W from Live(N)
  decr ExcessPressure(N) by 1
else ExcessPressure = 0
  for each child M of N
    if (W in Livesegment(M)), updatePressure(W,M)
if ExcessPressure(M)>ExcessPressure(N)
  ExcessPressure(N) = ExcessPressure(M)
```

13.5.2.5 Graph Coloring

To avoid changing the interference graph during simplification (a costly operation), a counter (**currentDegree**) is associated with each node that contains the number of current neighbors. If a node is spilled, this value is adjusted.

Initialization

```
for each V in interference graph
  currDegree(V) = degree(V)
  if currDegree(V) >= R
   insert(ConstrainedPQ, (V, cost(V))
  else insert(unConstrainedPQ, (V, currDegree(V))
```

Simplify

```
init Stack to empty
for i=1 ... num of vertices
  if unConstrainedPQ not empty
    V = delete-max(unConstrainedPQ)
  else V = delete-max(ConstrainedPQ)
  push V onto Stack
  adjustDegreeNeighbors(V)
return Stack
```

adjustDegreeNeighbors(V)

```
for each U in Neighbors(V)
  if search(ConstrainedPQ, U)
    decr currDegree(U)
    if currDegree(U) < R
      delete(ConstrainedPQ,U))
      insert(unConstrainedPQ, (U, currDegree(U))
    else
  else if search(unConstrainedPQ, U)
    if U not in Stack
      update(unConstrainedPQ, (U, --currDegree(U)))
```

13.5.2.6 Color Assignment

Two simple strategies are possible:[19] always pick the next available register in some fixed ordering of all registers or pick in the forbidden set of unassigned neighbors and still available. The first strategy maximizes use of lower numbered registers (may introduce unwanted false dependences) but is useful with caller-saved registers as fewer registers have to be saved/restored on call/return. With callee-registers, shrink wrapping is important (Section 13.6.5.5.2)

assignColors

```
repeat
  V = pop(Stack)
  ColorsUsed(V) = colors used by neighbors of V
  ColorsFree(V) = all colors - ColorsUsed(V)
  if |ColorsFree(V)|
    choose a free color and assign to node V
  else add V to spill list
until Stack empty
return spill list
```

[19]With region-based approaches, other strategies are possible. See Section 13.6.5.4.

Problems exist for assignment with real hardware. If the architecture has register pairs,[20] assignment of register pairs can be done before assignment of other registers. However, variables residing in register pairs may not be as important as others that may get spilled due to this phase ordering. Cost-based pruning, as discussed earlier, is necessary to minimize such spilling.

Some architectures or application binary interfaces (ABIs) require some values in fixed registers (e.g., arguments to a call). If the code generator handles this aspect, it may be suboptimal because it may load the register locally whereas register allocation may be able to handle it globally without introducing extra copies. Coalescing is important to avoid these copies. If the code generator does not handle this aspect, the virtual register VR requiring the dedicated register *r* can be made to interfere with all other VRs requiring a dedicated register other than *r*.

Some architectures (especially, VLIW) have multiple register banks. If the register banks are of different types, it is advantageous to allocate registers separately [21]. If they are of the same type, allocating from the total pool may require **moves** before use. It is also possible to add interference edges to prevent registers from a particular register bank to be allocated for a virtual register.

13.5.3 Priority-Based Coloring

An alternative form of global register allocation on a per-procedure basis via graph coloring is that of Chow and Hennessy [18]. This approach takes into account the savings accrued by a variable residing in a register vs. memory by computing the costs of loads and stores. Variables are ordered by priority based on the savings. This approach also uses the concept of live range splitting, introduced by Fabri [60], as an alternative to the spilling techniques used by Chaitin et al. [17]. A basic block is used as the unit of liveness whereas Chaitin uses the machine-level instruction as the unit. This coarse grained register allocation has a smaller interference graph and hence a faster allocation, but the result may be less efficient because a register cannot hold different values in different parts of a basic block. To mitigate this, Chow and Hennessy force the creation of a new basic block whenever the number of references crosses a limit. Another difference is that a live range is exact; if there is no benefit of putting a candidate live range in a register in a region, such a region is excluded. These include gaps that exist between the occurrences of the def-use chains. Because of these gaps, the nodes in a live range are not necessarily connected.

Before colors are assigned, unconstrained live ranges are removed from the interference graph. Unconstrained live ranges have a degree in the interference graph less than the number of registers available. Color assignment blocks occur when no legal color exists for the next live range to be colored and this can be resolved by splitting a live range. The basic purpose of splitting is to reduce the degree of a node in the interference graph by segmenting a long live range into smaller live ranges. Although the number of live ranges increases, each smaller live range usually interferes with fewer other candidates because the live ranges occupy smaller regions and the probability of overlap with other live ranges is thereby reduced.

The value of assigning a given variable to a register depends on the cost of the allocation and the resultant savings. The cost comes from the possible introduction of register–memory transfer operations to put the variable in a register and later to update its home location. Coloring greedily assigns colors to live ranges in a heuristic order determined by a priority function. The priority function captures the savings in memory accesses from assigning a register to a live range instead of keeping the live range in memory. It is proportional to the total amount of execution time savings, $S(l)$, gained due to the live range l residing in registers. $S(l)$ is computed by summing

[20] These are registers that have to be consecutive and start, say, even.

s_i over each live unit i in the live range weighted by the execution frequency w_i of the individual basic blocks:

$$S(l) = \sum_{i \in lr} s_i \times w_i \tag{13.5}$$

where a live unit is each register allocation candidate in conjunction with a basic block and s_i is:

$$s_i = \text{LODSAVE} \times u + \text{STRSAVE} \times d - \text{MOVCOST} \times n \tag{13.6}$$

where u denotes number of uses, d denotes the number of definitions and n denotes the number of register moves needed in that live unit.

The last factor to consider is $P(l)$, the size of the live range, which is approximated as the number of live units, M, in the range. A live range occupying a larger region of code takes up more register resource if allocated in the register. The total savings are therefore normalized by M so that smaller live ranges with the same amount of total savings can have higher priority. Thus, the priority function is computed as:

$$P(l) = \frac{S(l)}{M} \tag{13.7}$$

Their framework (Figure 13.4) has the following steps:

1. Build the interference graph and separate out unconstrained live ranges.
2. Repeat until no uncolored constrained live ranges exist or all registers are exhausted.
   a. Compute priorities of live ranges (using some heuristic function) if any have not already been computed.
   b. If any priority of live range l negative or if no basic block i in l can be assigned a register (because every color has been assigned to a basic block that interferes with i), delete l from the interference graph.
   c. Select the live range l with the highest priority.

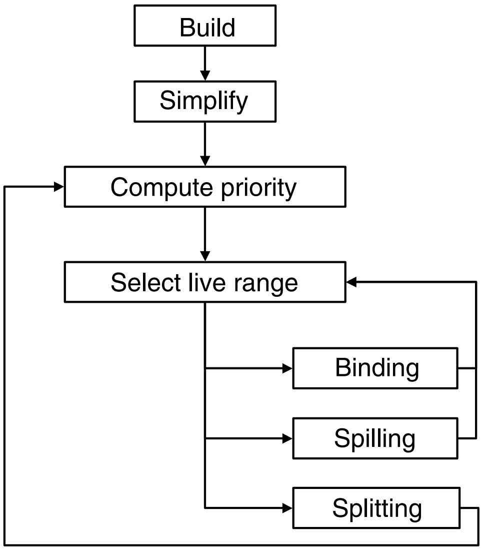


FIGURE 13.4 The framework of Chow and Hennessy register allocation.

   d. Assign a color to l not in its forbidden set and update the forbidden set of those live ranges that interfere with l.
   e. For each live range p that interferes with l, split p if its forbidden set covers the available colors.

3. Assign colors to unconstrained live ranges not in their corresponding forbidden sets.

One problem with the priority-based approach is that the priority function never changes unless it is split during the coloring process. Thus, it may fail to capture the "dynamicity" of the register-binding benefit because the degree of nodes in interference graph changes (see Section 13.6.5.3.2 in the context of region-based register allocation).

Live range construction — A live range is an isolated and contiguous group of nodes in the CFG. In the Chow and Hennessy approach, live ranges are determined by data flow analysis: those of live variable analysis and reaching definition analysis. A variable is live at block i if a direct reference of the variable exists in block i or at some point leading from block i not preceded by a definition. A variable is reaching in block i if a definition or use of the variable reaches block i. The live range $lr(v)$ of a variable v is given by $\mathit{live}(v) \cap \mathit{reach}(v)$.

Live range splitting — In splitting a live range, Chow and Hennessy separate out a component from the original live range, starting from the first live unit that has at least one reference to the variable. This component is made as large as possible to the extent that its basic blocks are connected. This has the effect of avoiding the creation of too small a live range segment. The live range splitting steps can be summarized as:

1. Find a live unit in l in which the first appearance is a definition. If this cannot be found, then start with the first live unit. Let this live unit be part of the new live range l' that is split from l. It is guaranteed that l' has at least one live unit that can be colored due to reasons in preceding step 2.
2. For each successor lu of the live units in l' in breadth-first order, add lu into l' as long as l' is colorable. Update l by removing all the live units in l'.
3. Update the interference graph and priority of l and l'.
4. If l and/or l' become unconstrained after the split, move them from the constrained pool to the unconstrained pool.

Spill code and shuffle code insertion — Spilling a live range l assumes that the value of l resides in memory and it is necessary to insert a store operation after every definition of l and a load operation before every use. The costs due to these operations are called *spill costs*.

When a live range is split into a number of smaller live segments and split live ranges are bound to different registers, *shuffle code* (also called *patch up code*, or *compensation code*) may be needed to move the data from one live range to the next. When a preceding live segment is bound to a register with the following segment spilled, a store operation is required. Likewise, if the preceding live segment is spilled and the following live segment is bound to a register, a load operation is required at the boundary. If two adjacent live segments are bound to different physical registers, a move operation is needed. In region-based register allocation, region boundaries provide implicit and natural splitting points to the compiler. In this case, the shuffle code for the region boundaries are identical to the shuffle code required in live range splitting.

13.5.3.1 Extensions.

We discuss many of the extensions (such as shrink wrapping) in the context of a region-based allocator (see Section 13.6.5). However, almost all the ideas discussed there can also be incorporated in a nonregion-based approach.

13.5.4 Linear Scan and Binpacking Approaches

The linear scan algorithm allocates registers to variables in a single linear-time scan of the variable live ranges. The linear scan algorithm is considerably faster than algorithms based on graph coloring, is simple to implement and results in a code that is almost as efficient as that obtained using more complex and time-consuming register allocators based on graph coloring. In one study [83], it is within 12% as fast as code generated by an aggressive graph coloring algorithm for all but 2 benchmarks. The algorithm is of interest in applications where compile time is a concern, such as dynamic compilation systems, JIT compilers and interactive development environments.

At each step, the algorithm maintains a list, *active*, of live intervals that overlap the current point and have been placed in registers. The active list is kept sorted in order of increasing end point. For each new interval, the algorithm scans the active list from beginning to end. The length of the active list is at most R. The worst-case scenario is that the active list has length R at the start of a new interval and no intervals from the active list are expired. In this situation, one of the current live intervals (from the active or the new interval) must be spilled. Several possible heuristics exist for selecting a live interval to spill. One heuristic is based on the remaining length of live intervals by spilling the interval that ends last, farthest away from the current point. This interval can be found quickly because active is sorted by increasing the end point: the interval to be spilled is either the new interval or the last interval in the active list, whichever ends later. In straight-line code, and when each live interval consists of exactly one definition followed by one use, this heuristic produces code with the minimal possible number of spilled live ranges [15, 81].

Another linear scan algorithm, the second-chance binpacking [97], invests more work at compile time to produce better code. It is a refinement of binpacking, a technique used in the Digital Equipment Corporation (DEC) GEM optimizing compiler [8]. The registers are the bins and the lifetimes of the virtual registers are the ones to be packed into the bins with the constraint that no overlapping lifetimes be packed into the same register. When a virtual register v is first encountered and no free registers are available, a spill candidate s is found based on a heuristic (farthest distance, loop depth, etc.) and its live range split at this point. The live range of v starts from here until it is victimized by another. If s is next encountered (read or write), another spill candidate is found and its live range split. The allocation and code generation are all done in a single linear pass inside a basic block; hence, this method is suitable for JIT compilers. However, at basic block boundaries, allocations have to be resolved again with a traversal of the CFG edges (an extra pass). The basic blocks in this approach are the regions in a region-based approach and hence many of the performance problems with an unoptimized region-based register allocation remain (see Section 13.6.4).

In the GEM system, the allocator uses a density function to control the allocation. A new candidate can displace a previous variable that has a conflicting lifetime if this action increases the density measure. After the allocation of temporaries to registers is completed, any unallocated or spilled temporaries are allocated to stack locations.

13.5.5 Integer Linear Programming Approaches

In some situations, it is important to exhaustively and systematically check (or close to it) all solutions due to irregularities in the architecture or in the problem domain by formulating the problem as an integer linear program (ILP) and solve it exactly with a general-purpose ILP solver. The Intel IA-32 architecture (e.g., Pentium), due to its dominance in the marketplace, is an example of an irregular architecture for which good register allocation is important. Even though eight registers exist, it has only six allocable registers because usually two are dedicated to specific purposes. However, many graph coloring register allocation algorithms do not work well for machines with few registers. For example, on a test suite of 600 basic block clusters comprising 163,355 instructions, iterated register coalescing produces 84 spill instructions for a 32-register machine, but 22,123 spill instructions for an 8-register machine (14% of all instructions) [2]. In addition, heuristics for live range splitting may be complex or suboptimal, and heuristics for register assignment usually do not take into account the presence of some addressing modes. These problems are aggravated when only a few registers are available.

The ILP problem is NP-complete, but approaches that combine the simplex algorithm with branch-and-bound can be successful on some problems. The first significant work is that of Goodwin for his Ph.D. thesis [37, 49]. Goodwin and Wilkin [49] formulate global register allocation as a 0-1 integer programming problem incorporating copy elimination, live range splitting, rematerialization, callee and caller register spilling, special instruction–operand requirements and paired registers. Based on the CFG of a program, a symbolic register graph is derived for each virtual register, each edge is labeled with a decision variable that when solved by the optimal register allocator (ORA) solver module indicates whether the virtual register is allocated (1) or not (0) to a real register at the point represented by the edge. Because ORA can take hundreds of seconds to solve for a large procedure, Goodwin and Wilkin [49] also have formulated near-optimal register allocation (NORA) as an ILP: one version (profile-guided hybrid allocation) does ORA on performance critical regions identified through profiling and graph coloring on the rest. A ORA-GCC hybrid takes an average of 4.6 sec to produce an allocation that is within 1% of optimal for 97% of SPEC92 benchmarks. GCC takes, however, 0.04 sec on an average.

Appel and George [2] take a different approach to near-optimal register allocation through a two-phase approach that decomposes the register allocation problem into two parts: spilling and then register assignment. Instead of formulating the ILP problem as whether variable v is in register r at program point p, a simpler formulation is whether v is in register or memory at program point p. This phase of register allocation finds the optimal set of splits and spills with the optimality criterion the dynamic weighted loads and stores but not register–register moves. In addition to computing where to insert loads and stores to implement spills, it also optimally selects addressing modes for CISC instructions that can get operands directly from memory. They report that it is much more efficient than the Goodwin and Wilken ILP-based approach to register allocation ($N^{1.3}$ vs. $N^{2.5}$ empirically) with many allocations within tens of milliseconds typically. They use a variant of Park and Moon optimistic coalescing algorithm that does a good (though not provably optimal) job of removing the register–register moves. Their results show the Pentium code that is 9.5% faster than code generated by SSA-based splitting with iterated register coalescing. It should be noted that the phase ordering (spilling followed by register assignment) in this approach can introduce suboptimal results but it is considerably faster.

13.5.6 Register Allocation for Loops

Hendren et al. [53] propose the use of cyclic interval graphs as a feasible and effective representation that accurately captures the periodic nature of live ranges found in loops. The thickness of the cyclic interval graph captures the notion of overlap between live ranges of variables relative to each particular point of time in the program execution. A new heuristic algorithm for minimum register allocation, the fat cover algorithm and a new spilling algorithm that makes use of the extra information available in the interval graph representation are proposed. These two algorithms work together to provide a two-phase register allocation process that does not require iteration of the spilling or coloring phases. They next extend the notion of cyclic interval graphs to hierarchical cyclic interval graphs and outline a framework for a compiler to use this representation when performing register allocation for programs with hierarchical control structure such as nested conditionals and loops.

Rau et al. [93] consider register allocation for software pipelined loops but without spilling. The problem is formulated as the minimization of the number of registers required to pack together the space–time shapes that correspond to live ranges of loop variants across the entire execution of the loop.

Callahan, Carr and Kennedy [30] use a source-to-source transformation (scalar replacement) on various loop nest types to find opportunities for reuse of subscripted variables and replace them by scalar references so that the likelihood increases that they can reside in registers. Because register pressure can be increased substantially, they try to generate scalars so that register pressure is also minimized. This is modeled as a knapsack optimization problem.

13.5.7 Interprocedural Register Allocation

Given register requirements for each procedure, interprocedural register allocation attempts to minimize execution cost. The simplest approach does not use any interprocedural register allocation and spill registers that might be used by both the caller and callee [19].

Wall [100] uses the fact that two procedures that are not simultaneously active can share the same registers for their locals and groups locals that can be assigned a common register. The locals of a procedure are in different groups than those of its descendants and ancestors in a call graph. Also, each interprocedurally shared global is placed in a singleton group. Groups are then allocated registers based on the total frequency in which their members are referenced. However, such an allocator may not be optimal because locals that are infrequently referenced can be grouped with locals frequently referenced.

Steenkiste and Hennessy [95] allocate registers to locals in a bottom-up fashion over the call graph for Lisplike languages that tend to have small procedures and spend much of their time in the leaf procedures. While registers are available, a procedure is assigned registers that are not already assigned to its descendants in the call graph. When the registers are exhausted, they switch to an intraprocedural allocation. However, this approach may introduce register spilling around calls in frequently executed procedures near the top of a call graph.

Santhanam and Odnert [96] perform interprocedural register allocation over clusters of frequently executed procedures. Spill code is attempted to be minimized by moving it to the root node of a cluster.

The Kurlander and Fischer approach [67] avoids register spilling across frequently executed calls and examines the entire call graph to generate a minimum cost allocation spilling register. They describe profile-based save-free as well as spilling version of their interprocedural register allocation. In the former, no spills are generated across calls (possible only for acyclic call graphs); the cost of allocating registers to procedures is modeled and a minimum found. The latter minimizes the allocation plus spill cost.

In save-free register allocation, interferences are modeled between candidates in the same procedure and between candidates in separate procedures connected along a path in the acyclic call graph. An *antichain* is the set of nodes that do not interfere. With k registers, the problem now becomes that of finding a maximum weight k-antichain sequence. This is solved using dual minimum cost flow [66].

In the spilling version, interferences are only modeled between candidates in the same procedure because spilling can potentially make the other candidates in separate procedures reside in the same register. The optimization problem is solved using dual minimum cost flow again. On SPEC92 benchmarks, they report an improvement between 0 to 11.2% (whereas the Steenkiste and Hennessy approach is between -3.2 and 7.4%) with an increase in compilation time between 0.2 and 5%.

13.5.8 Region-Based Approaches

A major premise of recent VLIW/EPIC architectures is the use of frequency-based approach for getting performance. It is advantageous to allocate optimization resources on code that take most of the execution time. Functions and procedures reflect the structuring of a program as conceived by a programmer but they typically have no bearing on identifying parts of the code that is most often executed. A region-based approach, where regions are identified through execution frequencies, can be used to identify important fragments of code. Once important regions are identified, two approaches can be followed. In the first approach, important regions are compiled first so that they do not have many of the constraints that later compiled regions may have. In the second more involved approach, differential compile time budgets are made available to regions depending on the importance of the regions. We discuss the first approach only because almost no experience exists yet with the second approach.

Regions can be smaller than a function and totally contained within it or they can be across functions. We consider only the former here. In the first case, execution time may suffer because optimizations have a more limited scope than their global counterparts and additional code at the boundary of two regions may have to be introduced. However, the compilation time can be lower because the time complexity of many components of a register allocation algorithm are superlinear. Hence, splitting a function into many parts results in a lower compilation time overall.

We briefly compare the traditional function-based approach (also called *horizontal model of compilation*) with region-based compilation approach (also called *vertical model of compilation*). For details, see [50, 54]. Traditionally, the compilation process has been built assuming functions as a unit of compilation. Figure 13.5 illustrates the process of compiling a program of *n* functions. However, the process suffers from the following limitations: the scope of compilation and optimization is limited to a procedure and not across critical paths spanning procedures; it lacks control of compilation unit size; and it is not able to reuse analysis information and has restricted use of profile information.

The drawbacks of function-based compilation can be addressed by allowing the compiler to repartition the program into a new set of compilation units called *regions*. A region is defined as an arbitrary collection of basic blocks selected to be compiled as a unit. Each region may be compiled completely before compilation proceeds to the next region. In this sense, the fundamental mode of compilation has not been altered and all previously proposed function-oriented compiler transformations may be applied. The benefits are the use of dynamic behavior of a program and control over the size of compilation unit.

13.5.8.1 Issues in Region-Based Compilation.

Region selection — A region-based compiler begins by repartitioning the program into regions. The region selector may select one region at a time or it may select all regions *a priori*, before

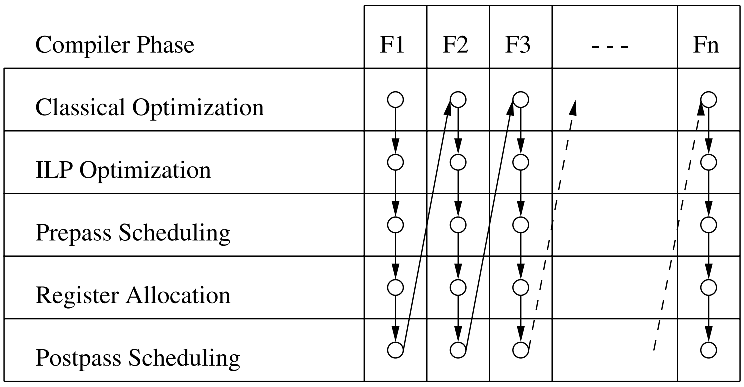


FIGURE 13.5 Block diagram of function-based compilation.

the compilation process begins on any region. The following criteria can be used for the region selection:

- Execution frequency information of operations
- Memory access information
- Structure of region (e.g., basic block, superblock [56] and hyperblock [80], loops)
- Maximum size of the region (to limit the complexity of region processing) [101]

Region selection can be performed on a single procedure or it can span more procedures. The scope of region selection can also be increased beyond a procedure by performing function inlining before region selection.

Phase ordering — In region-based compilation, the phases of compilation can be carried out on the regions in any order. Only the constraints arising due to the properties of the phases themselves should be obeyed. Different regions can be in two completely different phases of the compilation process at the same time. This approach is very flexible in comparison with the function-based approach in which each phase of compilation is applied to every basic block in the function before the next phase begins.

Incremental analysis — Each compiler phase performs certain global analysis on all the regions before starting the processing of individual regions. However, as the compilation proceeds, this information might be corrupted in certain cases and would require to be updated. This requires some form of incremental analysis.

Region boundary conditions — Separate compilation of a program using a traditional function-based compiler is facilitated by the fact that the boundary conditions of a function are fixed. The variables live across the single entry point and single exit point of a function are well defined by the parameter passing convention. However, a region is an arbitrary partition of the program CFG. The liveness and other necessary information is required at the boundary of these regions. This information can also change dynamically as the compilation proceeds.

13.5.8.2 Various Region-Based Approaches in the Literature.

We now review some of the work on region-based approaches to register allocation. We already have discussed register allocation at trace level in the Multiflow [28] compiler. One main weakness is that groups of traces that are executed together frequently are not handled, which is handled better by the notion of hyperblocks. Soffa, Gupta and Ombres [94] (Section 13.5.1.1) also use traces as regions but focus on decomposition of the graph to reduce compilation time. Though traces use frequency information, the clique separators (and hence the regions for register allocation) are determined by the exigencies of when live ranges start and end instead of groups of traces that are executed together frequently.

We have also discussed the work of Briggs [16] and Callahan and Koblenz [22]. Both of these, in the context of Chaitin's framework, consider many issues similar to a region-based approach but not in a explicit region-based framework. The structure of the program graph as needed by register allocation is derived through the use of SSA, or tiling or control trees.

Norris and Pollock [86] use the program dependence graph (PDG) representation of a program to decide the regions but such region formation is not sensitive to actual patterns of execution (though useful as a hierarchical organization of the program). For a given node, each subset of its control dependences that is common with those of another node is factored out and a region is created to represent it. Register allocation is performed hierarchically bottom up also and Chaitin's algorithm is used in each region. The hierarchical and PDG region-based approaches use the program structure and tend to allocate registers to the highest frequently executed regions first, because leaf nodes of such regions tend to correspond to the innermost loop bodies. However, the PDG region formation is not sensitive to actual patterns of execution and frequencies of execution of the region boundaries where patch codes are inserted. In branch-intensive code, the regions formed typically may not be any larger than a basic block, on the order of one C statement, resulting in unnecessary spill code [50]. The authors report a 2.7% improvement over a standard global register allocator.

Proebsting and Fischer use probabilistic register allocation [89], which is a hybrid of the priority-based and program structure-based approaches. The approach consists of three steps, local register allocation, global register allocation and register assignment. The global register allocation step partitions a program into regions based on the loop hierarchy and proceeds from the inner most loops to the outermost loops. When a variable is assigned to a register, the shuffle codes are placed at the entry and exit points of a loop. Like previous structure-based region formations, this is not sensitive to actual patterns of execution and frequencies of the region boundaries.

Hanks [50] explores region-based compilation in the IMPACT compiler. The compiler selects a region and performs classical global optimizations, instruction level parallelism (ILP) optimization,[21] instruction scheduling and register allocation.

Lueh and Gross [74] consider region-based register allocation for arbitrary regions. Lueh [77] uses graph fusion in his fusion-based register allocation. His approach starts with an interference graph for each region of the program where a region can be basic blocks, superblocks, loop nest or some combinations of these. Regions are merged in a bottom-up manner by the order of frequency of edges between regions, and the interference graph is fused whereas fused live range is simplified. Shuffle code is likely to be inserted at less frequently executed points. The split point is at region boundaries and it does not split large superblocks or hyperblocks, even if the register pressure is very high inside a block.

Hansoo and Gopinath [52] and Kim et al. [69] have considered a region-based approach that also handles predication for a research compiler system called Trimaran [98]. We first present a simplified version of this register allocator in Trimaran and then discuss in some detail refinements necessary to make it competitive with global schemes.

13.6 Region-Based Register Allocator in Trimaran

Trimaran has a region-based register allocator [68] with the following features: frequency-based priority coloring (in the spirit of Chow and Hennessy [18]), region-based with region reconciliation, fine-grained live ranges with predicated instructions and live ranges. See Appendix A for a description of Trimaran's compiler infrastructure.

13.6.1 Region-Based Register Allocation in the Chow and Hennessy Model

The approach in Trimaran builds on the framework of Chow and Hennessy (Figure 13.4) insofar as a region itself provides natural live range splitting in a region-based approach, and frequency information used in region formation can also be used in the priority function. By using this framework, regions such as hyperblocks that contain predicated instructions and superblocks are also handled in addition to basic blocks. However, the predicated instructions in hyperblocks [80] pose many interesting problems to many phases of the compiler including register allocation.

13.6.1.1 Live Range Construction

Chow and Hennessy use a basic block as the unit of liveness or coloring for faster compilation. Instead, it is also possible to use a finer granularity at the level of operations. A coarse interference

[21]From this point in the chapter ILP refers to instruction level parallelism; we have used ILP for integer linear programming earlier.

TABLE 13.1 Decrease (dec) in Percentage of Number of Edges (Rounded to 1000's) in Interference Graph with Basic Block (BB) or Hyperblock (HB) as Region

Prog.	BBc	BBf	dec%	HBc	HBf	dec%
espresso	773	678	12.2	3078	1734	43.7
eqntott	91	85	5.9	368	270	26.5
sc	270	223	17.3	513	364	29.1
gcc	4654	4268	8.3	7278	5633	22.6
m88ksim	368	284	22.7	852	504	40.9
compres	34	32	5.0	215	156	27.7
li	55	44	21.4	124	81	34.4
ijpeg	846	659	22.1	2898	1654	42.9
cccp	175	163	6.7	456	373	18.1
cmp	3	3	6.1	25	14	45.0
eqn	93	80	13.4	206	151	26.4
lex	232	218	6.2	923	746	19.2
tbl	258	248	4.1	561	425	24.2
yacc	161	149	7.2	1263	925	26.7

Note: c refers to coarse live ranges (i.e., BB or HB as unit of live range); f refers to fine grain live ranges (i.e., operation as unit of live range).

graph may result in less efficient allocation but the penalty of coarse-grained live ranges is small if the size of basic blocks is typically small.

For region-based compilation using hyperblocks or superblocks, this penalty can be greater for two reasons. First, the register pressure in a basic block with coarse-grained live ranges is not as large as in hyperblocks because the size of a basic block is usually smaller. Second, a coarse-grained live range may have extra interferences in superblocks or hyperblocks not present in basic blocks. Let variable x be live in blocks $B1$ and $B3$ only. Hence, it does not interfere with variable y that is live in $B2$, even if we use a basic block as the unit of a live range. However, if we create a superblock $SB1$ from blocks $B1$ and $B2$, a coarse-grained live range construction reports interferences between x and y.

By constructing live ranges recursively from sublive ranges, which can go all the way down to operation level, one can study the effect of granularity of live ranges for regions such as basic blocks and hyperblocks. Table 13.1 from [55] shows the difference in number of interference edges between basic block based live ranges and operation based live ranges. The reduction in interference edges with fine-grained live ranges makes for faster register allocation and hence a reduction in compilation time.

13.6.1.2 Interference Graph Construction

In Chow's approach, two live ranges are said to interfere if the intersection of their live ranges is not empty. However, Chaitin's definition of liveness is more accurate. Two live ranges interfere if one of them is live at the definition point of the other. This is necessary because live units can be as small as an operation. With predicated code, the computation of accurate liveness information is necessary to build the sparsest possible interference graph because it may have a role in reducing compilation time, as well as in better execution performance. Otherwise, an increase in the register requirements is possible. First, without knowledge about predicates, the data flow analysis must make conservative assumptions about the side effects of the predicated operations. Second, the solutions of the data flow analysis rely heavily on the connection topology among basic blocks in the flow graph, which is altered by the if-conversion process used to construct hyperblocks.

However, the performance improvement due to predicated analysis depends on many aspects. For example, hyperblock formation[22] is not undertaken if nested inner loops exist inside the selected region. Thus, only programs without inner loops but with many conditional branches and many local variables to be colored can benefit from predicate-aware liveness analysis. Also, the hyperblock construction maximizes the potential of instruction scheduling and this may disable many of the careful interference calculations of the predicate-aware register allocation. For example, if a branch is heavily weighted in one direction, the block on the opposite side may not be included in the hyperblock, so as to maximize speculation. Finally, if a block contains a function call with side effects, the block cannot be included in a hyperblock, because this prevents code motion and instruction scheduling suffers.

Other aspects such as coloring and splitting live ranges are broadly similar to the design in Chow and Hennessy but different priority functions may be appropriate, as discussed later in Section 13.6.5.3. In region-based register allocation, region boundaries provide implicit and natural splitting points to the compiler. In this case, the patch code for the region boundaries is similar to the shuffle code required in live range splitting. With predication, shuffle code also has to be predicated.

13.6.2 Simplified Overview of Trimaran Register Allocator

In the following sections, we give only the details that are relevant to understanding the basic region-based algorithm and avoid other details. The pseudo code is therefore highly schematic.

13.6.2.1 Live Range Construction

The nodes in a live range can be operations, basic blocks, superblocks or hyperblocks in the program. The construction of live range uses data flow analysis for liveness and reaching definitions within a region. However, here the analysis also takes predicates of operations into consideration. The live range is thus a list of ⟨**Operation**, **Pred_cookie**⟩, where the **Operation** is the operation in which the operand is referenced and **Pred_cookie** is the predicate expression associated with operation. The live range also stores additional information, such as interfering live ranges, forbidden registers, reference operands of definitions and uses and priority information that is required for register allocation.

13.6.2.2 Classification of Live Ranges

- *A live ranges*. These are strictly local to the region. These are register allocated by an intraregion register allocator and do not require global reconciliation.
- *Bin/Bout live ranges*. These have VRs that are live-in and live-out from or to other regions (and may be at the same time both live-in and live-out but they cannot be pass through). These are register allocated by an intraregion register allocator, but their bindings must be propagated for global reconciliation.
- *C live ranges*. These are the transparent ones. They are resolved on the basis of the binding of other adjacent global live ranges.

All *A*- and *B*-type live ranges are resolved (bound or spilled) during intraregion register allocation. All bindings for *C*-type live ranges are delayed (i.e., decided at a later time when bindings from adjacent regions become available).

[22]This refers to the front-end (IMPACT) part of the Trimaran compiler system.

13.6.2.3 Intraregion Register Allocation

```
driver(Region R) {
   for each subregion SR in R by frequency order
       if SR is base intra-region
           intra_region_register_allocation(SR)
       else driver(SR)

       resolve_inter_region_live_ranges(SR)
       mark_callee_used_registers(SR)

   reconcile_inter_region_live_ranges(R)
   reconcile_callee_used_registers(R)
}

intra_region_register_allocation(Region R) {
    live-range-set LRS = construct-live-ranges(R)

    for each filetype F
        live-range-set LRS_F = live ranges of filetype F in LRS

        construct-interference-graph(LRS_F)
        compute-priority(LRS_F)

        for each live-range LR in LRS_F by priority order
            if LR is pass-through, delay it
            else if suitable reg REG is found, bind LR to REG
            else if spilling is suitable, spill LR
            else split LR
}
```

Separate register files exist that correspond to different file types. Thus, the live ranges belonging to different file types cannot interfere. Hence, we can process the live ranges belonging to a single file type one at a time.

First, the interference graph is formed and the priorities are assigned to the live ranges. The unconstrained live ranges are separated and they are processed only after all constrained live ranges. The constrained live ranges are visited in the priority order. Live ranges are bound or spilled depending on the suitable cost function and the availability of registers.

13.6.2.4 Interregion Register Allocation

After all the *A*- and *B*-type live ranges are resolved, we resolve the interregion live ranges shown as follows:

```
resolve_inter_region_live_ranges (Region R) {
    for each non-intra live range LR of R
       add LR to global_LR

       if LR is a pass-through live range
           if all adjacent live ranges of LR in global_LR
               are delayed
               delay LR
```

```
            else
               resolve LR using most suitable adjacent live range
               adj_LR of LR in global_LR

        propagate_bindings(LR)
 }

 propagate_bindings(LiveRange LR) {
     if LR is not pass-through live range
         for all adjacent live ranges adj_LR of LR in global_LR
            Edge E = edge corresp to LR and adj_LR

            if def of value (corresp to LR) in LR or its ancestors,
                SRC = LR
                DEST = adj_LR
            else
                SRC = adj_LR
                DEST = LR

            if  SRC is spilled and DEST is spilled
                no patchup code
            else if SRC is spilled and DEST is bound
                add code for load on E
            else if SRC is bound and DEST is spilled
                add code for save on E
            else if SRC is bound and DEST is bound
                add code for move on E

            if code added to E, add E to edges_with_unordered_ops
 }
```

global_LR is a set of all interregion live ranges and is used to find the adjacent global live ranges. The bindings of *B*-type live ranges are propagated to the other adjacent global live ranges. Because the *C*-type live ranges are not resolved in intraregion register allocation, we first try to resolve them. A *C*-type live range is resolved if at least one adjacent global live range is already resolved; otherwise it is marked as delayed.

The propagation of binding consists of adding the requisite patch up information between the current live range and all its global adjacent live ranges. The patch up information is added on the edge joining the two live ranges. *edges_with_unordered_ops* is the list of all the edges on which the patch up information is stored.

After all the regions are processed, the patch up code is generated using the patchup information stored on the edges earlier. In the new basic blocks created, stores should be put before the loads. Also a circular set of moves can be between registers in the code in a new basic block; a register has to be spilled to break this cycle.

```
 reconcile_inter_region_live_ranges (Region R) {
     for each Edge E in edges_with_unordered_ops
         if source operation of E is a dummy branch
            add patch_up code in the region containing source
            operation of E
```

```
          else if destination operation of E has only one
              incoming edge
              add patch_up code in the region containing
                destination
              operation of E

          else
             create a new basicblock corresponding to E and
               add the patch_up
             code in this new block
 }
```

13.6.2.5 Reconciliation of Callee-Saved Registers

If the callee-saved registers are used by a procedure, it must save and store these registers at its entry and exit, respectively. The *mark_callee_used_registers()* marks all the callee-saved registers used by the register allocation. The reconciliation code to these registers is added to the epilogue and prologue regions of the procedure in *reconcile_callee_used_registers()*.

13.6.3 Region-Based Compilation and Callee and Caller Cost Models

Many standard assumptions need revision with region-based register allocation when addressing machines with caller-saved registers and callee-saved registers. We first consider caller and callee cost models. Caller registers do not pose major difficulties: the cost of save and load of a caller-saved register around function calls is absorbed by the live range in the region.

There are many ways to assign the cost of callee save and restore to the live ranges that use a callee-saved register. For example, consider a region that is a function. The first user of a callee-saved register can bear both costs of save and reload and cost the rest of them as if allocating a caller register [18]. Alternatively, the first user of a callee-saved register bears the save cost and the last user bears the reload cost, but this has not been reported in the literature. Lueh [76] reports another model that all users of a callee-saved register bear the costs. When the color assignment phase finishes, his approach spills all live ranges that were bound to callee-saved register e if:

$$\sum_{lr \in \delta(e)} spill\_cost(lr) < callee\_cost(e)$$

where $\delta(e)$ is the set of live ranges that were bound to e. It has been reported that this model gives incremental performance benefits on some benchmarks compared with the first one but has no perceptible difference in some benchmarks [74].

Now consider a region that is completely within a function. Because we process regions based on frequency information, one possibility is that the first live range that uses a callee-saved register in a region with the highest weight in the procedure bears all the costs. If $spill\_cost(lr) > callee\_cost(e)$, then we can definitely allocate a callee register to the live range in the region (R). Another model is if:

$$\sum_{lr \in \delta(e)_R} spill\_cost(lr) > callee\_cost(e)$$

$$\delta(e)_R = \{lr_i | lr_i \text{ is the live range that were bound to } e \text{ in region } R\}$$

then we can allocate a callee register to the live range in region. Otherwise, we are in a difficult situation because with our current region based vertical model of compilation, we are not in a position to know about the live ranges in other regions (even adjoining ones in the same function), as they have not yet been processed. However, we use the heuristic that live ranges in regions with bigger weights can absorb all the callee costs due to the larger number of accesses in them.

Next, consider a region across more than one procedure. The live ranges can now be possibly across functions. Allocating a caller register on one side of the function implicitly results in splitting the live range on a function call whereas allocating a callee register may or may not result in splitting the live range. It cannot result in splitting the live range if the callee register is not used in the called function. It results in splitting if it is used in the called function but the location of the splitting depends on whether shrink wrapping has been used. If shrink wrapping is used, the original live range is kept as long as is feasible. Again, assigning callee costs to various live ranges in this case is even more difficult due to the vertical model of compilation. However, it seems best to allocate callee-registers to live ranges that straddle functions when shrink wrapping is used.

One way out of these problems is to deviate slightly from the vertical model of compilation and first process all regions only to construct live ranges. In that case, we can compute the spill costs of all variables beforehand and use that to decide when to allocate callee registers by using the same formula as before, keeping in mind the function boundaries. However, many current systems can only handle limited types of regions.

Another model that needs revision is the allocation of caller registers to live ranges in leaf procedures. In function-based register allocation, caller registers are preferred for leaf procedures. As a heuristic, we can use the same concept for regions also: if no function calls exist in the region, it is a *leaf*. Strictly, all the regions have to be checked to see whether a function is a leaf but because of the frequency-based order for processing the regions, live ranges in regions of high weight with no function calls can be allocated caller registers and this preference is propagated in the reconcile part of the algorithm to other less frequently executed regions.

13.6.3.1 Optimization Formulation

The caller and callee cost functions should be accurate enough that these can be negative also (i.e., it is cheaper to spill than to bind to a register even if registers are available). The model can be local (considering the region alone); consider only immediate neighbors that have been processed or all connected regions that have been processed. Because we are considering pass-through live ranges, the benefit is negative — the difference of weighted cost of spill code (= 0) and caller and callee cost. For a caller register bound to a pass-through live range, if we assume local cost model, the benefit can be zero (no function calls) or negative (function calls need st/ld bracketing). If immediate neighbors are considered, the reconciliation cost (shuffle code) comes into picture on the edges incident on the region but they cannot in all cases be known due to the vertical model of compilation.

Similarly, for a callee register, it involves store and load costs weighted by frequency in a local model for the first live range using the register. Later live ranges use it at no cost (similar to [18]); or all the live ranges in the region that use the register can share this cost.

Let the number of variables be n. Let $b(k, i, j), k \in \{ER, EE\}$, be the benefit of allocating a caller-saved register (*ER*) or callee-saved register (*EE*) to the ith variable for its jth segment of the live range. This is equal to the difference of spill cost and caller and callee cost. Let $c(k, i, j)$ be the cost of compensation code (weighted by frequency) for the jth live range segment of the ith variable in all the adjacent live regions. As each variable can be split during register allocation, let the number of split segments of a live range for a variable i be $L(i)$.

The callee cost model can be complex and a register allocation with weighted regions has to use heuristics. Let *Ncaller*, *Ncallee* be the number of available caller and callee registers in the architecture. Let $x(k, i, j) = 1, k \in \{ER, EE\}$ if the ith variables jth segment is allocated to a caller and callee register. Otherwise, set it to zero. Then at each minimum unit of live range:

$$x(ER, i, j) + x(EE, i, j) <= 1 \; \forall i \in \{1, \ldots, n\}, j \in \{1, \ldots, L(i)\}$$

$$\sum_{i:1,\ldots,n} x(ER, i, j) <= \; Ncaller \; \forall \; j \in \{1, \ldots, L(i)\}$$

$$\sum_{i:1,\ldots,n} x(EE, i, j) <= \; Ncallee \; \forall \; j \in \{1, \ldots, L(i)\}$$

$$x(k, i, j) \; \in \{0, 1\} \; \forall \; i \in \{1, \ldots, n\}, j \in \{1, \ldots, L(i)\}, k \in \{ER, \ldots, EE\}$$

The optimization problem is to maximize:

$$\sum_{i:1,\ldots,n, j:1,\ldots,L(i), k \in ER,\ldots,EE} x(k, i, j) \cdot (b(k, i, j) - c(k, i, j))$$

while minimizing $L(i)$ for each i. First, $L(i)$ and $b(k, i, j)$ cannot be computed straightforwardly because the priority depends on the spill costs, caller(callee) costs and the current set of compensation costs, but the act of splitting changes the costs and introduces new live ranges. A combinatorial or iterative method is needed for an approximate solution. Luckily, the priority function prioritizes the live ranges and ensures that any costs are borne by less important live ranges processed later: this happens as the splitting comes into play only after allocation of all available registers to higher priority live ranges and some or all the remaining interfering live ranges getting split. (All pass-through live ranges have already been set aside as candidates for coloring by the design of the algorithm in core register allocation.) Similarly, $L(i)$ is minimized for the important live ranges.

Second, as $c(i, k, ENDS)$ (where *ENDS* refers to segments of live ranges that are adjacent to nearby regions) cannot be determined unless other regions have also been processed by this time, we need again to use some combinatorial or iterative method for an exact or approximate solution. To simplify the problem, one can ignore $c(k, i, ENDS)$ (set it to zero) where not known and then solve. This has the effect of incorporating costs of reconcile code only in later regions; this means that live ranges in regions with bigger weights are given more flexibility in selecting the type of register needed and later live ranges in other regions are "requested" to use the same type of register.

With the preceding heuristic, we can solve the preceding problem by ordering all $b(k, i, j) - c(k, i, j)$ in descending order (using the value of c where known and zero otherwise) and pick only $Ncaller + Ncallee$ of them at each minimum unit of live range while taking care to include only one caller or callee for each variable segment and picking only *Ncaller* and *Ncallee* registers. Other heuristics are possible: for example, compute the difference of caller or callee costs and prioritize [77].

13.6.4 Performance Problems in a Basic Region-Based Register Allocator

Although region-based register allocation typically has a lower compilation time compared with global register allocation, it has a potential weakness because of the limited scope for optimization with region types that are totally contained within function boundaries. Although we want to reduce compilation time as much as possible, we want execution performance to be comparable to a global (within a function) algorithm. With this mind, one can study ways of improving execution performance of a region-based register allocator while preserving the compile time advantages. One prime technique is the propagation of carefully selected information across regions to circumvent the limited scope while compiling a region. It has been shown that the goal can be achieved with savings of about 20 to 50% in compilation time whereas execution time is lower or only a few percentage points higher than the global algorithm [53]. The same study also provides some quantitative data to characterize the relationship between region size and performance (execution and compile time). Based on these data, region restructuring using frequency of execution and register pressure can be attempted.

13.6.4.1 Compilation Time Advantage and Execution Penalty with Region-Based Compilation.

To make region-based compilation practical, we need to reduce compile time as much as possible without increasing the execution time. If the size of intermediate code used in the optimization phases (e.g., reconcile code between two regions) is decreased due to execution time optimizations, it may also result in better compilation time. Reduction in compilation time comes about primarily due to the selection of an appropriate size of a region and the granularity of live range. Reduction in compilation time can also come about secondarily due to reduction in the number of interference edges in coloring, the number of live range splittings, the amount of reconcile code to be inserted between regions, etc. on account of execution time optimizations.

13.6.4.1.1 *Comparison of Region-Based with Function-Based Compilation.*

To understand the impact of a region-based register allocation, consider the data on compile and execution times for a “naive” compiler (i.e., no optimizations across regions as discussed later) on a selected set of benchmarks chosen from SPEC95 and common UNIX utilities (Figure 13.6) for a VLIW/EPIC machine with 4 integer units, 2 FP units, 2 memory units and 1 branch unit with 32 GPRs and 32 FPRs run on the Trimaran system. The region granularity we have used is a hyperblock, which is typically larger than a basic block.

From Figure 13.6, we see that the compilation time is mostly lower but also that it is larger for a few programs. This is to be expected because an increase in reconcile code (due to poor code generated on account of limited scope of naive region-based compilation) can also result in increased compile time. The execution time, on the other hand, is larger in every case. The code can be worse due to reasons such as follows:

- A live range has dissimilar register bindings across two adjacent regions.
- A pass-through live range has a binding that is inappropriate.
- Even with abundant registers available, only lower numbered registers are allocated to variables because each region is compiled separately without knowledge of usage of registers in other neighboring regions.
- The priority of a live range should depend on the actual cost of load and store saved. This requires knowing how a live range is allocated in other regions. Even if it is live according to information in this region, it may still be spilled in a neighboring region. Hence, the cost of allocating a register to this live range depends on information in other regions.

For all the preceding reasons, we need to introduce a controlled amount of global information (across regions) to offset the limited scope of region-based compilation.

13.6.5 Efficient Region-Based Register Allocation Framework

For efficient region-based register allocation,[23] we still have to overcome the problem of suboptimal coloring due to the limited scope in region-based register allocation, compared with a global approach

[23]This work is done jointly with Hansoo Kim (full details are available in his thesis [55]). We give only a summary.

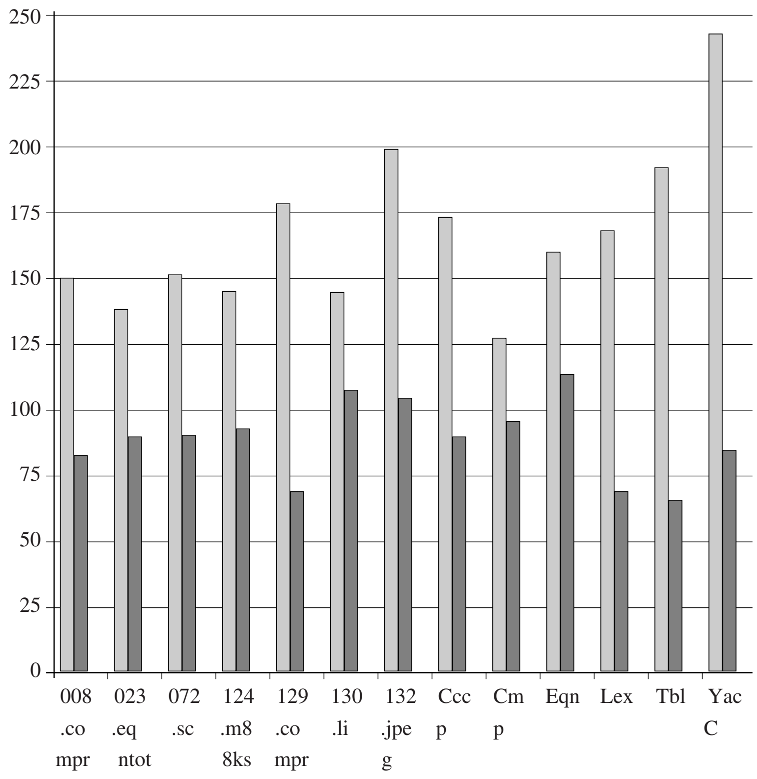


FIGURE 13.6 Execution time penalty (first bar) and compilation time advantage (second bar) with naive region-based compilation (function-based = 100).

(that looks typically function at a time). The enhanced region-based register allocation framework, shown in Figure 13.7, is based on the Chow and Hennessy framework. First, use is made of delayed binding and propagation of bindings across regions based on frequency of execution of regions. The liveness computations across regions are conservatively taken to be nonpredicated (i.e., all nonfalse predicates are promoted to true at region boundaries for liveness computations). This is for simplicity in the algorithms as well as the likely limited benefit of the predicated approach across regions. A systematic frequency-based approach to live range splitting, rematerialization and shrink wrapping is also used. The priority function is further refined for the region-based approach: for example, accurate live-in and live-out information is taken into account in the priority function by using data flow information, to avoid pessimistic assumptions about the need for storing and restoring across regions.

In addition, the concept of region restructuring based on estimated register pressure is needed to further improve the performance of region-based register allocation.

13.6.5.1 Propagation of Bindings Across Regions

With regions, one large live range can be divided into several live segments. As register allocation in each region is performed fully before the next, coloring each segment independently may be suboptimal because each live segment may be assigned a different register, resulting in patch up code at each live segment boundary. Ideally, if both adjacent live ranges are assigned the same physical register or both spilled, no compensation code is required. This requires propagation of register bindings. In principle, this problem is equivalent to the general register allocation problem (model each instruction as a region) and hence heuristics are needed.

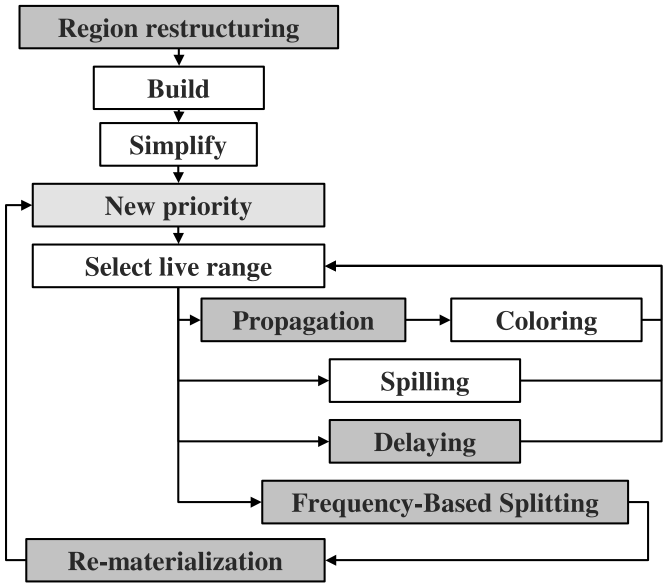


FIGURE 13.7 The framework of region-based register allocation.

To insert compensation code between two adjacent regions, another region may be required. Creating a new block can degrade the runtime performance in two ways. First, a new block needs an extra branch operation. Second, the ILP factor of the compensation code block is low because only a few operations are in it. In many cases, fortunately, these compensation codes can be moved up or down without introducing a new block.

For a good register assignment, knowing the execution frequencies of the program units such as basic blocks, superblocks, hyperblocks and others can help in propagating in the potential candidates for binding a pass-through live range by using the adjacent live range information and frequency information. If a pass-through live range does not have any adjacent live range that is already bound, register binding for that live range is delayed. If the candidates are unsuitable (forbidden, binding benefit is not beneficial when compared with spill with patch up, caller binding unsuitable due to presence of functions, etc.), the live range is further delayed. Otherwise, the preferred binding is found and reconciled by using bindings of adjacent live ranges and computing benefit of using some neighboring binding in frequency order of control flow, skipping only delayed ones. The benefit is calculated as the difference of register binding benefit and weighted compensation cost at entry and exit points. If delayed bindings also exist on either side of this pass through, compensation costs are set to zero with the expectation that the actual patch up costs can be absorbed by live ranges that get allocated later. In addition, the assignment of a binding to a delayed live range is delayed until its highest frequency control flow edge has a register binding.

One can also propagate information on unavailable registers. If a live range for a variable in the current region uses a register that is forbidden in some delayed live segment in some other region, a patch up code is unavoidable. The patch up code may be avoidable by constraining register binding for a current live segment by propagating in an exclusion list for delayed segments in neighboring regions. When a binding is delayed, the forbidden register information is propagated as unavailable registers from higher frequency region to lower ones. In addition, when the register binding for a pass-through live range is decided, the information about an unavailable register is used only when the region that passes this information has a control flow frequency larger than the frequency from a region that has a register binding.

Many programs show large performance improvement [52, 55] — this occurs especially in cases when the spill codes are placed in the outer loop instead of the inner loop. As register size increases, the performance improvement increases up to a certain point and then starts to decrease:

1. Under high register pressure (small register file size), many live segments are spilled and propagation is not critical.
2. As register size increases, some live ranges that were spilled with smaller register sets may now be allocated to registers. Propagation strategy is important to reduce patch up code. However, if enough registers exist, each live range can have its own color and even a simple propagation scheme is enough to reduce the patch up code.

13.6.5.2 Intelligent Splitting of Live Ranges

In region-based register allocation where the regions are selected before the coloring process, the regions also provide implicit live range splitting points. However, interregion live range splitting still plays an important role when the regions are constructed aggressively to obtain the higher level of ILP. These regions may need further splitting because selected regions are usually large and a high register pressure exists. Here, we explore two important strategies for improving the effectiveness of graph coloring register allocation: live range splitting and rematerialization. Splitting divides an uncolorable live range into several smaller and potentially more easily colorable live ranges, called live segments later. To preserve the semantics of the program, the shuffle code is inserted at the split points to connect the flow of values between the split live ranges. Rematerialization is another spill code saving technique. Rematerialization recomputes a value by need instead of reloading from memory, whenever this is possible and more profitable than spilling.

Two main issues in live range splitting and rematerialization are: (1) how to choose the right live ranges to split and to rematerialize, and (2) how to find the right places to split and to rematerialize. These decisions can significantly affect the quality of the generated code. The priority function can guide the register allocator to find the live range to be split. If the splitting point is located in a hot spot of the program, then the shuffle code can be executed often, which slows program execution.

13.6.5.2.1 *Frequency-Based Splitting.*

The priority-based approach of Chow and Hennessy [18], while using execution frequencies in the priority function, does not take into account execution frequencies when searching (breadth first) for a split point with highest register pressure.

Their approach uses a simple heuristic to split live ranges:

1. The first component of the original live range should be as large as possible with operations that are connected. This avoids the creation of too many small fragments.
2. The second component should be small enough to have fewer interfering live ranges than the number of available registers, so that at least one of the components is colorable.

The region-based register allocator proposed by Hank [50] does not split live ranges in a region but instead spills them if they are not colorable. The problem with these approaches is that the split point may have high execution frequency (e.g., a loop back edge); any patch up code is thus costly. Briggs [16] introduced live range splitting into the Chaitin style of register allocation. The split points are determined using an SSA representation of the program and loop boundaries. Because live range splitting is performed prior to the coloring phase, the decisions concerning split points and live range selections for splitting are made prematurely. Many heuristics have been used to eliminate this extra shuffle code, including biased coloring and conservative coalescing.

The frequency-based splitting (FBS) approach uses frequency information to guide live range splitting and attempts to split along the edges where the execution frequency is the lowest in the region.

From all the blocks considered in a region, the highest execution frequency block, called the *seed*, is selected. This first component of the live range is then expanded by adding successor or predecessor blocks in the order of the execution frequency of the control flow arc as long as the following conditions are met:

1. The expanded live range is still colorable.
2. The connecting edge to the block added has the highest execution edge frequency for that block.

The second condition is necessary to avoid cases when splitting expands too aggressively and the split points are chosen on the higher frequency edges.

13.6.5.2.2 *Rematerialization with Live Range Splitting.*

Frequency-based rematerialization (FBR) is a natural extension of FBS by incorporating rematerialization in the splitting process. The main observation is that splitting points computed by FBS are also good places to insert rematerialization code, for these reasons:

- First, FBS splitting points are, by definition, the program points of low execution frequency, and rematerialization code that is inserted at these places can also be infrequently executed.
- Second, FBS always splits a live range into two live ranges such that at least one is colorable. Rematerialized code inserted in the colorable live range is guaranteed not to be spilled.
- Finally, rematerialization can dramatically reduce the sizes of split live ranges over that of FBS, making it more likely that these live ranges can be colored as well as other live ranges.

The basic framework of FBR is very similar to FBS. After splitting the live range `(LR)` into two live ranges `(LR1` and `LR2)`, a check is made of all the cross edges `(E)` between `LR1` and `LR2` whether the value crossing `E` is rematerializable. If it is, the rematerialized code is inserted and the new live range information updated. Unlike splitting, which introduces a new definition and a new use (in the shuffle code), rematerialization introduces a definition but does not introduce a new use.

Experiments show that FBS and FBR achieve better quality of code and reduce the execution time by 10 and 7%, respectively [52, 55]. Furthermore, a combination of these two techniques improves performance by as much as 12%. Most of the performance improvement of FBR can be attributed to rematerialization of function call addresses, branch target addresses and address offsets from the stack pointer.

13.6.5.3 Enhanced Priority Functions

The priority order for coloring is an important factor in register allocation because it determines the variables to be spilled, thereby affecting the amount of spill code. Region-based approaches need to use more accurate priority functions as explained later.

13.6.5.3.1 *Accurate Live-In and Live-Out Information.*

For a given live range lr in blocks B_1 to B_m, the priority function can be defined as follows:

$$PR(lr) = \sum_{i\ \in 1,\ldots,m} ((D_i - C_i) \cdot ST\_COST + \quad (U_i - C_i) \cdot LD\_COST) \cdot w_i$$

where D_i, U_i, C_i is the number of definitions, uses and function calls, respectively, in B_i whereas w_i is its weight (assuming only caller registers for now).

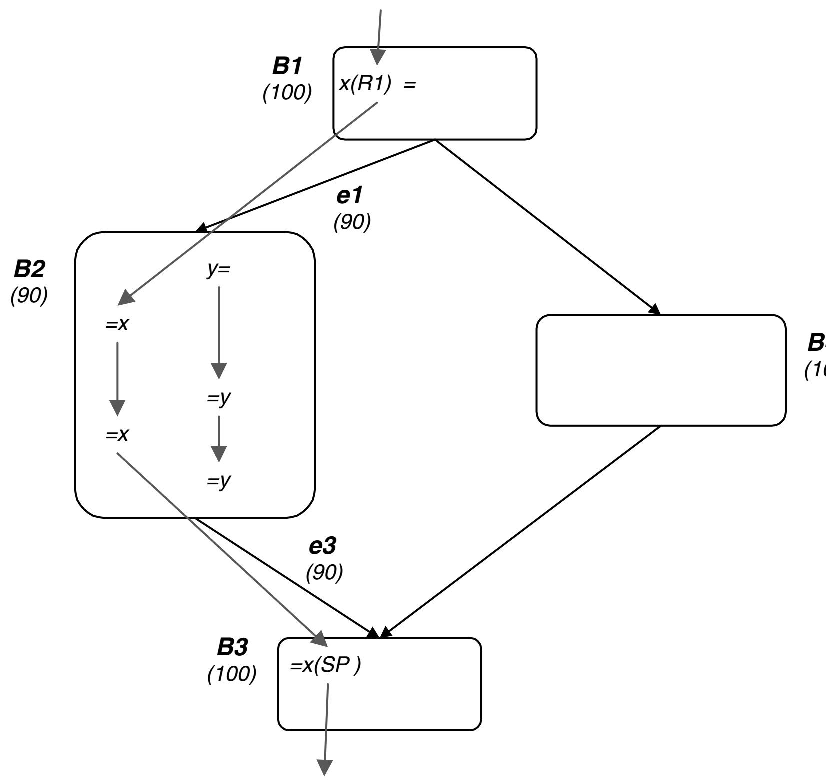


FIGURE 13.8 Priority measure dependence on region size on a four-block CFG.

In Figure 13.8, global priority $PR(x)$ and $PR(y)$ for live ranges x and y by Chow's approach [18] is:

$$PR(x) = ST\_COST \cdot 100 + LD\_COST \cdot 2 \cdot 90 + LD\_COST \cdot 100$$

$$PR(y) = ST\_COST \cdot 90 + LD\_COST \cdot 2 \cdot 90$$

If our compiler is region based and if region $B2$ is the current region for register allocation, the priority of x and y in region $B2$ is:

$$PR(x_{B_2}) = LD\_COST \cdot 2 \cdot 90$$

$$PR(y_{B_2}) = ST\_COST \cdot 90 + LD\_COST \cdot 90 \cdot 2$$

Range y has higher priority than x, and x can be spilled in region $B2$ if only one register is available. If x is bound to a register in $B1$ and $B3$, we need a STORE for x on entry and a LOAD on exit of region $B2$. Hence, the saving depends on the register bindings in neighboring regions.

To handle the preceding, Chow's priority function has to be modified to include register bindings in neighboring regions and edge frequency of live-in and live-out value of a live range. Let $B_i(x)$ be the live range of x in region B_i, $freq(B_i, B_k)$ be the edge frequency between region B_i and B_k and $freq(B_i)$ be the region weight of B_i. Let $N_{in}(B_i(x))$ be the set of live segments that precede $B_i(x)$ and bound to a register and $N_{out}(B_i(x))$ be the set of live segments that succeed $B_i(x)$ and also bound to a register. Define priority $PR(B_i(x))$, the number of STORE/LOAD operations that can be saved by binding x to a register instead of spilling (including compensation code at region entry or exit), as follows:

$$N_{in}(B_i(x)) = \{B_j(x) \mid (freq(B_j) > freq(B_i)) \wedge (B_j(x) \text{ is a preceding live segment of } B_i(x)) \\ \wedge\ (B_j(x) \text{ is bound to register})\}$$

$$N_{out}(B_i(x)) = \{B_j(x) \mid (freq(B_j) > freq(B_i)) \wedge (B_i(x) \text{ is a preceding live segment of } B_j(x)) \\ \wedge\ (B_j(x) \text{ is bound to register})\}$$

$$\begin{aligned} PR(B_i(x)) = {} & ST\_COST \cdot D_{B_i} + LD\_COST \cdot U_{B_i} \\ & + \sum_{B_j(x) \in N_{in}(B_i(x))} ST\_COST \cdot freq(B_j, B_i) \\ & + \sum_{B_k(x) \in N_{out}(B_i(x))} LD\_COST \cdot freq(B_i, B_k) \end{aligned}$$

13.6.5.3.2 ***Dynamic Priority.***
Chow's priority function is normalized by the size of the live range, approximated as the number of live units (basic blocks) in it. However, normalization by the degree of the node in interference graph (as in Chaitin) may be better for region-based compilation because some structural information is already available through the formation of regions. The problem with the standard priority-based approach is that the priority of a live range never changes throughout the coloring process unless it is split. Thus, it may fail to capture the dynamic benefit of a register binding.

To reflect this aspect, one can update the priority function dynamically by using the varying interference degree of uncolored live ranges as normalization factor while register allocation proceeds. Many programs show significant improvement with this optimization [52, 55].

13.6.5.3.3 ***Predicated Priority.***
Predication can impact coloring in the following ways: the priority measure may need to take into account predicates or the live range analysis has to be predicate aware.

Consider Figure 13.9 where the weights of $B2$ and $B3$ are 90 and 10, respectively. Hence $LiveRange(x)$ has a higher priority than $LiveRange(y)$ in $B2$. In the corresponding hyperblock, as in Figure 13.9(b), the priority of $LiveRange(x)$ and $LiveRange(y)$ is the same, even if they are executed a different number of times dynamically. To correct this problem, the priority measure has to be further modified to reflect frequency information through the use of predicate expressions. Once each of D_i, U_i and C_i (call it X_i) is redefined to be a summation over each of n occurrences of any variable x multiplied with its predicated probability $p_j(x)$: $X_i = \sum_{j \in 1,\ldots,n} p_j(x)$, we can still use the previous definition of priority (see Section 13.6.5.5 for details).

The performance improvement with the three changes to the priority function (accurate live-in and live-out information, dynamic priority and predicate-aware priority function) is at least 5% improvement in many programs [52, 55]. With increased complexity of priority function, it is important that the compile time does not increase substantially. The data [52, 55] show that compilation time overhead is marginal. In some cases, the time decreases: the extra time in priority computation has been compensated by fewer spill code insertions.

13.6.5.4 Partitioning of Registers Across Regions

In region-based register allocation with large number of registers (the likely case with EPIC architectures), it is important to do a high-level (global) partitioning of available registers into sets of registers for coloring in each region. This prevents reuse of registers across regions that may result in excessive patch up code because each region may do coloring without information about coloring in adjacent regions.

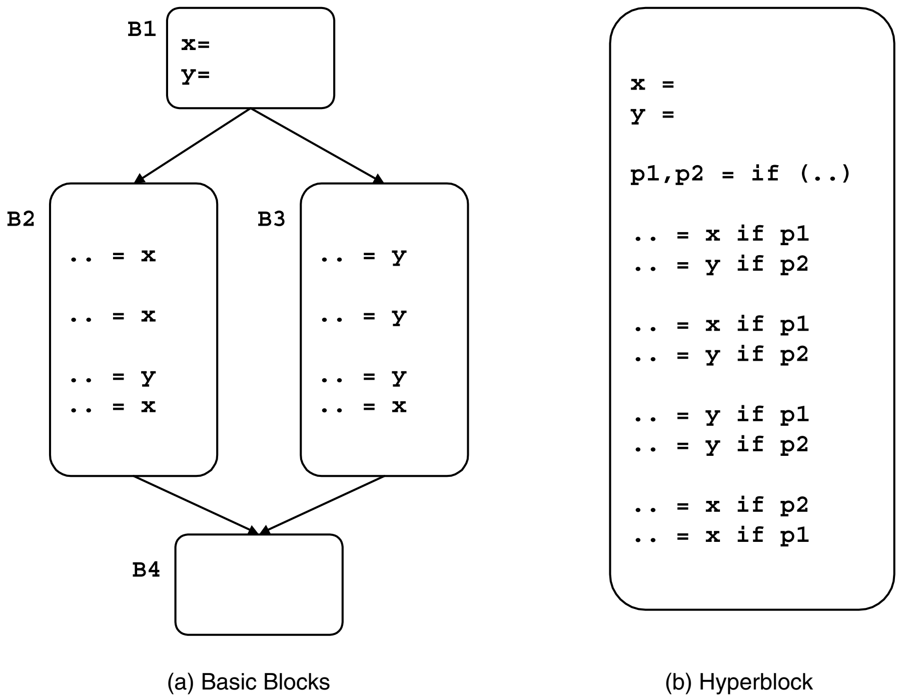


FIGURE 13.9 Priority measures with predication.

Note that such "gratuitous" patch up code may be necessary not only with one region in between just considered but also with two, three or more regions in between. Hence, there is a need for intelligent precoloring so that gratuitous patch up code does not result even when plentiful registers are present. The basic problem is that each region may independently allocate registers from the same subset of registers even when other subsets are available.

Some graph-theoretical ideas that have been used in the past to decompose a graph into subgraphs for register allocation are clique separators [94] and approximation through multicommodity flows [65]. In the first approach, the maximum number of cliques in which a live range can occur, and chosen as separators, decides the regions. In the latter approach, one finds a minimum balanced directed cut of DAG to split DAG into roughly equal pieces G1 and G2 such that minimum number of live ranges cross the cut and so that edges exist from G1 to G2 but not from G2 to G1. We can now process (register allocation or scheduling) G1 before G2. However, in region-based approaches, the partitions into regions is based on frequency information that is unlikely to be the same as the partitions obtaining in the graph-theoretical approaches. In addition, such approaches can add to compile time.

Due to the preceding difficulties, one simple approach can be the use of a hash function based on variable name and register size to locate a free register. Another approach is the "clockhand" algorithm.[24] Maintain a "freelist" of registers with a clockhand. For each region, first compute its clockhand (i.e., the starting place in the freelist from where registers get allocated) by choosing the best noninterfering clockhand of all neighboring already allocated regions. At the end of register

[24]The term is borrowed from OS scheduling where the processing of certain lists is started from where last terminated.

allocation for this region, store the first clockhand used and the last register allocation to enable clockhand computation by other later nearby regions. As register file size increases, the clockhand algorithm improves performance in many cases [52, 55].

13.6.5.5 Optimizations Across Function Boundaries with Regions

13.6.5.5.1 *Caller and Callee Models.*

Allocating different type of registers to live ranges requires different store and reload costs, and this needs to be reflected in the formula for priority function. For each live range, Chow and Hennessy use the caller-saved priority function $P_r(lr)$ and the callee-saved priority function $P_e(lr)$. The basic priority function given in Equation (13.8) can be redefined, therefore, as:

$$Priority(lr) = \frac{max\ (P_r(lr),\ P_e(lr))}{size(lr)} \tag{13.8}$$

$$P_r(lr) = \sum_{i\ in\ B_1,\ldots,B_m} (D_i \cdot ST\_COST + U_i \cdot LD\_COST$$

$$- C_i \cdot (ST\_COST + LD\_COST)) \cdot W_i \tag{13.9}$$

$$P_e(lr)d = \sum_{i\ in\ B_1,\ldots,B_m} (D_i \cdot ST\_COST + U_i \cdot LD\_COST$$

$$- (ST\_COST + LD\_COST)) \cdot W_i \tag{13.10}$$

where D_i is the number of definitions, U_i is the number of uses, C_i is the number of procedure calls in block B_i and W_i is the weight of B_i with m live units in the live range.

In the case of caller-saved priority function, the cost of store and reload of the registers around the function calls weighted by frequency is considered in the priority function. For the callee-saved register, the extra cost occurs only once for each callee-saved register at each procedure entry point. Thus, only the first live range that uses a given callee-saved register needs to account for these savings and restoring costs. Once these costs have been considered, the same callee-saved register can be used to contain other live ranges for free. Hence, $P_e(lr)$ for a given live range can assume two values. By using predication as discussed in Section 13.6.5.3, the priority function for a region-based approach can be given as:

$$Priority(lr) = max\ (P_r(lr),\ P_e(lr)) \tag{13.11}$$

$$P_r(lr) = \sum_{i\ in\ B_1,\ldots,B_m} (D_i \cdot ST\_COST \cdot PR(D_i) + U_i \cdot LD\_COST \cdot PR(U_i)$$

$$- B_i \cdot (ST\_COST + LD\_COST) \cdot PR(B_i)) \cdot w_i \tag{13.12}$$

$$P_e(lr) = \sum_{i\ in\ B_1,\ldots,B_m} (D_i \cdot ST\_COST \cdot PR(D_i) + U_i \cdot LD\_COST \cdot PR(U_i)$$

$$- (ST\_COST + LD\_COST)) \cdot w_i \tag{13.13}$$

We do not need to normalize by size of live range (as in the Chow and Hennessy approach) because region formation already has split live ranges.

13.6.5.5.2 *Shrink Wrapping.*

This optimization places the store and load for callee-saved registers so that they occur only over regions where the registers are used. This ensures that the live range is split with the goal of the least number of dynamically executed store and load instructions for the callee register. Due to lack of frequency information, the placement can be suboptimal and Chow [28] reports many negative results. If the frequency of execution of a live range is less than that of the procedure, shrink wrapping can be beneficial. By constructing dominator and postdominator trees, we can find the best place to insert the ld/st code for a callee-saved register. The algorithm is as follows:

1. For each callee-saved register, identify all the blocks B_i using the register.
2. Create the dominator or postdominator tree (for finding the st/ld point, respectively), compute the number of executions of each block in the tree and store it as the weight of the block.
3. For each B_i in the dominator and postdominator tree:

   a. Select the minimum weight node n on the path from root s to B_i as the st/ld point.
   b. Delete each child node of n from the tree and reduce the weight of n from all nodes on the path from s to n.

An additional optimization is the insertion of the ld/st code on edges instead of in code blocks because the edges may be less frequently executed than the blocks. The problem of finding optimal edges for insertion of ld/st code can be reduced to the *max-flow min-cut* problem [90]. In the called procedure, consider all the blocks with the first (last) use of a variable allocated to the callee register. Connect all these blocks to a pseudo destination (source) with infinite capacity edges. Find the min-cut.[25] This gives the store (load) locations.

CUT performs better than DOM in some cases, but worse than DOM or even the base case sometimes [52, 55]. The latter is due to the overhead of creating a new block when we need to insert code on control flow edges (for the same reasons when creating new blocks for inserting patch up code between regions: see Section 13.6.5.1).

13.6.6 How to Further Reduce Execution Time

One can modify region formation also to take into account aspects important to register allocation (in addition to current criteria such as execution frequency of operations; memory accesses; structure of regions such as basic block, superblock, hyperblock or loop; and size of the region in operations). Alternatively, region restructuring can use information such as the maximum bandwidth of live variables at each program point in each region of basic block, superblack and hyperblock to form new regions from old. This may help register allocation. Starting from a seed block (the most frequently executed block not yet in a region), we add successor or predecessor (block n) of the current region based on execution frequency. The block n is included as long as the register bandwidth of n is less than the number of available registers. The expansion stops if one of the following condition holds:

- No blocks are left that do not belong to the region.
- The selected block already belongs to the other regions.
- The selected block has higher register bandwidth than the number of available registers.

By using all the preceding optimizations, in most cases, region (block)-based register allocation has an execution performance comparable with, or better than, function-based register allocation [52, 55]. In addition, it has a faster compilation time, because the interference graph construction requires $O(N^2)$ time.

[25]If irreducible loops are among the blocks, the preceding formulation is not efficient because it considers some extra edges as part of the min-cut problem. A more complete solution would eliminate irreducible loops by removing the edges corresponding to the loop in the canonical irreducible loop with three nodes.

13.7 Region-Based Framework for Combining Register Allocation and Instruction Scheduling

Because regions are typically subprocedure level, it is important to decide on the phase-ordering issues, especially with respect to register allocation and instruction scheduling. We next discuss at a high level the framework present in Trimaran.[26]

13.7.1 Trimaran Framework

The following is the outline of the current implementation of instruction scheduling and register allocation in Trimaran:

```
horizontal_sched_RA(Procedure P) {
    pre_pass_schedule_all_regions(P)

    register_allocate_all_regions(P)
    generate_reconcile_code(P)

    post_pass_schedule_all_regions(P)
}

pre_pass_schedule_all_regions(Procedure P) {
    for each region R in P
        pre_pass_schedule(R)
}

register_allocate_all_regions(Procedure P) {
    for each region R in P
        register_allocate(R)
}

post_pass_schedule_all_regions(Procedure P) {
    for each region R in P
        post_pass_schedule(R)
}
```

Figure 13.10 shows the state of the graph at various stages of the compilation. The rectangular blocks represent the original regions of the procedure. The small squares represent the additional basic blocks added to accommodate the patch up code. Black regions indicate the unscheduled portion. Figure 13.10(B) shows the code added due to intraregion register reconcile. Figure 13.10(C) shows the patch up blocks added due to global register reconciliation.

The prepass scheduling phase is function based and is performed on all the regions of the function at a time. This phase considers the operands to be assigned to virtual registers and not physical registers. Register allocation follows after the prepass scheduling phase. The register allocator itself is region based, but it cannot be mixed with the scheduling phase because of the function-based nature of scheduling. Spill code is generated within the regions during this phase.

[26]This section is based on work with my former students, Pradeep Jain and Yogesh Jain [43].

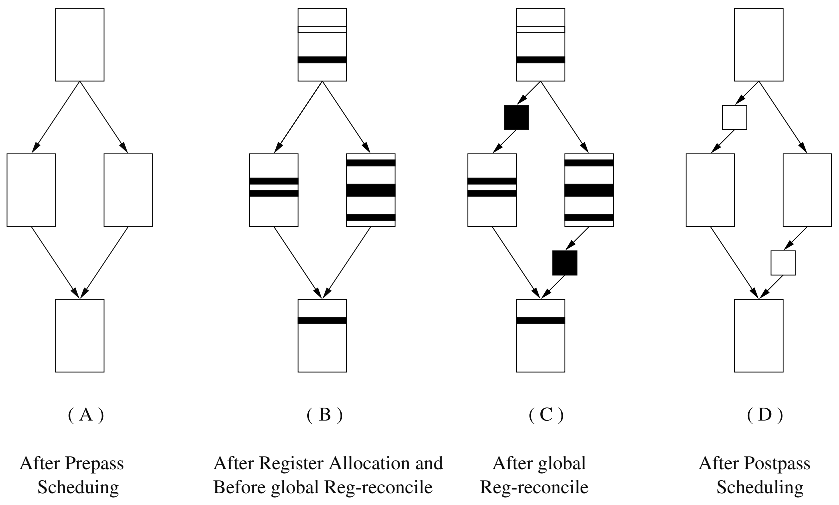

(A)
After Prepass Scheduling
(B)
After Register Allocation and Before global Reg-reconcile
(C)
After global Reg-reconcile
(D)
After Postpass Scheduling

FIGURE 13.10 Horizontal compilation: IR graph at various stages.

Also, new basic blocks are added to the graph to accommodate patch up code for interregion live operands.

The graph seen by the postpass scheduling phase is different from the one seen by the prepass scheduling phase because it contains additional spill code operations within the regions and it contains extra basic blocks that contain patch up operations for interregion live operands. Postpass scheduling is similar to prepass scheduling in all other respects. We briefly discuss meld scheduling next to help in understanding the region-based framework for combined instruction scheduling and register allocation.

13.7.1.1 Meld Scheduling

Scheduling algorithms are usually oblivious of constraints coming in from blocks or regions that are already scheduled and hence may make scheduling decisions that lead to poor schedules. It is, however, possible to improve the schedules if the constraints from already scheduled regions are propagated to the regions yet to be scheduled. Meld scheduling is an example of this approach [4].

In a noninterlocked machine, the processor does not interlock to ensure that the inputs are available before issuing an operation. For such machines, the compiler schedules code to guarantee that an operation completes before a dependent operation issues. Within a scheduling region, the scheduler delays the issue of a dependent operation to ensure that its inputs are available. Across scheduling regions, a scheduler must ensure that certain constraints are satisfied on entry or exit of a region. For instance, one convention is to assume that on entry all operations have their inputs available. In this case, the scheduler must guarantee that all operations complete before control is transferred to another region. In contrast, a meld scheduler generates latency constraints imposed by scheduled regions, propagates constraints to the boundaries of a region to be scheduled and translates these constraints to edge constraints recognized by the local region scheduler. We omit further discussion because additional details are available in [4].

13.7.2 Stepwise Development of a Combined Region-Based Framework

We next present a region-based framework for combining instruction scheduling and register allocation. Design decisions are given in the following subsection by using a stepwise refinement approach.

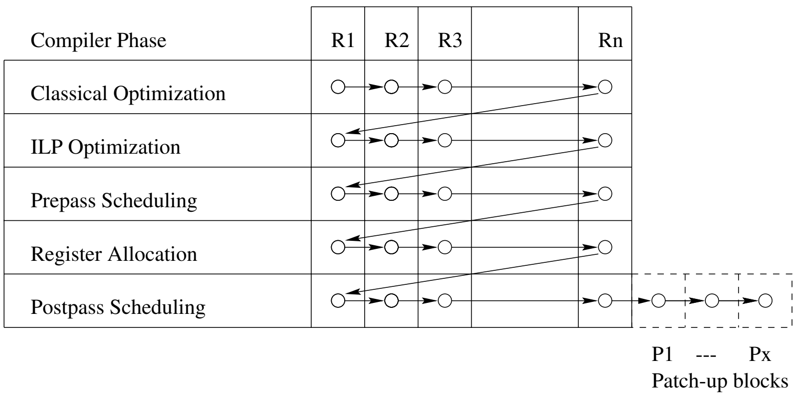


FIGURE 13.11 Block diagram for horizontal compilation model.

13.7.2.1 Horizontal Compilation Model

The current implementation of both register allocation and scheduling in Trimaran is based on the horizontal compilation model as discussed in Section 13.7 (see also Figure 13.11). In this model, the compilation phases are completely decoupled. This has the following drawbacks:

- If the register allocator wants to use any information from the prepass scheduler, such information should be stored for all the regions. This increases memory requirements. The same occurs if the postpass scheduler wants to use any information from the register allocator.
- Most of the flow analysis and liveness analysis is common for prepass and postpass scheduling phase. To reuse this information, it should be stored for all the regions. This again increases memory requirements. Currently, this analysis is undergoing recalculation for the postpass scheduler phase. This increases the compilation time. The difference is considerable because most of the analysis is complex, especially in the context of predicated code.

13.7.2.2 Vertical Compilation Model I

To avoid storing the information for all the regions between the successive use of information, we run all three phases on a region in sequence. This reduces memory requirements. The Figure 13.12 shows the order of execution of compilation phases in this model. The following is the outline of this model:

```
vertical_sched_RA_model_1(Procedure P) {
    for each region R in Procedure P

        pre_pass_schedule(R)
        register_allocate(R)
        post_pass_schedule(R)

    generate_reconcile_code(P)
    schedule_patch_up_blocks(P)
}
```

During the register allocation phase, the actual reconcile code is not generated; only the information for reconcile code is stored on the corresponding edges. After all the original regions are processed, the reconcile code for interregion live ranges is generated. The new patch up basic blocks generated are then scheduled separately. In this model, the original regions are not scheduled after adding the patch up blocks. This could result in an inferior schedule if meld scheduling is enabled.

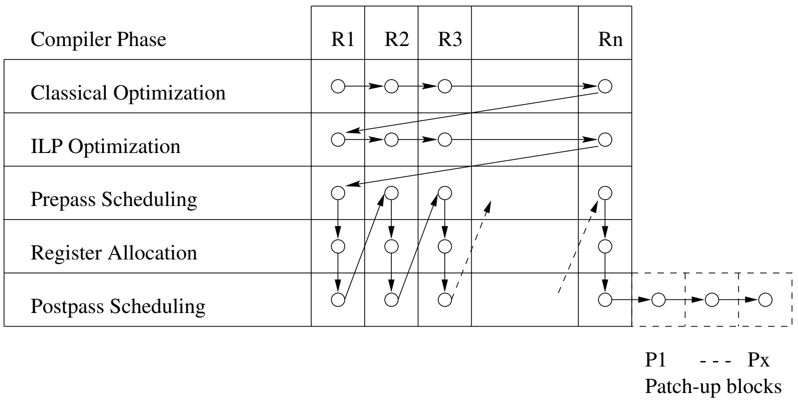


FIGURE 13.12 Block diagram for vertical compilation model I.

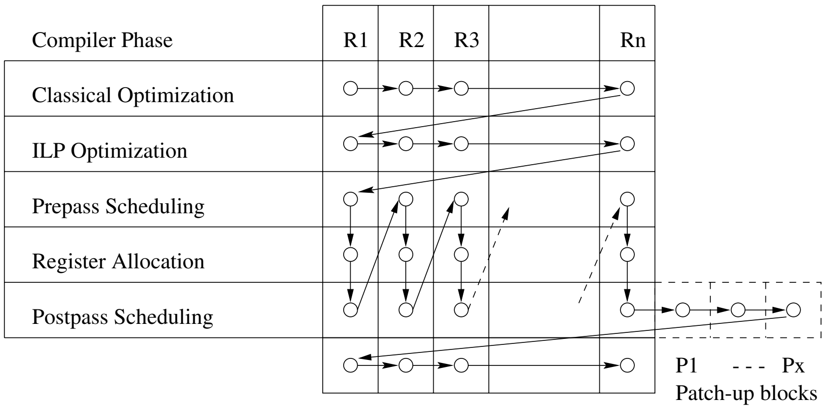


FIGURE 13.13 Block diagram for vertical compilation model II.

13.7.2.3 Vertical Compilation Model II

To ensure that the presence of patch up blocks is reflected in the scheduling of the original regions, an additional postpass scheduling phase is introduced. The outline of the model is given as follows (Figure 13.13):

```
vertical_sched_RA_model_2(Procedure P) {
    for each region R in Procedure P
        pre_pass_schedule(R)
        register_allocate(R)
        post_pass_schedule(R)

    generate_reconcile_code(P)

    for each region R in Procedure P
        post_pass_schedule(R)
}
```

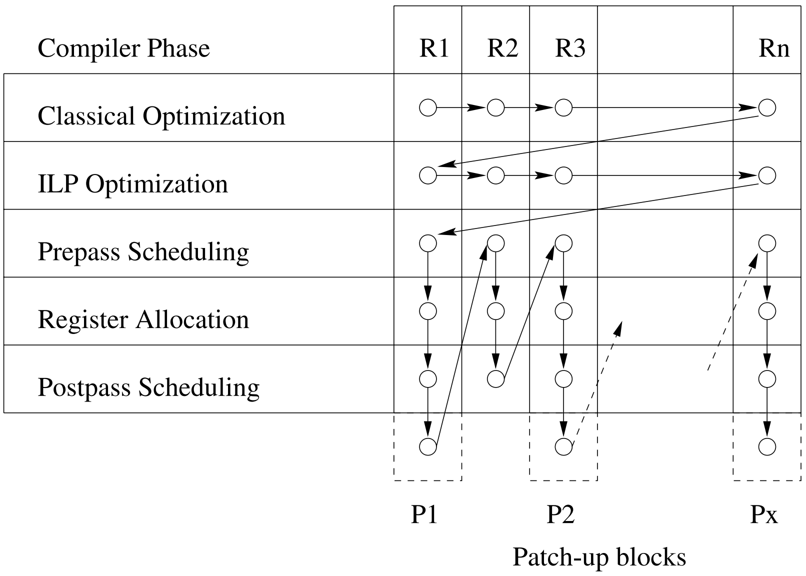


FIGURE 13.14 Block diagram for vertical compilation model III.

The final schedule of this model is as good as the horizontal model. However, the scheduling of each region is done three times in this model in contrast to two times in the previous model. This is obviously costly in terms of compile time.

13.7.2.4 Vertical Compilation Model III

In the final model (Figure 13.14), a reconcile code corresponding to a region is run immediately after its register allocation. The outline of this model is given next:

```
vertical_sched_RA_model_3(Procedure P) {
    for each region R in Procedure P
        pre_pass_schedule(R)

        register_allocate(R)
        generate_inter_region_reconcile_code(R)

        post_pass_schedule(R)
}
```

13.7.3 Problems in Vertical Compilation Model III and Solutions

13.7.3.1 Eager Propagation of Interregion Reconcile Information

The register reconcile information is generated immediately after the register allocation of the region. It is desired that this information be propagated completely so that further processing can make use of it.

The postpass scheduling of the region takes care of the intraregion patch up code added during the register allocation of the region. Under certain conditions, the interregion reconciliation code may be added to either of the two regions. This is possible when the exit-op of the region is a dummy-branch or when only one edge is incoming to the region. Postpass scheduling takes care of such reconciliation code also.

In all other cases, extra patch up blocks are added to accommodate the reconciliation code. These patch up blocks are made similar to the original blocks in all respects, so that the meld scheduling of the remaining regions takes into account the latency produced by these regions. Thus, the meld scheduling works with the most accurate latency information.

13.7.3.2 Handling of Patch Up Basic Blocks

When should we schedule the patch up basic blocks? For calculating the meld constraints, the paths consisting of only scheduled blocks are considered. As shown in Figure 13.15, the meld constraints from region *A* are propagated to region *B* if patch up block *P* is scheduled. Otherwise, it is assumed that the scheduling of *P* can absorb the latency constraints of both regions *A* and *B*.

In the absence of *P*, the meld constraints of region *A* are propagated to region *B*. This is not possible if *P* is present and is unscheduled. In short, the meld constraints become more restricted if the patch up blocks are left unscheduled. Thus, the patch up basic blocks should be scheduled as soon as they are created.

13.7.3.3 Handling the Pass-Through Live Ranges

Consider Figure 13.16. The dark line represents a global live range that consists of local live ranges $L1$, $L2$, $L3$ and $L4$. Assume that $L2$ and $L3$ are pass-through live ranges and the regions are register allocated in the order B, C, A and D.

In the original horizontal model of register allocation in Trimaran, the pass-through live range is left unresolved until at least one of its adjacent global live ranges is resolved. Thus, when *B* is register

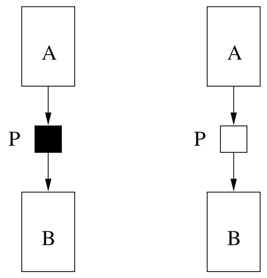


FIGURE 13.15 Meld constraint propagation through patch up basic block.

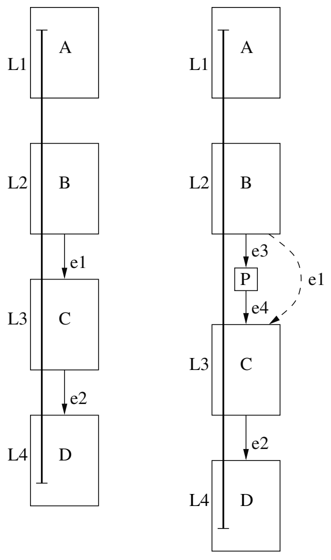


FIGURE 13.16 Reconciliation of pass-through live ranges.

allocated, $L2$ is left unresolved. Similarly, when C is register allocated, $L3$ is left unresolved. When A is register allocated, this binding is propagated to $L2$ and then to $L3$. This propagation only adds the corresponding patch up code on the edge and does not generate any patch up block in the graph until the end of register allocation of all the regions.

In a vertical model, the new patch up block P is added corresponding to edge $e1$ when C is processed. When A is register allocated, the propagation of binding of $L1$ might result in additional patch up code in P. This would require rescheduling of P. This cannot be allowed because earlier regions have used the meld constraints based on the old schedule of P.

One can avoid the preceding problem by not allowing the binding of the pass-through live ranges to be delayed. For a pass-through live range, if an already resolved global adjacent live range is not found, we can spill it. This could result in spilling of some pass-through live ranges that are successfully bound in the horizontal model in Trimaran. However, the performance is not affected much, as shown by experimental results.

13.7.3.4 Hiding Patch Up Basic Blocks from Register Allocator with Delayed Pass Throughs

As pointed out earlier, the meld scheduling considers the patch up basic blocks to be similar to the original blocks. However, how should the register allocator view the patch up basic blocks? When the resolution of pass-through live ranges is delayed, as in the horizontal model in Trimaran as pointed out in the previous section, it is possible for a patch up code to be added to an edge in more than one visit.

Again consider Figure 13.16. If P is considered to be similar to the original blocks by the register allocator, then the patch up block for the pass-through live range is required on the edge $e3$ and $e4$. Because P is already scheduled, new basic blocks would be required corresponding to $e3$ and $e4$. This is obviously not a practical solution.

Thus, for the purpose of register allocator, the patch up blocks should be dealt differently. One simple solution is to assume that the register allocator views only the original edges of the graph and the patch up blocks are added only corresponding to the original edges. However, the original edges must be deleted; otherwise the meld scheduling bypasses the patch up block. Thus, the original edges of the graph are stored in a map and hash structure before beginning to process the regions. The register allocator reads the edges for the operations from the preceding data structures instead of the actual edges of the operations.

The preceding approach is not required if we use the spilling technique as described in Section 13.7.3.3. If that technique is not used, this design allows the handling of pass-through live ranges properly.

13.7.3.5 Prologue and Epilogue Regions

If the callee-saved registers are used by the register allocator, a reconcile code must be generated for them. It consists of adding the load and store operation in the prologue region and epilogue region of the procedure, respectively. Because the callee-saved registers used are known only after the register allocation of all the regions, the reconcile code is added to the epilogue and prologue region after processing all the region. As a result, these prologue and epilogue regions should be scheduled at the end. The same care is not required for these regions in the horizontal model because the reconcile code for the callee-saved registers is added in the global–register allocation phase and the postpass scheduling of the prologue and the epilogue takes place later.

13.7.4 Final Design

All the preceding issues can be incorporated in a framework for combined region-based instruction scheduling and register allocation. The outline of this model is given as follows:

```
vertical_sched_RA_model_final(Procedure P) {
    form a sorted list of original regions of the procedure P,
    called initial_regions

    for each region R in initial_regions
        for each entry_op OP of R
            bind < OP, list of in_edges of OP > to
              initial_inedges

        for each exit_op OP of R
            bind < OP, list of out_edges of OP > to
              initial_outedges

    for each region R in initial_regions
        pre_pass_schedule(R)

        register_allocate(R)
        generate_inter_region_reconcile_code(R)

        post_pass_schedule(R)

        generate_spill_code_for_used_callee_registers()
        schedule_prolog_and_epilog(R)
}
```

13.8 Toward Integrating Instruction Scheduling and Register Allocation

As mentioned earlier (Section 13.1.4), a full-blown combined algorithm requires the region to be as small as an instruction itself but that requires a very tight coupling between the two optimizations as in the Multiflow compiler [36] or in mutation scheduling [86].[27] Some not very tightly coupled approaches are as follows. Goodman and Hsu [40] use an on-the-fly approach that detects excessive register pressure during instruction scheduling and spills live ranges to reduce it. When register pressure is not high, instruction scheduling exploits ILP. Once register pressure builds up, it switches to a new state that gives preference to instructions that reduce register pressure and possibly results in lower ILP. Spilling is also used by Norris and Pollock [85]; they use a parallel interference graph, which represents all interferences that can occur in legal schedules, to detect excessive register pressure. Live range splitting is used by Berson, Gupta and Soffa to reduce register pressure: when the register pressure is high and no ready instructions that reduce register pressure exist, a live range is split by putting a store into the ready list, putting the corresponding load into the not ready list and moving all dependencies of all unscheduled uses of the value from the original definition to the load. They use *reuse dags* that help identify excessive sets: these

[27]Mutation scheduling attempts to unify code selection, register allocation and instruction scheduling, into a single framework in which trade-offs between functional, register and memory bandwidth resources of the target architecture are on the fly in response to changing resource constraints and availability.

represent groups of instructions with parallel scheduling that requires more resources than are available.

Here we discuss a register constraint aware or register pressure aware scheduling as opposed to performing scheduling and register allocation in one pass.[28] Using the framework in the previous section, we attempt to integrate instruction scheduling and register allocation by first using a profile-insensitive list scheduler, then incorporating register pressure to make it sensitive to register allocation costs and finally making it profile sensitive.

The outline of the approach is as follows:

- Use a profile-insensitive list scheduler, such as a critical path scheduler. This is equivalent to the list scheduling with the distance from the last exit as the list priority. This gives us a scheduling rank.
- Compute a register rank using, for example, the CRISP model from [81].
- Combine the two ranks to get a schedule sensitive to register pressure.
- Convert this profile-insensitive schedule to a profile-sensitive schedule that is also sensitive to register pressure, for example, using Chekuri's heuristic [27].

13.8.1 Combined Register Allocation and Instruction Scheduling Problem Priority Model

The CRISP priority model is based on the following two rank functions [81]:[29]

Register rank γ_R — For each instruction v, define:

$$\begin{aligned}
Thick_2(v) &= |\{(i, j, V_r) | (i, v) \in DG^* \wedge (v, j) \notin DG^* \wedge (j, v) \notin DG^* \wedge i \text{ and} \\
&\qquad j \text{ access virtual register } V_r\}| \\
&= \sharp \text{ value ranges from an unrelated instruction to a descendant of } v \\
Thick_1(v) &= |\{(i, j, V_r) | (v, j) \in DG^* \wedge (i, v) \notin DG^* \wedge (v, i) \notin DG^* \wedge i \text{ and} \\
&\qquad j \text{ access virtual register } V_r\}| \\
&= \sharp \text{ value ranges from an ancestor of } v \text{ to an unrelated instruction}
\end{aligned}$$

The instructions are then sorted in increasing order of $\Delta_v = (Thick_1(v) - Thick_2(v))$. The register rank $\gamma_R(v)$ is the rank of instruction v in this sorted order. Instructions with the same Δ value are assigned the same rank. Δ_v is helpful because it can prioritize a node v relative to nodes that are incomparable to v (i.e., those that are neither an ancestor nor a descendant of v).

When $Thick_1(v)$ is large, a large number of nodes are independent of node v and it would be beneficial from register considerations to schedule node v later so as to shorten the live ranges of these nodes. Thus, we make a larger $Thick_1(v)$ value lead to a larger Δ_v rank. Conversely, if $Thick_2(v)$ is large, it would be beneficial to schedule v earlier. Thus, we make a larger $Thick_2(v)$ value lead to a smaller Δ_v rank.

Scheduling rank γ_S — We can use the rank function from any suitable instruction scheduling algorithm as the scheduling rank. We assume the critical path rank function as the scheduling rank, γ_S. For any choice of parameters $\alpha, \beta \in [0, 1]$ such that $\alpha + \beta = 1$, we now have the following heuristic.

[28]This section is based on work with my former students, Pradeep Jain and Yogesh Jain [43].

[29]There is also one more model for the rank functions that takes into account the number of functional units; we use the one given as more intuitive.

(α, β)-**Combined algorithm** — This heuristic creates a combined rank function $\gamma = \alpha\gamma_S + \beta\gamma_R$, and orders the instructions into a list in increasing order of rank. It then runs the greedy list scheduling algorithm using this list to obtain a schedule.

The heuristic described earlier is for a fixed α or β value. The parameter α (or β) can be viewed as a tuning parameter, and can be set differently for different processor architectures, size of basic blocks, superblocks and hyperblocks.

By using the preceding two ranks, we can schedule and allocate registers in each region, using both a prepass and an optimized postpass as described in the previous section. However, this schedule is not profile sensitive. We next discuss Chekuri's heuristic [27] for generating a schedule that is profile sensitive.

13.8.2 Chekuri's Heuristic

Let $G = (V, E)$[30] denote the precedence graph derived from the source program. Each vertex in the graph represents an operation τ_i with specified execution time t_i, which is the time required to execute τ_i. Each vertex also carries a weight w_i. The weight w_i is the probability that the program exits at vertex i. We are to schedule the vertices of the graph on the functional units to minimize the weighted execution time, $F = \Sigma_i w_i f_i$, where $f_i = s_i + t_i$ is the finish time of operation τ_i and s_i is its start time.

A subgraph is said to be closed under precedence if for every vertex u of the subgraph, all vertices preceding u are also in the subgraph. We define the rank of a vertex i to be the ratio $r_i = t_i/w_i$. For any set of vertices $A \subseteq V$, we define its weight as $w(A) = \Sigma_{v_i \in A} w_i$, and its execution time as $t(A) = \Sigma_{v_i \in A} t_i$; based on this the rank is $r(A) = t(A)/w(A)$. The notion of the rank of a set of vertices is meant to capture their relative importance comparing the sum of their weights to the cost of executing them. Intuitively, the sum of the weights is the contribution made by the set of vertices to the weighted finish time, whereas the sum of their execution times is the delay suffered by the rest of the graph as a result of scheduling the set of vertices first. A schedule for the case where the number of functional units is one is termed a sequential schedule.

Given a weighted precedence graph G, define G^* to be the smallest precedence-closed proper subgraph of G of minimum rank. Chekuri et al. [27] prove that, for any graph G, there exists an optimal sequential schedule where the optimal schedule for G^* occurs as a segment that starts at time zero. Given G^*, we can recursively schedule G^* and the graph formed by removing G^* and putting their schedules together to obtain an optimal schedule for the entire graph. Though the problem of finding G^* for an arbitrary precedence graph is NP-hard, for the case where the precedence graph is an superblock or hyperblock, G^* may be easily computed by a greedy method for superblocks or exhaustive enumeration (when there are a small number of exits) for hyperblocks [27].

Thus, we may obtain an optimal sequential schedule for a given precedence graph G, which in turn may be used to obtain a schedule for the multiple functional unit case. Chekuri et al. [27] also show that list scheduling, using the optimal sequential schedule as the list, gives a good approximate solution for the hyperblock case. They show that, if all operations have equal execution time, the list scheduling algorithm, using the optimal sequential schedule as the list, is an approximation algorithm with a performance ratio of 2.

The modified rank of any schedule A is defined as:

$$mrank(A) = \frac{\text{length of schedule for } A}{\text{sum of exit probabilities in } A} \tag{13.14}$$

[30]The notation and terms are borrowed from [27].

The intuition behind the modified rank is that the numerator is the time required to retire A, whereas the denominator is the benefit in retiring A. Thus, the ratio reflects the amount of computational time required per unit of exit probability. Minimizing this ratio in selecting G^* (which uses the modified rank) has the effect of maximizing the "return on investment" in the schedule. Given a graph G, G^* can be found by the following simple procedure.

Algorithm — *Finding G^* under modified rank*:

```
For each branch $b$ of the $G$,
  Construct a list schedule for subgraph $G_b$ rooted at $b$,
    ignoring profile info
    Let $T$ be the length of the schedule and let $W$ be
      sum of exit probabilities of all exits in $G_b$.
    rank($G_b$) = $T/W$
```

G^* is G_b for the earliest b in control order that has the minimum modified rank.

The heuristic converts any list scheduler for precedence graphs, to one that is sensitive to profile information for G. The heuristic takes a profile-insensitive list scheduling algorithm, and bootstraps it to be profile sensitive. To start, the heuristic finds G^* under modified rank, using the insensitive list scheduler. It then makes the list for G^* the initial portion of the list for G. The heuristic deletes G^* from G, and iterates, appending the lists each time until all G is consumed.

Algorithm — *Profile-sensitive scheduler*:

```
1. Profile-list=empty.
2. Find $G^*$ under modified rank using the insensitive scheduler.
3. Append the scheduler list of $G^*$ to Profile-list.
4. Remove $G^*$ from the DAG.
5. If there are branches remaining, then goto step 2.
6. List schedule using Profile-list.
```

13.8.3 Performance Evaluation

Table 13.2 gives the percentage of improvement of combined instruction scheduling and register allocation priority model over the phase-ordered approach using (α, β) algorithm for values of

TABLE 13.2 Execution Time Percentage of Improvement of (α, β)-Combined Algorithm over Phase-Ordered Horizontal Approach

Program	$\alpha = 0$	$\alpha = 0.2$	$\alpha = 0.4$	$\alpha = 0.6$	$\alpha = 0.8$	$\alpha = 1$
a5	−0.3	0.2	0.2	0.2	0.2	0.7
bmm	22.3	23.1	24.1	24.2	24.2	24.2
dag	0.9	0.9	0.9	0.9	0.9	0.9
fir	6.3	6.3	6.4	6.4	6.4	6.4
hyper	3.6	3.6	3.6	3.6	3.6	3.6
idct	5.8	11.7	13.5	13.5	13.5	13.5
mm_double	16.2	19.1	19.9	19.9	19.9	19.9
nbradar	3.9	4.2	4.2	4.2	4.2	4.2
Nested	0.08	0.2	0.3	0.3	0.3	0.3
Polyphase	−1.5	−4.5	5.6	5.9	6.1	6.1
strcpy	0.0	0.0	0.0	0.0	0.0	0.0
Wave	12.8	18.8	19.4	19.4	19.4	19.4

TABLE 13.3 Execution Time Percentage Improvement of Profile-Sensitive Instruction Scheduling with (α, β)-Combined Algorithm over Phase-Ordered Horizontal Approach

Program	$\alpha = 0$	$\alpha = 0.2$	$\alpha = 0.4$	$\alpha = 0.6$	$\alpha = 0.8$	$\alpha = 1$
a5	8.8	4.5	2.1	0.5	0.4	0.2
dag	18.2	18.9	11.1	12.1	23.7	21.8
fir	20.6	19.7	20.4	3.5	−2.3	−2.4
idct	31.1	28.2	25.8	27	21.5	21.2
Nested	15.9	13.0	10.6	10.7	0.2	0.2
Polyphase	6.7	0.9	−3.4	−9.1	−10.1	−3.6
Wave	29.3	27.0	14.8	8.6	9.3	9.3
wc	21.0	21.0	17.2	15.1	10.8	4.3

$\alpha = 0.0, 0.2, 0.4, 0.6, 0.8, 1.0$ on some benchmarks on a machine with 64 GPR, 64 FPR, 2 integer units, 2 floating point units, 2 memory units and 1 branch units with latencies of 2 and 4 for integer and floating point units, 4 for integer and floating multiply, 12 for integer and floating divide, 3 for load, 1 for store and 1 for branch. Performance improvement occurs in most cases with a maximum improvement of 24.2%. Figure 13.17 is a percentage of improvement graph between (α, β)-combined algorithm and phase-ordered horizontal approach. This graph is made by taking the best rank function value for each benchmark in Table 13.2 in order.

Table 13.3 gives the improvement of region-based combined profile-sensitive and register pressure-sensitive instruction scheduling and register allocation model over profile-insensitive horizontal model. Critical path scheduling is used as the profile-insensitive scheduling algorithm to derive the heuristic and compare its performance against critical path scheduling. From Table 13.3, positive performance improvement occurs in most cases with a maximum improvement of 31.1%. A performance improvement of more than 15% in more than 75% of the cases occurs, showing that the (α, β) ranks do a good job of incorporating register pressure sensitivity into the profile-sensitive instruction scheduling algorithm. Figure 13.17 shows the improvement of profile-sensitive instruction scheduling that is register pressure sensitive over the phase-ordered horizontal approach. This graph is drawn by taking the best rank function value for each benchmark in Table 13.3 in order.

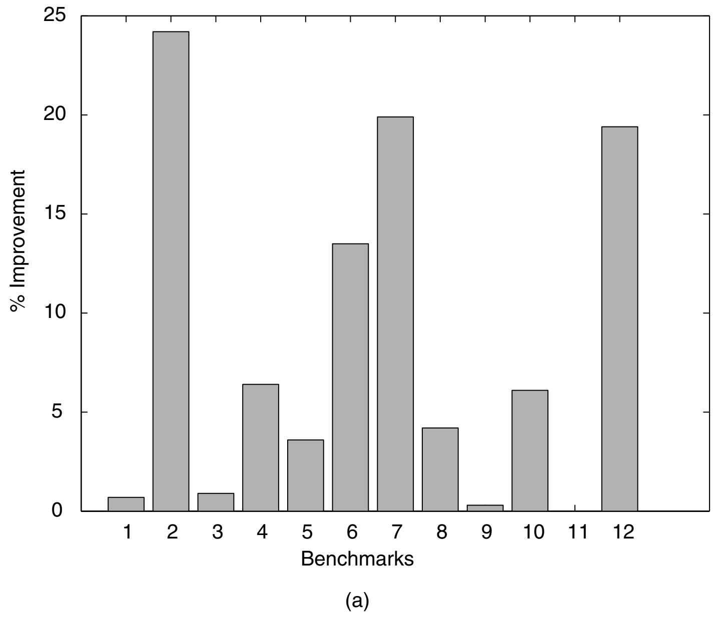
13.9 Case Studies

13.9.1 Gcc

We first discuss version *gcc* 2.7.2.3 (1997) and later discuss changes in the recent version 3.0.2 (2001).

Gcc first does a local nongraph-coloring register allocation. This is followed by a global graph-coloring allocation. Unlike a Chaitin-style graph-coloring allocation, it spills a real register during the reload phase instead of a symbolic register during the simplification phase. Foster and Grossman [32, 71] show that the *gcc* allocator is competitive with the Chaitin-style approach.

In the most basic register allocation (stupid allocation: selected when no optimization flag is chosen), only user-defined variables can go in registers that are declared "register." They are assumed to have a life span equal to their scope. Other user variables are given stack slots in the register transfer list (RTL)-generation pass and are not represented as pseudo registers. A compiler-generated temporary is assumed to live from its first mention to its last mention. Because each pseudo register life span is just an interval, it can be represented as a pair of numbers, each of which identifies an



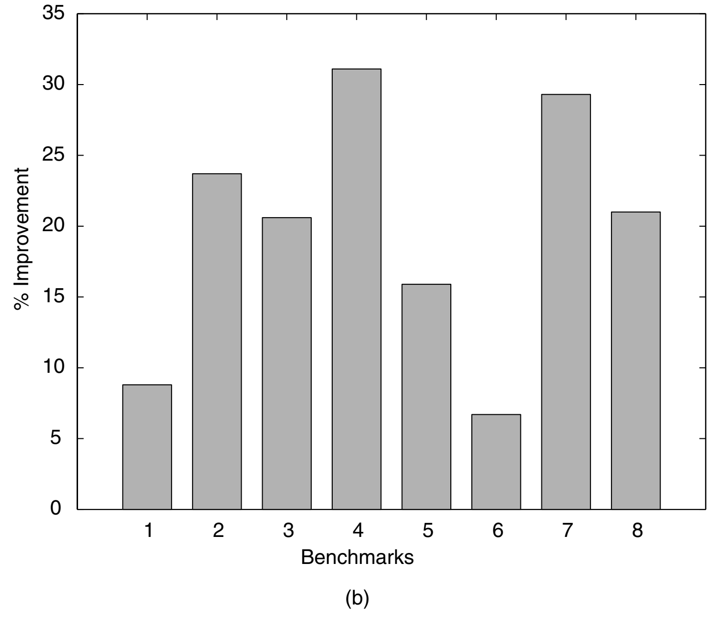


FIGURE 13.17 Execution time percentage of improvement over phase-ordered horizontal approach of (a) (α, β)-combined algorithm; (b) profile-sensitive instruction scheduling that is sensitive to register pressure.

insn by its position in the function (number of insns before it). These pseudo registers are ordered by priority and assigned hard registers in priority order.

If the optimization flag is chosen, it runs the following register allocation-related phases (we also give phases related to instruction scheduling for completeness):

Register scan (regclass.c). This pass finds the first and last use of each register, as a guide for common subexpression elimination (CSE). *Gcc* uses value numbering in its CSE pass.

Register movement (regmove.c). This pass looks for cases where matching constraints would force an instruction to need a reload, and this reload would be a register to register move. It then attempts to change the registers used by the instruction to avoid the move instruction.

Instruction scheduling (sched.c). Instruction scheduling is performed twice. The first time is immediately after instruction combination and the second is immediately after reload. *Gcc* supports both single-issue and multiissue instruction scheduling. It also supports critical path scheduling. Both basic block and critical path scheduling are performed using forward list scheduling algorithms. *Gcc* also supports delay slot scheduling for processors, which have branch and call delay slots.

Register class preferencing (regclass.c). The intermediate code (RTL) is scanned to find out which register class is best for each pseudo register.

Local register allocation (local-alloc.c). This pass allocates hard registers to pseudo registers that are used only within one basic block. It uses weighted counts to allocate values into registers, which live in a single basic block. It also performs coalescing.

Global register allocation (global.c). This pass allocates hard registers for the remaining pseudo registers (those whose life spans are not contained in one basic block) using weighted counts.

Reloading. This pass renumbers pseudo registers with the hardware registers numbers they were allocated. Pseudo registers that did not get hard registers are replaced with stack slots. Then it finds instructions that are invalid because a value has failed to end up in a register, or has ended up in a register of the wrong kind. The technique used is to free up a few hard registers that are called *reload regs*, and for each place where a pseudo register must be in a hard register, copy it temporarily into one of the reload regs.

All the pseudos that were formerly allocated to the hard registers that are now in use as reload registers must be spilled. This means that they go to other hard registers, or to stack slots if no other available hard registers can be found. Spilling can invalidate more insns, requiring additional need for reloads, so we must keep checking until the process stabilizes. *Gcc* attempts to generate optimized spill code whenever possible. For example, it runs a small CSE pass after register allocation and spilling to remove redundant spill code.

The reload pass also optionally eliminates the frame pointer and inserts instructions to save and restore call-clobbered registers around calls. Instruction scheduling is repeated here to try to avoid pipeline stalls due to memory loads generated for spilled pseudo registers. Conversion from usage of some hard registers to usage of a register stack may be done at this point (reg-stack.c). Currently, this is supported only for the FP registers of the Intel 80387 coprocessor that is organized as a stack in the hardware.

Some specialized register-related optimizations in *gcc* are:

Scalar replacement of aggregates involves treating fields in an array, structure or union as unique entities, which can be optimized like normal scalar variables. Of particular importance is the ability to hold aggregates in registers.

Loop unrolling occurs through command line driven loop unrolling with register splitting (variable expansion).

Leaf routine optimizations enable better register allocation with more efficient prologues or epilogues.

Caller-save optimizations support allocation of variables to call-clobbered registers in the presence of calls (with lazy save or restores around call points).

Conditional move or predication support traditional conditional move instructions as well as simple instruction predication (nullification).

The new target-independent register allocation enhancements in *gcc* 3.0.2 are as follows:

Constant equivalence handling for multiply-set pseudos is the same as rematerialization.
Reload inheritance attempts to optimize spill code, particularly unnecessary memory loads or stores caused by register spilling.
New regmove optimizations include improvements to reduce register pressure for 2 address machines such as the IA32, Motorola 68K and Hitachi SH/H8 series. Reduced register pressure leads to more efficient code, particularly in routines where the number of user variables and compiler-generated temporaries is larger than the physical register set for the target processor.
Local spilling in reload and postreload flow analysis involves postreload flow analysis allowing the compiler to detect and eliminate dead code that was created or exposed by register allocation and reloading.

13.9.2 Jalapeno

Jalapeno [58] is a Java virtual machine (JVM) for servers. Its interoperable compilers enable quasi-preemptive thread switching and precise location of object references. The Jalapeno dynamic optimizing compiler is designed to obtain high-quality code for methods that are observed to be frequently executed or computationally intensive. The Jalapeno quick compiler primary register allocator uses Chaitin's graph-coloring algorithm. Coloring is not appropriate (due to long compile time) for some methods (e.g., long one-basic-block static initializers that need many symbolic registers). For such methods, the quick compiler has a simpler, faster algorithm.

However, the optimizing compiler employs the linear scan global register-allocation algorithm [97] to assign physical machine registers to symbolic machine-specific intermediate representation registers. This algorithm is not based on graph coloring, but greedily allocates physical to symbolic registers in a single linear time scan of the symbolic register live ranges. This algorithm is said to be several times faster than graph-coloring algorithms and resulting in code that is almost as efficient.

13.A Appendix: Definitions and Terms

- *Basic block* means maximal branch-free sequence of instructions.
- *Predication* is a feature of many recent architectures in which an instruction is executed only if a predicate that is supplied as part of the instruction is true. Predication [59] has been included in EPIC-style architectures because it helps in many ways. It enables modulo scheduling [92] to reduce code expansion with a schedule that may only have kernel code. More commonly, predication handles branch-intensive programs better because trace scheduling or superblock scheduling cannot handle clusters of traces that should be considered together.
- *Superblock* is a single entry block but with multiple exits. Superblock scheduling avoids compensation code for simplicity, for example, an instruction can move past a side exit E only if any values computed by it are not live out E.
- *Hyperblock* is a predicated version of a superblock. Hyperblock formation substitutes a single block of predicated instructions for a set of basic blocks containing conditional control-flow between those blocks [80].
- *Control tree* gives the nesting of (minimal) intervals obtained by interval analysis on a flow graph [82].
- *Virtual registers* are variables and temporaries that are promotable to reside in a register. These typically exclude variables that can be aliased or whose pointers have been taken.
- *Scratch registers* are registers that are always available for temporary use.
- *Register pressure* means that at a program point, the number of variables are live and potentially assignable to registers.

- *Live range* is the range of instructions, basic blocks, etc. over which a variable is live.
- *Reaching definitions* mean at a program point, all the definitions defined elsewhere are valid.

13.B Appendix: Trimaran Compiler Infrastructure

The Trimaran compiler infrastructure (Figure 13.18) is composed of the following components [98]:

- A machine description facility, MDES, is for describing ILP architectures [42].
- A parametrized ILP architecture is called HPL-PD [70].
- A compiler front end, called IMPACT, is for C. It performs parsing, type checking and a large suite of high-level (i.e., machine-independent) optimizations.
- A compiler back end, called ELCOR, is parametrized by a machine description, performing instruction scheduling, register allocation and machine-dependent optimizations. Each stage of the back end may easily be replaced or modified by a compiler researcher [47].
- An extensible IR (intermediate representation) has both an internal and a textual representation, with conversion routines between the two. The textual language is called Rebel. This IR supports modern compiler techniques by representing control flow, data and control dependence and many other attributes. It is easy to use in its internal representation and its textual representation.
- A cycle-level simulator of the HPL-PD architecture is configurable by a machine description and provides runtime information on execution time, branch frequencies and resource utilization. This information can be used for profile-driven optimizations as well as for provision of validating of new optimizations.
- An integrated graphical user interface (GUI) is used for configuring and running the Trimaran system. Included in the GUI are tools for the graphic visualization of the program IR and of the performance results.

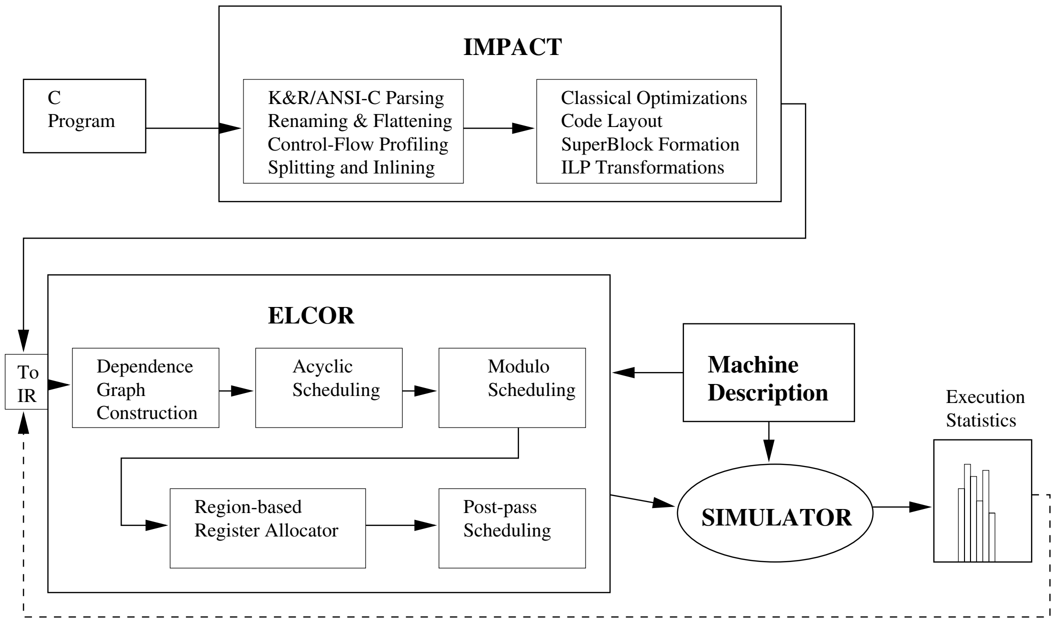


FIGURE 13.18 Trimaran compiler infrastructure.

From a register allocation point of view, the following infrastructure is relevant.

Modulo scheduling of counted loops — A loop scheduler consists of two parts. Modulo scheduler allocates resources for the loop kernel subject to an initiation interval. Stage scheduler moves operations across stages to reduce register usage of the loop. The existing implementation of loop scheduling generates kernel only code, which requires support for predication and rotating register files.

Acyclic scheduling of superblocks and hyperblocks — The three variations of acyclic scheduling in ELCOR are, namely, cycle scheduler, backtracking scheduler and meld scheduler. A cycle scheduler generates a schedule by constructing instructions from operations for each issue cycle in order. A backtracking scheduler is a modified version of the cycle scheduler that can either do limited backtracking to only support scheduling of branch operations with branch delay slots, or do unlimited backtracking. A meld scheduler is a modified version of the cycle scheduler that can propagate operation latency constraints across scheduling region boundaries. This results in tighter scheduling of operations across region boundaries.

Rotating register allocator — When a counted loop is software pipelined, a set of virtual registers in the loop are designated as rotating registers. A rotating register allocator allocates such registers to the rotating register files right after modulo scheduling. Stage scheduling can be used after modulo scheduling to reduce the rotating register requirements of a loop. The remaining registers are allocated to the static register file after scalar scheduling of the rest of the program.

Region-based scalar register allocator — This allocator is based on the Chow and Hennessy approach [68].

References

[1] A. Appel, *Modern Compiler Implementation in C*, Cambridge University Press, London, 1998.

[2] A. Appel and L. George, Optimal Spilling for CISC Machines with Few Registers, SIGPLAN 2001.

[3] A.V. Aho, J. Hopcroft and J.D. Ullman, *Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[4] S. Abraham, V. Kathail and B. Deitrich, *Meld Scheduling: Relaxing Scheduling Constraints Across Region Boundaries*, HPL Technical report HPL-97-39. Hewlett-Packard Laboratories, February 1997.

[5] S. Arora, C. Lund, R. Motwani, M. Sudan and M. Szedgedy, Proof verification and hardness of approximation problems, in *Proceedings 33rd IEEE Symposium on the Foundations of Computer Science*, IEEE Computer Society, Los Angeles, CA, 1992, pp. 14–23.

[6] A. Appel and Kenneth J. Supowit, Generalizations of the Sethi–Ullman algorithm for register allocation, September 1986.

[7] S. Arora and S. Safra, Probabilistic checking of proofs; a new characterization of NP, in *Proceedings 33rd IEEE Symposium on Foundations of Computer Science*, IEEE Computer Society, Los Angeles, CA, 1992, pp. 2–13.

[8] D. Blickstein, P. Craig, C. Davidson, R. Faiman, K. Glossop, R. Grove, S. Hobbs and W. Noyce, The GEM optimizing compiler system, *Digital Equipment Corp. Tech. J.*, 4(4), 121–135, 1992.

[9] Bernstein, Cohen, Lavon, and Rainish, Performance Evaluation of Instruction Scheduling on the IBM RS6000, Proceedings of Micro-25, 1992.

[10] P. Briggs, K. Cooper and L. Torczon, Rematerialization, in ACM SIGPLAN Conference on Programming Language and Design, June 1992, pp. 311–321.

[11] P. Briggs, K. Cooper and L. Torczon, Improvements to graph coloring register allocation, *ACM Program Languages Syst.*, 16(3), 428–455, 1994.

[12] P. Bergner, P. Dahl, D. Engebretsen and M. O'Keefe, Spill code minimization via interference region spilling, in Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation, *SIGPLAN Notices*, 32(6), 287–295, June 1997.

[13] D. Berson, R. Gupta and M.L. Soffa, Integrated instruction scheduling and register allocation techniques, in *Proceedings of the 11th International Workshop on Languages and Compilers for Parallel Computing*, *Lecture Notes in Computer Science*, Springer-Verlag, 1998.

[14] M. Bellare, O. Goldreich and M. Sudan, Free bits, PCP and non-approximability-towards tight results, In *Proceedings 36th IEEE Symposium on Foundations of Computer Science*, IEEE Computer Society, Los Alamitos, CA, 1995, pp. 422–431; updated 1998.

[15] L.A. Belady, A study of replacement algorithms for virtual-storage computer, *IBM Syst. J.*, 5(2), 1966.

[16] P. Briggs, Register Allocation via Graph Coloring, Ph.D. thesis, Rice University, Houston, TX, April 1992.

[17] G.J. Chaitin, M.A. Auslander, A.K. Chandra, J. Cocke, M.E. Hopkins and P.W. Markstein, Register allocation via coloring, *Comput. Languages*, 6, 47–57, 1981.

[18] F.C. Chow and J.L. Hennessy, The priority–based coloring approach to register allocation, *ACM Trans. Programming Languages Syst.*, 12(4), 501–536, 1990.

[19] F. Chow, M. Himmelstein, E. Killian and L. Weber, Engineering a RISC compiler system, in Proceedings COMPCON, March 1986, pp. 132–137.

[20] F. Chow, Minimizing Register Usage Penalty at Procedure Calls, in Conference on Programming Language Design and Implementation (PLDI), 1988, pp. 85–94.

[21] K.D. Cooper, T.J. Harvey and L. Torczon, How to build an interference graph, *Software–Pract. Exper.*, 1998.

[22] D. Callahan and B. Koblenz, Register Allocation via Hierarchical Graph Coloring, in PLDI '91. Proceedings of the Conference on Programming Language Design and Implementation, 1991, pp. 192–203.

[23] S. Carr and K. Kennedy, Scalar Replacement in the Presence of Conditional Control Flow, Technical report TR92283, Rice University, CRPC, Houston, TX, November 1992.

[24] F. Chow, K. Knobe, A. Meltzer, R. Morgan and K. Zadeck. Register Allocation, manuscript, 1994.

[25] K.D. Cooper and J. Lu, Register Promotion in C Programs, in Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation, 1997.

[26] P.P. Chang, S.A. Mahlke, W.Y. Chen, N.J. Warter and W.W. Hsu, IMPACT: An architectural framework for multiple-instruction-issue processors, in Proceedings of the 18th International Symposium on Computer Architecture, May 1991, pp. 266–275.

[27] C. Chekuri, R. Motwani, R. Johnson, B.K. Natarajan, B.R. Rau and M. Schlansker, An Analysis of Profile-Driven Instruction Level Parallel Scheduling with Application to Super Blocks, Proceedings of the 29th Annual International Symposium on Microarchitecture (MICRO-29), Paris, France, December 1996.

[28] R.P. Colwell, R.P. Nix, J.J. O'Donnell, D.B. Papworth and P.K. Rodman, A VLIW Architecture for a Trace Scheduling Compiler, in Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems, April 1987, pp. 180–192.

[29] K. Cooper and T. Simpson, Register Promotion in C Programs, in Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation, 1977.

[30] D. Callahan, S. Carr and K. Kennedy, Improving Register Allocation for Subscripted Variables, in Proceedings of the SIGPLAN 1990 Conference on Programming Language Design and Implementation (PLDI 90), White Plains, NY, June 1990.

[31] R.A. Freiburghouse, Register allocation via usage counts. *Commun. ACM*, 17(11), 638–642, November 1974.

[32] Foster and Grossman, IEEE Southeastconference, April 1992; quoted in [71].

[33] U. Feige, S. Goldwasser, L. Lovasz, S. Safra and M. Szegedy, Approximating Clique is Almost NP-Complete, in Proceedings 32nd IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Los Angeles, CA, 1991, pp. 2–12.

[34] C.W. Fraser and D.R. Hanson, *A Retargetable C Compiler: Design and Implementation*, Benjamin/Cummings, Redwood City, CA, 1995.

[35] Farach and Liberatore, On Local Register Allocation, SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms), 1998.

[36] S. Freudenberger and J. Ruttenberg, Phase ordering of register allocation and instruction scheduling, in Code Generation — Concepts, Tools, Techniques, May 1992, pp. 146–170.

[37] D.W. Goodwin, Optimal and Near-Optimal Global Register Allocation, Ph.D. thesis, University of California — Davis, Davis, CA, 1996.

[38] L. George and A. Appel, Iterated register coalescing, *ACM Trans. Programming Language Syst.*, 18, 300–324, 1996.

[39] Gcc compiler, Free Software Foundation, *www.fsf.org*.

[40] J.R. Goodman and W.C. Hsu, Code Scheduling and Register Allocation in Large Basic Blocks, Proceedings of the IEEE-ACM Supercomputing Conference, 1988.

[41] K. Gopinath and J.L. Hennessy, Copy Elimination in Functional Languages, Proceedings of ACM Symposium on Principles of Programming Languages (POPL), Austin, TX, January 1989.

[42] J.C. Gyllenhaal, W.-M.W. Hwu and B.R. Rau, HMDES Version 2.0 Specification, Technical report IMPACT-96-3, University of Illinois at Urbana-Champaign, 1996.

[43] K. Gopinath, P. Jain and Y. Jain, A Region Based Framework for Combined Register Allocation and Instruction Scheduling for EPIC Architectures, TR No. IISC-CSA-2002-4, CSA, IISc, 2002; *drona.csa.iisc.ernet.in/techrep/years/2002/index.html*.

[44] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman, San Francisco, CA, 1979.

[45] G.J. Chaitin, Register allocation and spilling via graph coloring, in *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, ACM, New York, 1982, pp. 98–105.

[46] M.R. Garey, D.S. Johnson, G.L. Miller and C.H. Papadimitriou, The complexity of coloring circular arcs and chords, *SIAM J. Algebraic Discrete Methods*, 1(2), 216–227, June 1980.

[47] S. Aditya, V. Kathail and B.R. Rau, ELCOR'S Machine Description System: Version 3.0, HPL Technical report HPL-98-128, Hewlett-Packard Laboratories, July 1998.

[48] J. Gillies, R. Ju and Schlansker, Global predicate analysis and its application to register allocation, in Proceedings of the 29th International Workshop on Microprogramming and Microarchitecture, 1996.

[49] D.W. Goodwin and K.D. Wilken, Optimal and near-optimal global register allocation using 0-1 integer programming, *Software Pract. Exp.*, 26(8), 929–965, 1996.

[50] R.E. Hank, Region-Based Compilation, Ph.D. thesis, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, May 1996.

[51] Hsu, Fischer, and Goodman, Minimization of Loads/Stores in Local RA, *IEEE Trans. Software Eng.*, October 89.

[52] H. Kim and K. Gopinath, Efficient Register Allocation through Region-Based Compilation for EPIC Architectures, TR No. IISC-CSA-2002-3, CSA, IISc, 2002. *drona.csa.iisc.ernet.in/˜techrep/years/2002/index.html.*

[53] L.J. Hendren, G.R. Gao, E.R. Altman and C. Mukerji, A Register Allocation Framework Based on Hierarchical Cyclic Interval Graphs, ACAPS Technical Memo 33 (revised), McGill University, February 26, 1993.

[54] R.E. Hank, W.W. Hwu and B.R. Rau, Region-based compilation: introduction, motivation and initial experience, *Int. J. Parallel Programming*, 25(2), 113–146, April 1997.

[55] H. Kim, Ph.D. thesis, Computer Science Department, Courant Institute, New York Institute, New York, 2001.

[56] W.W. Hwu, S.A. Mahlke, W.Y. Chen, P.P. Chang, N.J.Warter, R.A. Bringman, R.G. Ouellette, R.E. Hank, T. Kiyohara, G.E. Haab, J.G. Holm and D.M. Lavery, The superblock: an effective technique for VLIW and superscalar compilation, *J. Supercomputing*, January 1993.

[57] Hennessy and Patterson, *Computer Architecture*, Morgan Kaufmann, San Francisco, 1996.

[58] Alpern et al., The Jalapeno virtual machine, *IBM Syst. J.*, 39(1), 2000.

[59] J.R. Allen, K. Kennedy, C. Porterfield and J. Warren, Conversion of Control Dependence to Data Dependence, in Proceedings of the 10th ACM Symposium on Principles of Programming Languages, 1983, pp. 177–189.

[60] J. Fabri, Automatic Storage Optimization, SIGPLAN '79, 1979.

[61] J. Fisher, Trace scheduling: a general technique for global microcode compaction, *IEEE Trans. Comput.*, C-30(7), 478–490, 1981.

[62] R. Johnson and M. Schlansker, Analysis of Predicated Code, in Micro-29, International Workshop on Microprogramming and Microarchitecture, 1996.

[63] A.B. Kempe, On the geographical problem of four colors, *Am. J. Math.*, 2, 193–200, 1879.

[64] V. Klee, What are the intersection graphs of arcs in a circle?, *Am. Math. Mon.* 76, 810–813, 1969.

[65] P. Klein, A. Agrawal, R. Ravi and Satish Rao, Approximation through multicommodity flows, in *31st Annual Symposium on Foundations of Computer Science*, IEEE, Vol. 2, 1990, pp. 726–737.

[66] K. Cameron, Antichain sequences, *Order*, 2(3), 249–255, 1985; quoted in [67].

[67] S.M. Kurlander and C.N. Fischer, Minimum Cost Interprocedural Register Allocation, in 23rd ACM SIGPLAN SIGACT Symposium on Principles of Programming Languages, St. Petersburg, FL, January 1996, pp. 230–241.

[68] H. Kim, K. Gopinath and V. Kathail, Region Based Register Allocation for EPIC Processors with Predication, Parallel Computing '99, 1999, pp. 36–44.

[69] H. Kim, K. Gopinath, V. Kathail and B. Narahari, Fine-Grained Register Allocation for EPIC Processors with Predication, in International Conference on Parallel and Distributed Processing Techniques and Applications, 1999.

[70] V. Kathail, M. Schlansker and B.R. Rau, HPL-PD Architecture Specification: Version 1.1, Technical report HPL-93-80 (R.1), Hewlett-Packard Laboratories, July 1998.

[71] K. Wilken, Newsgroup posting in comp.compilers (95-11-214).

[72] K. Knobe and A. Meltzer, *Control Tree-Based Register Allocation*, TR, Compass, 1990.

[73] K. Knobe and K. Zadeck, Register Allocation Using Control Trees, Brown University, Providence, RI, CS-92-13, March 1992.

[74] A.-Tabatabai, G.-Y. Lueh and Gross, Global RA Based on Graph Fusion, in LCPC '96, Workshop on Languages and Compilers for Parallel Computing, August 1996.

[75] Lowry and Medlock, Object code optimization, *CACM*, 12, 1, 1969.

[76] G.-Y. Lueh, Issues in RA by Graph Coloring, Technical report CMU-CS-96-171, CMU, 1996.

[77] G.-Y. Lueh, Fusion Based Register Allocation, Ph.D. thesis, CMU, 1997.

[78] C. Lund and M. Yannakakis, On the hardness of approximating minimization problems, in *Proceedings 25th ACM Symposium on Theory of Computing*, ACM, New York, 1993.

[79] M.C. Golumbic, *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, New York, 1980.

[80] S.A. Mahlke, D.C. Lin, W.Y. Chen, R.E. Hank and R.A. Bringmann, Effective Compiler Support for Predicated Execution Using the Hyperblock, in Micro-25, Proceedings of the 25th International Workshop on Microprogramming and Microarchitecture, 1992, pp. 45–54.

[81] R. Motwani, K.V. Palem, V. Sarkar and S. Reyen, Combining Register Allocation and Instruction Scheduling, Technical report, Courant Institute, TR 698, July 1995.

[82] S.S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann, San Francisco, 1997.

[83] M. Poletto and V. Sarkar, Linear Scan Register Allocation, *ACM TOPLAS*, 1999.

[84] S. Novack and A. Nicolau, *Mutation Scheduling: A Unified Approach to Compiling for Fine-Grain Parallelism? Languages and Compilers for Parallel Computing, Lecture Notes in Computer Science*, Vol. 892, Springer-Verlag, 1994.

[85] Norris and Pollock, A Scheduler Sensitive Global Register Allocator, Proceeding Supercomputing '93, OR, 1993.

[86] Norris and Pollock, Register Allocation over the PDG, in PLDI '94. Proceedings of the Conference on Programming Language Design and Implementation, 1994.

[87] N. Shankar, Efficiently Executing PVS, SRI TR, 1999.

[88] T.A. Proebsting and C.N. Fischer, Linear-Time, Optimal Code Scheduling for Delayed-Load Architectures, ACM SIGPLAN Conference on Programming Languages and Design and Implementation, 1991.

[89] T.A. Proebsting and C.N. Fischer, Probabilistic Register Allocation, in PLDI '92. Proceedings of the Conference on Programming Language Design and Implementation, 1992, pp. 300–310.

[90] C.H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithm and Complexity*, Prentice Hall, New York, 1982.

[91] K.V. Palem and B. Simons, Scheduling time-critical instructions on RISC machines, *ACM TOPLAS*, 15(4), 632–658, 1993.

[92] B. Rau, Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops, Proceedings of the 27th Annual Symposium on Microarchitecture, December 1994.

[93] B.R. Rau, M. Lee, P. Tirumalai and M.S. Schlansker, Register Allocation for Software Pipelined Loops, Proceedings SIGPLAN '92 Conference on Programming Language Design and Implementation, San Francisco, June 17–19, 1992.

[94] Soffa, Gupta and Ombres, Efficient register allocation via coloring using clique separators, *ACM TOPLAS*, 16(3), 370–386, May 1994.

[95] P.A. Steenkiste and J.L. Hennessy, A simple interprocedural register allocation algorithm and its effectiveness for LISP, *ACM Trans. Programming Languages Syst.*, 11(1), 1–32, January 1989.

[96] V. Santhanam and D. Odnert, Register Allocation Across Procedure and Module Boundaries, in Proceedings ACM SIGPLAN Conference on Programming Language Design and Implementation 90, White Plains, NY, June 1990.

[97] O. Traub, G. Holloway and M.D. Smith, Quality and Speed in Linear-Scan Register Allocation, in Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation, 1998.

[98] Trimaran, An infrastructure for compiler research in ILP, *www.trimaran.org.*

[99] R.L. Wexelblat, Ed., *History of Programming Languages*, Academic Press, New York, 1981.

[100] D.W. Wall, Global Register Allocation at Link-Time, in Proceedings of SIGPLAN '86 Symposium on Compiler Construction, July 1986, pp. 264–275.

[101] T. Way, B. Breech and L. Pollock, Region Formation Analysis with Demand-Driven Inlining for Region-based Optimization, PACT 2000.

[102] A. Wigderson, Improving the performance guarantee for approximate graph coloring, *J. ACM*, 30, 729–735, 1983.

[103] D. West, *Introduction to Graph Theory*, Prentice Hall, New York, 1996.

[104] M.N. Wegman and F.K. Zadeck, Constant propagation with conditional branches, *TOPLAS*, 13(2), 181–210, April 1991.

14

Architecture Description Languages for Retargetable Compilation

Wei Qin
Princeton University

Sharad Malik
Princeton University

14.1 Introduction

Retargetable compilation has been the subject of some study over the years. This work is motivated by the need to develop a single compiler infrastructure for a range of possible target architectures. Technology trends point to the growth of application-specific programmable systems, which makes it doubly important that we develop efficient retargetable compilation infrastructures. The first need for this is in directly supporting the different target architectures that are likely to be developed to fuel these trends. The second, and possibly more important, need is in the design space exploration for the instruction set architecture and microarchitecture of the processor under development. The evaluation of any candidate architecture needs a compiler to map the applications to the architecture and a simulator to measure the performance. Because it is desirable to evaluate multiple candidates, a retargetable compiler (and simulator) is highly valuable.

The retargetability support largely needs to be provided for the back end of the compiler. Compiler back ends can be classified into the following three types: customized, semiretargetable and retargetable. Customized back ends have little reusability. They are usually written for high-quality proprietary compilers or developed for special architectures requiring nonstandard code generation flow. In contrast, semiretargetable and retargetable compilers aim to reuse at least part of the compiler back end by factoring out target-dependent information into machine description systems. The difference between customization and retargetability is illustrated in Figure 14.1.

In semiretargetable compilers (e.g., lcc [31] and gcc [37]), back ends for different targets share a significant amount of code. This reduces the time needed to port the compilers to new target machines. However, they still require a nontrivial amount of programming effort for each target due

0-8493-1240-7/03/$0.00+$1.50

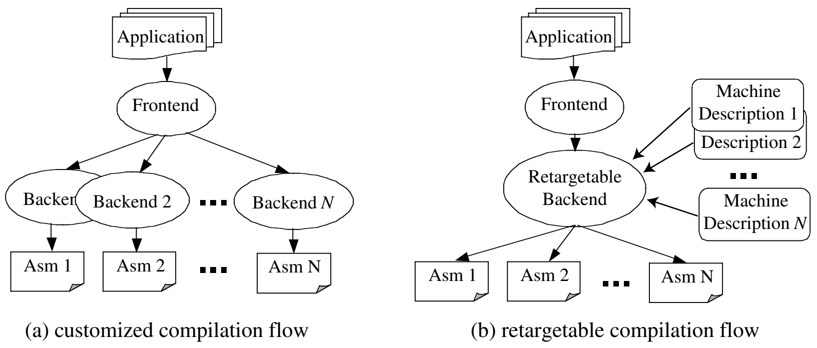


FIGURE 14.1 Illustration of different compilation flows.

to the idiosyncrasies of the target architectures or their calling conventions. In a semiretargetable compiler implementation, the instruction set of the target is usually described in tree patterns as is required by popular code generation algorithms [1]. General-purpose register files and usages of the registers also need to be specified. This type of machine description system serves as the interface between the machine-independent and the machine-dependent part of the compiler implementation. It typically involves a mixture of pattern descriptions and C interface functions or macros. It is the primitive form of an architecture description language.

A fully retargetable compiler is aimed at minimizing the coding effort for a range of targets by providing a friendlier machine description interface. Retargetable compilers are important for application-specific instruction-set processor (ASIP) (including digital signal processor [DSP]) designs. These cost-sensitive processors are usually designed for specialized application domains and have a relatively narrow market window. It is costly for vendors to customize compiler back ends and other software tools for each one of these processors. Moreover, for code density or manufacturing cost concerns, these architectures are often designed to be irregular. Their irregularity prevents general-purpose semiretargetable compilers from generating good quality code because those compilers lack the optimization capability for irregular architectures. As a result, for most ASIP designs, programmers have to fall back on assembly programming that suffers from poor productivity and portability. As architectures with higher degrees of parallelism are gaining popularity, it is increasingly hard for assembly programmers to produce high-quality code. A fully retargetable optimizing compiler is called for to alleviate this software development bottleneck.

To provide sufficient information for optimizing retargetable compiler back ends as well as other software tools including assemblers and simulators, architecture description languages (ADLs) are developed. ADLs also enable design space exploration (DSE) by providing a formalism for modeling processing elements. DSE is important in computer architecture design because no single design is optimal in every aspect. Designers have to try a number of alternative designs and quantitatively estimate or calculate their performance and various other metrics of interest before reaching an acceptable trade-off. The common metrics of interest include power consumption, transistor count, memory size and implementation complexity (it affects time to market). DSE is especially important in the design of ASIPs due to their sensitivity to cost.

Figure 14.2 shows the Y chart [2] of a typical ASIP DSE flow. In such a flow, a designer, or possibly even an automated tool, tunes the architecture description. The retargetable compiler compiles the applications, usually a set of benchmark programs, into architecture-specific code. The simulator executes the code and produces performance metrics such as cycle count and power consumption. The metrics are analyzed and used to guide the tuning of the architecture description. The process iterates until a satisfactory cost-effective architectural trade-off is found. In an embedded system development environment, DSE also helps to determine the optimal combination of hardwired components and

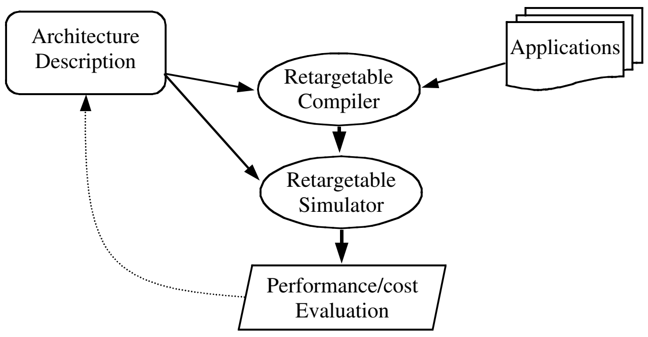


FIGURE 14.2 DSE for ASIP design.

programmable elements. For both purposes, DSE over a reasonably large architecture space is prohibitive if significant manual coding effort to port the compiler and simulator for each target is involved. A retargetable software tool chain driven by an ADL is indispensable for DSE.

Over the past decade, several interesting ADLs have been introduced together with their supporting software tools. These ADLs include MIMOLA [3], UDL/I [4], n-metalanguage (nML) [5], instruction set description language (ISDL) [6], Maril [7], HMDES [8], target description language (TDL) [33], language of instruction set architecture (LISA) [9, 10], Rockwell architecture description language (RADL) [11], EXPRESSION [12], Philips research machine description language (PRMDL) [13] and computer system description language (CSDL) [34]. Usually an ADL-driven tool chain contains a retargetable compiler, an assembler, a disassembler and a functional simulator. In some cases, a cycle accurate simulator and even microarchitecture synthesis tools are included.

In this chapter, we start with a survey of the existing ADLs. Next, we compare and analyze the ADLs to highlight their relative strengths and weaknesses. A characterization of the necessary elements that form an ADL follows, and a generic structural organization of the information within an ADL is given. This study then forms the basis for describing the difficulties faced by ADL designers and users. Finally, we briefly discuss future architecture trends and their implications for requirements of good ADLs.

14.2 Architecture Description Languages

Traditionally, architecture description languages have been classified into three categories: structural, behavioral and mixed. The classification is based on the nature of the information provided by the language.

14.2.1 Structural Languages

An important trade-off in the design of an ADL is the level of abstraction vs. the level of generality. Because of the increasing diversity of computer architectures, it is very difficult to find a formalism at a high abstraction level to capture the interesting characteristics of all types of processors. A common way to obtain a higher level of generality is to lower the abstraction level. A lower abstraction level can capture concrete structural information in more detail. Register transfer (RT) level is a popular abstraction level. It is low enough to model concrete behaviors of synchronous digital logic and high enough to hide gate-level implementation details. It is a formalism commonly used in hardware design. In an RT-level description, data movement and transformation between storage units is specified on a clock cycle basis. Several early ADLs were based on RT-level descriptions.

It is worth noting that the RT-level abstraction should not be confused with the RT lists, though both are often abbreviated to RTL. The latter is a higher level of abstraction used for operation semantics specification. The major difference between the two is the treatment of time. Cycle time is an essential element in an RT-level description. All RT-level events are associated with a specific time stamp and are fully ordered. In contrast, for RT lists, time is not a concern. Only causality is of interest and events are only partially ordered.

14.2.1.1 MIMOLA

One RT-level ADL is MIMOLA [3], an interesting computer hardware description language and high-level programming language developed at the University of Dortmund, Germany. It was originally proposed for microarchitecture design. Over the years, MIMOLA has undergone many revisions and a number of software tools have been developed based on it. The major advantage of a MIMOLA architecture description is that a single description can be used for hardware synthesis, simulation, test generation and code generation purposes. A tool chain including the MSSH hardware synthesizer, the MSSQ code generator, the MSST self-test program compiler, the MSSB functional simulator and the MSSU RT-level simulator were developed based on the MIMOLA language [3]. MIMOLA has also been used by the RECORD [14] compiler as its architecture representation language.

MIMOLA contains two parts: the hardware part where microarchitectures are specified in the form of a component netlist and the software part where application programs are written in a PASCAL-like syntax.

Hardware structures in MIMOLA are described as a netlist of component modules each of which is defined at the RT level. The following is a simple arithmetic unit module, slightly adapted from an example in [3]:

```
MODULE Alu
  (IN i1, i2: (15:0); OUT outp: (15:0));
  IN ctr: (1:0));
CONBEGIN
  outp <— CASE ctr OF
    0: i1 + i2 ;
    1: i1 - i2 ;
    2: i1 AND i2 ;
    3: i1 ;
    END AFTER 1;
CONEND;
```

The module declaration style is similar to that of the Verilog hardware description language [15]. The starting line declares a module named *Alu*. The following two lines describe the module port names, directions and widths. The port names can be used as variables in the behavior statements as references to the signal values on the ports. If more than one statement exists between *CONBEGIN* and *CONEND*, they are evaluated concurrently during execution. The *AFTER* statement in the preceding example describes the timing information.

For a complete netlist, *CONNECTIONS* need to be declared to connect the ports of the module instances, for example [3]:

```
CONNECTIONS    Alu.outp -> Accu.inp
               Accu.outp -> alu.i1
```

The MSSQ code generator can extract instruction set information from the module netlist for use in code generation. It can transform the RT-level hardware structure into a connection operation graph (COG). The nodes in a COG represent hardware operators or module ports whereas the edges represent the data flow through the nodes. The instruction tree (I tree) that maintains a record of the instruction encoding is also derived from the netlist and the decoder module. During code generation, the PASCAL-like source code is transformed into an intermediate representation. Then pattern matching is done from the intermediate code to the COG. Register allocation is performed at the same time. The MSSQ compiler directly outputs binary code by looking up the I tree.

For the compiler to locate some important hardware modules, linkage information needs to be provided to identify important units such as the register file, the instruction memory and the program counter. The program counter and the instruction memory location can be specified as follows [3]:

```
LOCATION_FOR_PROGRAMCOUNTER PCReg;
LOCATION_FOR_INSTRUCTIONS IM[0..1023];
```

where PC and IM are previously declared modules. However, even with the linkage information, it is still a very difficult task to extract the COG and the I trees due to the flexibility of an RT-level structural description. Extra constraints need to be imposed for the MSSQ code generator to work properly. The constraints limit the architecture scope of MSSQ to microprogrammable controllers in which all control signals originate directly from the instruction word [3].

The MIMOLA software programming model is an extension of PASCAL. It is different from typical high-level programming languages in that it allows programmers to designate variables to physical registers and to refer to hardware components through procedures calls. For example, to use an operation performed by a module called Simd, programmers can write

x: = Simd(y,z);

This special feature helps programmers to control code generation and to map intrinsics effectively. Compiler intrinsics are assembly instructions in the form of library functions. They help programmers to utilize complex machine instructions while avoiding writing in-line assembly. Common instrinsics candidates include single instruction multiple data [SIMD] parallel instructions.

14.2.1.2 UDL/I

Another RTL hardware description language used for compiler generation is UDL/I [4] developed at Kyushu University in Japan. It describes the input to the COACH ASIP design automation system. A target-specific compiler can be generated based on the instruction set extracted from the UDL/I description. An instruction set simulator can also be generated to supplement the cycle accurate RT-level simulator. As in the case of MIMOLA, hints need to be supplied by the designer to locate important machine states such as the program counter and the register files. To avoid confusing the instruction set extractor, restrictions are imposed on the scope of supported target architectures. Superscalar and very-long-instruction-word (VLIW) architectures are not supported by the instruction set extractor [4].

In general, RT-level ADLs enable flexible and precise microarchitecture descriptions. The same description can be used by a series of electronic design automation (EDA) tools such as logic synthesis, test generation and verification tools; and software tools such as retargetable compilers and simulators at various abstraction levels. However, for users interested in retargetable compiler development only, describing a processor at the RT level can be quite a tedious process. The instruction set information is buried under enormous microarchitecture details about which only the logic designers care. Instruction set extraction from RT-level descriptions is difficult without restrictions on description style and target scope. As a result, the generality of the RT-level abstraction is very hard for use in the development of efficient compilers. The RT-level descriptions are more amicable to hardware designers than to retargetable compiler writers.

14.2.2 Behavioral Languages

Behavioral languages avoid the difficulty of instruction set extraction by abstracting behavioral information out of the microarchitecture. Instruction semantics are directly specified in the form of RT lists and detailed hardware structures are ignored. A typical behavioral ADL description is close in form to an instruction set reference manual.

14.2.2.1 nML

nML is a simple and elegant formalism for instruction set modeling. It was proposed at the Technical University of Berlin, Germany. It was used by the retargetable code generator CBC [16] and the instruction set simulator SIGH/SIM [17]. It was also used by the CHESS [18] code generator and the CHECKERS instruction set simulator at IMEC. CHESS and CHECKERS were later commercialized [38].

Designers of nML observed that in a real-life processor, usually several instructions share some common properties. Factorization of these common properties can result in a simple and compact representation. Consequently, they used a hierarchical scheme to describe instruction sets. The topmost elements of the hierarchy are instructions, and the intermediate elements are partial instructions (PI). Two composition rules can be used to specify the relationships among the elements: the AND rule that groups several PIs into a larger PI and the OR rule that enumerates a set of alternatives for one PI. Thus, instruction definitions in nML can be in the form of an and-or tree. Each possible derivation of the tree corresponds to an actual instruction.

The instruction set description of nML utilizes attribute grammars [39]. Each element in the hierarchy has a few attributes and the attribute values of a nonleaf element can be computed based on its children's attribute values. Attribute grammar was adopted by ISDL and TDL too.

An example illustrating nML instruction semantics specification is provided as follows:

```
 op num_instruction(a:num_action, src:SRC, dst:DST)
 action {
       temp_src = src;
       temp_dst = dst;
       a.action;
       dst = temp_dst;
}
op num_action = add | sub | mul | div
op add()
action = {
      temp_dst = temp_dst + temp_src
}
...
```

The *num_instruction* definition combines three PIs with the AND rule. The first PI, *num_action*, is formed through an OR rule. Any one of *add, sub, mul* and *div* is a valid option for *num_action*. The number of all possible derivations of *num_instruction* is the product of the size of *num_action*, SRC and DST. The common behavior of all these options is defined in the action attribute of *num_instruction*. Each option of *num_action* should have its own action attribute defined as its specific behavior, which is referred to by the "a.action" line. Besides the action attribute shown in the example, two additional attributes, image and syntax, can be specified in the same hierarchical manner. Image represents the binary encoding of the instructions and syntax is the assembly mnemonic.

Though classified as a behavioral language, nML is not completely free of structural information. Any structural information referred to by the instruction set architecture (ISA) must be provided in the description. For example, storage units should be declared because they are visible to the instruction set. Three storage types are supported in nML: RAM, register and transitory storage. Transitory storage refers to machine states that are retained only for a limited number of cycles, for instance, values on buses and latches. The timing model of nML is simple: computations have no delay. Only storage units have delay. Instruction delay slots can be modeled by introducing storage units as pipeline registers and by propagating the result through the registers in the behavior specification [5].

nML models the constraints between operations by enumerating all valid combinations. For instance, consider a VLIW architecture with three issue slots and two types of operations A and B. If, due to some resource contention, at most one type A operation can be issued at each cycle, then the user has to enumerate all the possible issue combinations including ABB, BAB, BBA and BBB. The enumeration of all the valid cases can make nML descriptions lengthy. More complicated constraints, which often appear in DSPs with irregular instruction level parallelism (ILP) constraints or VLIW processors with a large number of issue slots, are hard to model with nML. For example, nML cannot model the constraint that operation X cannot directly follow operation Y.

14.2.2.2 ISDL

The problem of constraint modeling is tackled by ISDL with explicit specification. ISDL was developed at Massachusetts Institute of Technology (MIT) and used by the Aviv compiler [19] and the associated assembler. It was also used by the retargetable simulator generation system GENSIM [20]. ISDL was designed to assist hardware–software codesign for embedded systems. Its target scope is VLIW ASIPs.

ISDL mainly consists of five sections:

- Storage resources
- Instruction-word format
- Global definition
- Instruction set
- Constraints

Similar to the case of nML, storage resources are the only structural information modeled by ISDL. The register files, the program counter and the instruction memory must be defined for each architecture. The instruction–word format section describes the composing fields of the instruction–word. ISDL assumes a simple treelike VLIW instruction model: the instruction–word contains a list of fields and each field contains a list of subfields. Each field corresponds to one operation. For VLIW architectures, an ISDL instruction corresponds to an nML instruction and an ISDL operation corresponds to a PI in nML.

The global definition section describes the addressing modes of the operations. Production rules for tokens and nonterminals can be defined in this section. *Tokens* are the primitive operands of instructions. For each token, assembly format and binary encoding information must be defined. An example token definition of a register operand is:

```
Token ''RA''[0..15] RA {[0..15];}
```

In this example, following the keyword *Token* is the assembly format of the operand. Here any one of RA0 to RA15 is a valid choice. Next, RA is the name of the token for later reference. The second [0..15] is the return value. It is to be used for behavioral definition and binary encoding assignment by nonterminals or operations.

Nonterminal is a mechanism provided to exploit commonalities among operations. It can be used to describe complex addressing modes. A nonterminal typically groups a few tokens or other nonterminals as its arguments, whose assembly formats and return values can be used when defining the nonterminal. An example nonterminal specification is:

```
Non_Terminal SRC:
     RA  {$$ = 0x00000 | RA;} {RFA[RA]} {} {} {} |
     RB  {$$ = 0x10000 | RB;} {RFB[RB]} {} {} {};
```

The example shows a nonterminal named *SRC*. It refers to token RA and RB as its two options. The first pair of braces in each line defines binary encoding as a bit-or result. The $$ symbol indicates the return value of the current nonterminal, a usage probably borrowed from Yacc [21]. The second pair of braces contains the action. Here the *SRC* operand refers to the data value in the register RFA[RA] or RFB[RB] as its action. The following three empty pairs of braces specify side effects, cost and time. These three exist in the instruction set section as well.

The instruction set section of ISDL describes the instruction set in terms of its fields. For each field, a list of alternative operations can be described. Similar to nonterminals, an operation contains a name, a list of tokens or nonterminals as its arguments, the binary encoding definition, the action definition, the side effects and the costs. Side effects refer to behaviors such as the setting or clearing of a carry bit. Three types of cost can be specified: execution cycles, encoding size and stall. Stall models the cycle number of pipeline stalls if the next instruction uses the result of the current instruction. The timing model of ISDL contains two parameters: latency and usage. Latency is the number of instructions to be fetched before the result of the current operation becomes available and usage is the cycle count that the operation spends in its slot. The difference or the relationship between the latency and the stall cost is not clear in the available publications of ISDL.

The most interesting part of ISDL is its explicit constraint specification. In contrast to nML, which enumerates all valid combinations, ISDL defines invalid combinations in the form of Boolean expressions. This often results in a much simpler constraint specification. It also enables ISDL to capture much more irregular ILP constraints. Recall the constraint example that instruction X cannot directly follow Y, which cannot be modeled by nML. ISDL can describe the constraint as follows [6]:

```
~(Y *) & ([1] X *, *)
```

The “[1]” indicates a cycle time delay of one fetch bundle. The “∼” is a symbol for not and “&” for logical and. Such constraints cannot be specified by nML. Details of the Boolean expression syntax are available in the ISDL publication [19]. The way ISDL models constraints affects the code generation process: a constraint checker is needed to check whether the selected instructions meet the constraint. Iterative code generation is required in case of checking failure.

Overall, ISDL provides the means for compact, hierarchical instruction set specification of instruction sets. Its Boolean expression based constraint specification helps to model irregular ILP effectively. A shortcoming of ISDL is that the simple treelike instruction format forbids the description of instruction sets with multiple encoding formats.

14.2.2.3 CSDL

CSDL [34] is a family of machine description languages developed for the Zehpyr compiler infrastructure mainly at the University of Virginia. The CSDL family currently includes a function calling convention specification language (CCL) [22], specification language for encoding and decoding (SLED) [35], a formalism describing instruction assembly syntax and binary encoding and λ-RTL [36], a register transfer lists language for instruction semantics description.

SLED was developed as part of the New Jersey Machine-Code toolkit that assists programmers to build binary editing tools. A retargetable linker mld [42] and a retargetable debugger ldb [43] based on the toolkit have been reported.

Similar to ISDL, SLED uses a hierarchical model for machine instructions. An instruction is composed of one or more tokens and each token is composed of one or more fields. The directive patterns help to group the fields together and to bind them to binary values. The directive constructor helps to connect the fields into instruction words. A detailed description of the syntax can be found in SLED publications [35].

SLED does not assume a single format of the instruction set. Thus, it is flexible enough to describe the encoding and assembly syntax of both reduced instruction set computer (RISC) and complex instruction set computer (CISC) instruction sets. Like nML, SLED enumerates legal combinations of fields. There is neither a notion of hardware resources nor explicit constraint descriptions. Therefore, without significant extension, SLED is not suitable for use in VLIW instruction-word description.

The very portable optimizer (vpo) in the Zephyr system provides the capability of instruction selection, instruction scheduling and global optimization. Instruction sets are represented in RT-lists form in vpo. The raw RT-lists form is verbose. To reduce description effort, λ-RTL was developed. A λ-RTL description can be translated into RT lists for the use of vpo.

According to the developers [36], λ-RTL is a high-order, strongly typed, polymorphic, pure functional language based largely on Standard ML [40]. It has many high-level language features such as name space (through the module and import directives) and function definition. Users can even introduce new semantics and precedence to basic operators. This functional language has several elegant properties [36], that are beyond the scope of this chapter.

CSDL descriptions describe only storage units as hardware resources. Timing information such as operation latencies is not described. Thus, the languages themselves do not supply enough information for VLIW processor code generation and scheduling. They are more suitable for conventional general-purpose processor modeling.

The behavioral languages share one common feature: hierarchical instruction set description based on attribute grammars [39]. This feature helps to simplify the instruction set description by exploiting the common components between operations. However, the lack of detailed pipeline and timing information prevents the use of these languages as an extensible architecture model. Information required by resource-based scheduling algorithms cannot be obtained directly from the description. Also, it is impossible to generate cycle accurate simulators based on the behavioral descriptions without some assumptions of the architecture control behavior (i.e., an implied architecture template has to be used).

14.2.3 Mixed Languages

Mixed languages extend behavioral languages by including abstract hardware resources in the description. As in the case of behavioral languages, RT lists are used in mixed languages for semantics specification.

14.2.3.1 Maril

Maril is a mixed architecture description language used by the retargetable compiler Marion [7]. It contains both instruction set information as well as coarse-grained structural information. Maril descriptions are intended for use in code generation for RISC-style processors only. So unlike the case with ISDL, no distinction is made between an instruction and an operation in Maril. The structural information contains storage units as well as highly abstract pipeline units. The compiler back ends for the Motorola 88000 [44], the MIPS R2000 [45] and the Intel i860 [46] architectures were developed based on Maril descriptions.

Maril contains three sections:

14.2.3.1.1 *Declaration.*

The declaration section describes structural information such as register files, memory and abstract hardware resources. Abstract hardware resources include pipeline stages and data buses. The resources are used for reservation table description. In the compiler community, the term *reservation table* has been used as a mapping from operation to architecture resources and time. It captures the operation resource usage at every cycle starting with the fetch. The reservation table is often a one-to-many mapping because there can be multiple alternative paths for one instruction.

Beside hardware structures, the declaration section also contains information such as the range of immediate operands or relative branch offset. This is necessary information to generate correct code.

14.2.3.1.2 *Runtime Model.*

The runtime model section specifies the conventions of the generated code. It deals mostly with the function calling convention. This section is the parameter system of the Marion compiler. It is not intended to be a general framework for calling convention specification that is complex enough to qualify as a description language by itself [22]. Calling conventions are not the primary interest of this chapter and are not discussed further.

14.2.3.1.3 *Instruction.*

The instruction section describes the instruction set. Each instruction definition in Maril contains five parts. The first part is the instruction mnemonics and the operands that can be used to format assembly code. The optional second part declares data-type constraints of the operation for code selection use. The third part describes, for each instruction, a single expression. The patterns used by the tree-pattern matching code generator are derived from this expression. A limitation on the expression is that it can contain only one assignment. This limitation is reasonable because the code generator can usually handle tree-patterns only. However, it forbids Maril from describing instruction behavior such as side effects on machine flags and postincrement memory references, because those involve multiple assignments. Consequently, a Maril description is generally not accurate enough for use by a functional simulator. The fourth part of the instruction declaration is the reservation table of the instruction. The abstract resources used in each cycle can be described here, starting from instruction fetch. The last part of an instruction specification is a triple of (cost, latency, slot). Cost is used to distinguish actual operations from dummy instructions that are used for some type conversions. Latency is the number of cycles before the result of this instruction can be used by other operations. Slot specifies the delay slot count. Instruction encoding information is not provided in Maril.

An example of the integer Add instruction definition of Maril is as follows [7]:

```
%instr Add r, r, r (int)
       {$3 = $1+$2;}
       {IF; ID; EX; MEM; WB} (1,1,0)
```

The operands of the *Add* instruction are all general-purpose registers, as denoted by the *r*'s. The ";" in the reservation table specification delimits the cycle boundary. The *Add* instruction goes through five pipeline stages in five cycles. It has a cost of one, takes one cycle and has no delay slot.

In general, Maril is designed for use in RISC processor code generation and scheduling. Some of its information is tool specific. It cannot describe a VLIW instruction set, and does not provide enough information for accurate simulation. It does not utilize a hierarchical instruction set description scheme as is done by most behavioral languages. Nonetheless, compared with the behavioral languages, it carries more structural information than just storage units. The structural resource-based reservation table description enables resource-based instruction scheduling, which can bring significant performance improvement for deeply pipelined architectures.

14.2.3.2 HMDES

Another language with emphasis on scheduling support is HMDES [8] developed at UIUC for the IMPACT research compiler. IMPACT mainly focuses on ILP exploration. As a result, the instruction reservation table information is the major content of HMDES. Although there is no explicit notion of instruction or operation bundle in HMDES, it can be used for VLIW scheduling purposes by representing VLIW issue slots as artificial resources. The designers of IMPACT are interested in wide issue architectures in which a single instruction can have numerous scheduling alternatives. For example, if in an 8-issue architecture an *Add* instruction can go through any of the 8 decoding slots into any of the 8 function units, there are in total 64 scheduling alternatives for it. To avoid laboriously enumerating the alternatives, an and-or tree structure is used in HMDES to compress reservation tables. Figure 14.3 shows the reservation table description hierarchy of HMDES. The leaf node resource usage is a tuple of (resource, time).

A special feature of HMDES is its preprocessing constructs. C-like preprocessing capabilities such as file inclusion, macroexpansion and conditional inclusion can be used. Complex structures such as loop and integer range expansion are also supported. The preprocessing does not provide extra description power, but it helps to keep the description compact and easy to read.

Instruction semantics, assembly syntax and binary encoding information are not part of HMDES because IMPACT is not designed to be a fully retargetable code generator. It has a few manually written code generation back ends. After machine specific code is generated, IMPACT queries the machine database built through HMDES to do ILP optimization.

An HMDES description is the input to the MDES machine database system of the Trimaran compiler infrastructure that contains IMPACT as well as the ELCOR research compiler from HP Labs. The description is first preprocessed, and then optimized and translated to a low-level representation file. A machine database reads the low-level files and supplies information to the compiler back end through a predefined query interface. A detailed description of the interface between the machine description system and the compiler back end can be found in the documentation for ELCOR [23].

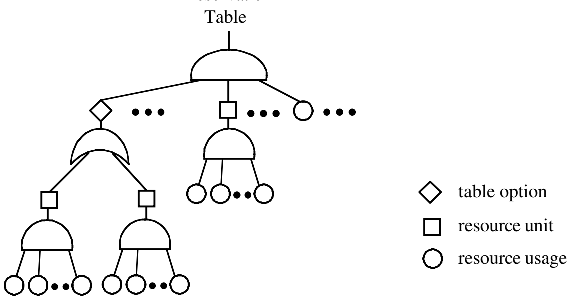


FIGURE 14.3 HMDES reservation table description hierarchy.

14.2.3.3 TDL

TDL has been developed at Saarland University in Germany. The language is used in a retargetable postpass assembly-based code optimization system called PROPAN [33].

In PROPAN, a TDL description is transformed into a parser for target-specific assembly language and a set of ANSI C files containing data structures and functions. The C files are included in the application as a means of generic access to architecture information. A TDL description contains four parts: resource section, instruction set section, constraints section and assembly section.

14.2.3.3.1 *Resource Section.*

Storage units such as register files and memory have built-in syntax support in this section. Moreover, TDL allows the description of the cache, which partially exposes the memory hierarchy to the compiler and allows for more optimization opportunities. Function units are also described in this section. TDL also provides flexible syntax support for the user to define generic function units that do not belong to any predefined category.

14.2.3.3.2 *Instruction Set Section.*

Like those of the behavioral languages, the TDL organization is based on attribute grammars [39]. TDL supports VLIW architectures, thus distinguishing the notions of operation and instruction. No binary encoding information is provided by TDL.

An example of a TDL operation description adapted from [50] is:

```
DefineOp IAdd "%s = %s + %s"{
    dst1 = "$1"in{gpr}, src1 = "$2"in{gpr}, src2 = "$3"in{gpr}},
    {ALU1(latency=1, slots=0, exectime=1)
      |ALU2(latency=1, slots=0, exectime=2);
        WbBus(latency=1)},
    {dst1 := src1 + src2;}.
```

The operation definition contains the name, the assembly format, a list of the predefined attributes such as source and destination operand position and type, the reservation table and the semantics in RT-lists form. In this example, the destination operand $1 corresponds to the first "%s" in the assembly format and is from the register file named *gpr*. The operation can be scheduled on either function unit *ALU1* or *ALU2*. It also uses the result write back bus (*WbBus*) resource. The operation performs addition. A very detailed TDL version of the RT-lists semantics description rule can be found in [50].

This section also contains operation class definition that groups operations into groups for the ease of reference. Instruction-word formats can be described using the operation classes. For instance, the declaration [50] "InstructionFormat ifo2 [opclass3, opclass4];" means that the instruction format *ifo*2 contains one operation from *opclass*3 and one from *opclass*4.

Similar to ISDL, TDL also provides a nonterminal construct to capture common components between operations.

14.2.3.3.3 *Constraints Section.*

Recall that ISDL has a constraint specification section. The Boolean expression used in ISDL is based on lexical elements in the assembly instruction. TDL also uses Boolean expressions for constraint modeling, but the expressions are based on the explicitly declared operation attributes from the preceding sections.

A constraint definition includes a premise part followed by a rule part, separated by a colon. An example definition from [50] is as follows:

```
op1 in {MultiAluFixed} & op2 {MultiMulFixed} :
    !(op1 && op2) | op1.src1 in {iregC} & op1.src2 in {iregD}
   &op2.src1 in {iregA} & op2.src2 in {iregB}
```

The example specifies the issue constraint for two operations, one from the *MultiAluFixed* class and the other from the *MultiMulFixed class*. Either they do not coexist in one instruction-word or their operands have to be in specific register files. The constraint specification is as powerful as that of ISDL and has a cleaner syntax. The Boolean expression can be transformed into an integer linear programming constraint to be used in the PROPAN system.

14.2.3.3.4 *Assembly Section.*

This section describes the lexical elements in the assembly file including comment syntax, operation and instruction delimiters and assembly directives.

On the whole, TDL provides a well-organized formalism for VLIW DSP assembly code description. Preprocessing is supported. Description of various hardware resources including caches is also supported. The RT-lists description rules are exhaustively defined. A hierarchical scheme is used to exploit common components among operations. Both resource based (function units) and explicit Boolean expression based constraint modeling mechanisms are provided. However, the timing model and the reservation table model of TDL seem to be restricted by the syntax. No mechanism is available for users to flexibly specify operand latencies — the cycle time when operands are read and written. These restrictions prevent the use of TDL for accurate RISC architecture modeling. Another limitation is that register ports or data transfer paths are not explicitly modeled in the resource section. The two are often resource bottlenecks in VLIW DSPs. The limitations can be overcome with extensions to the current TDL.

A related but more restrictive machine description formalism for assembly code analysis and transformation applications can be found in the SALTO framework [48]. The organization of the SALTO machine description is similar to that of TDL.

14.2.3.4 EXPRESSION

A problem of an explicit reservation table description in the preceding mixed languages is that it may not be natural and intuitive enough. A translation from pipeline structures to abstract resources has to be done by description writers. Such manual translation can be annoying in the case of DSE. The architecture description language EXPRESSION [12] avoids human effort in doing the translation. Instead, it describes a netlist of pipeline stages and storage units directly and automatically generates reservation tables based on the netlist. In contrast to MIMOLA that uses fine-grained netlists, EXPRESSION uses a much coarser representation. A netlist-style specification is friendly to architects and makes graphic input possible.

EXPRESSION was developed at University of California at Irvine. It is used by the research compiler EXPRESS [24] and the research simulator SIMPRESS [25] developed there. A GUI front end for EXPRESSION has also been developed [25]. EXPRESSION takes a balanced view of behavioral and structural descriptions and consists of a distinct behavioral section and a structural section.

The behavioral section is similar to that of ISDL in that it distinguishes instructions and operations. However, it does not cover assembly syntax and binary encoding and does not use a hierarchical structure for instruction semantics. The behavioral section contains three subsections: operation, instruction and operation mapping. The operation subsection is in the form of RT lists. Detailed semantics description rules are not publicly available. A useful feature of EXPRESSION is that it groups similar operations together for ease of later reference. Operation addressing modes are also defined in the operation subsection.

The instruction subsection describes the bundling of operations that can be issued in parallel. Instruction width, slot widths and function units that correspond to the slots are declared in this section. This information is essential for VLIW modeling.

The operation mapping subsection specifies the transformation rules for code generation. Two types of mapping can be specified: mapping from a compiler's intermediate generic operations to target-specific operations and mapping from target-specific operations to target-specific operations. The first type of mapping is used for code generation. The second type is required for optimization purposes. Predicates can be specified for conditional mappings. Providing the mapping subsection in EXPRESSION makes the code generator implementation much easier, but it also makes the EXPRESSION language tool dependent. The most interesting part of EXPRESSION is its netlist based structural specification. It contains three subsections: component declaration, path declaration and memory subsystem declaration.

In the component subsection, coarse-grained structural units such as pipeline stages, memory controllers, memory banks, register files and latches are specified. Linkage resources including ports and connections can also be declared in this part. A pipeline stage declaration for the example architecture in Figure 14.4 is:

```
(DECODEUnit ID
      (LATCHES decodeLatch)
      (PORTS ID_srcport1 ID_srcport2)
      (CAPACITY 1)
      (OPCODES all)
      (TIMING all 1)
)
```

The example shows the declaration of the instruction decoding stage. The *LATCHES* statement refers to the output pipeline register of the unit. The *PORTS* statement refers to the abstract data ports of the unit. Here the *ID* unit has two register read ports. *CAPACITY* describes the number of instructions that the pipeline stage can hold at a time. Common function units have a capacity of one, whereas fetching and decoding stages of VLIW or superscalar processors can have a capacity as wide as its issue width. *OPCODES* describes the operations that can go through this stage. *All* here refers to an operation group containing all operations. *TIMING* is the cycle count that operations spend in the unit. Each operation takes one cycle. *TIMING* can also be described on a per operation basis.

In the path subsection, pipeline paths and data paths between pipeline stages and storage units are specified. This part connects the components together into a netlist. The pipeline path declaration stitches together the pipeline stages to form a directed acyclic pipeline graph, in which the pipeline stages are vertices and paths are directed edges. An example path declaration for the simple DLX architecture [26] shown in Figure 14.4 is:

```
(PIPELINE FE ID EX MEM WB)

(PIPELINE FE ID F1 F2 F3 F4 WB)
```

Recall that the *OPCODES* attribute of pipeline stages declares the operations that can go through each stage. So for each operation, the possible paths that it can go through can be inferred by looking for paths every node of which can accommodate the operation. Each path corresponds to a scheduling alternative. Because the time spent by each operation at each stage is specified in the *TIMING* declaration in the component subsection, a reservation table can be generated from the paths.

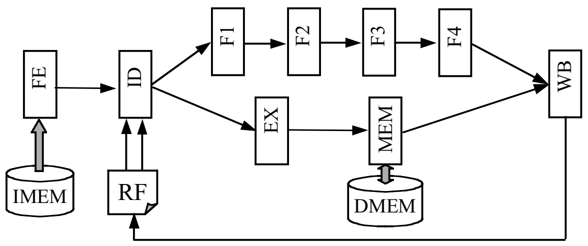


FIGURE 14.4 Example DLX pipeline.

Register file ports and data transfer path usage information can also be derived from the operands of the operations and the connections between the pipeline stages and the storage units. The ports and the data transfer paths can also be resources in the reservation tables.

Compared with the explicit reservation table specification style of HMDES and Maril, the EXPRESSION netlist style is more attractive. However, the way that EXPRESSION derives reservation tables [12] excludes cases in which an operation can occupy two pipeline stages in the same cycle. This may happen in floating point pipelines or when artificial resources [27] need to be introduced for irregular ILP constraint modeling.

The last structural part is the memory model. A parameterized memory hierarchy model is provided in EXPRESSION. Memory hierarchy specification is important, especially as we go from single processing element descriptions toward system-on-chip descriptions. Advanced compilers can make use of the memory hierarchy information to improve cache performance. EXPRESSION and TDL are the only ADLs that address memory hierarchy.

In general, EXPRESSION captures the data path information in the processor. However, as with all the preceding mixed languages, EXPRESSION does not explicitly model the control path. Thus, it actually does not contain complete information for cycle accurate simulation. An architecture template is needed to supplement the control path information for simulation purposes. The behavioral model of EXPRESSION does not utilize hierarchical operation specification. This can make it tedious to specify a complete instruction set. The VLIW instruction composition model is simple. It is not clear if interoperation constraints such as sharing of common fields can be modeled. Such constraints can be modeled by ISDL through cross-field encoding assignment.

14.2.3.5 LISA

The emphasis of the behavioral languages is mostly on instruction set specification. Beyond that, mixed languages provide coarse-grained data path structural information. However, control path specification is largely ignored by both the behavioral languages and the preceding mixed languages. This is probably due to the lack of formalism in control path modeling. Complex control-related behaviors such as speculation and zero-overhead loops are very difficult to model cleanly. Control behavior modeling is important for control code generation (e.g., branch and loop code generation), as well as for cycle-accurate simulation. As the pipeline structures of high-end processors get increasingly complicated, branching and speculation can take up a significant portion of program execution time. Correct modeling of control flow is crucial for accurate simulation of such behaviors with feedback that can provide important guidance to ILP compiler optimizations.

The architecture description language LISA [9] was initially designed with a simulator centric view. It was developed at Aachen University of Technology in Germany. The development of LISA accompanies that of a production quality cycle-accurate simulator [10]. A compiler based on LISA is undergoing development at this moment.

Compared with all the preceding languages, LISA is closer to an imperative programming language. Control-related behavior such as pipeline flush or stall can be specified explicitly. These operation primitives provide the flexibility to describe a diversity of architectural styles such as superscalar, VLIW and SIMD.

LISA contains mainly two types of top-level declarations: resource and operation. Resource declarations cover hardware resources including register files, memories, program counters, pipeline stages, etc. The *PIPELINE* definition declares all the possible pipeline paths that the operations can go through. A *PIPELINE* description corresponding to the example shown in Figure 14.4 is as follows:

```
PIPELINE pipe = {FE; ID; EX; MEM; WB}

PIPELINE pipe_fp = {FE; ID; F1; F2; F3; F4; WB}
```

Similar to the case of Maril, the preceding ";" symbols are used to delimit the cycle boundary.

Machine operations are defined stage by stage in LISA. *OPERATION* is the basic unit defining operation behavior, encoding, and assembly syntax on a per stage basis. At some pipeline stages such as the instruction fetch or decoding stage, several operations may share some common behavior. LISA exploits the commonality by grouping the specifications of similar operations into one. As a slightly modified version of an example in [10], the decoding behavior for arithmetic operations at the DLX *ID* stage can be described as follows:

```
    OPERATION arithmetic IN pipe.ID {
         DECLARE {
              GROUP opcode={ADD || ADDU || SUB || SUBU}
              GROUP rs1, rs2, rd = {fix_register};
    }
    CODING {opcode rs1 rs2 rd}
    SYNTAX {opcode rd ``,'' rs1 ``,'' rs2}
    BEHAVIOR {
         reg_a = rs1;
         reg_b = rs2;
         cond = 0;
    }
    ACTIVATION {opcode, writeback}
}
```

The example captures the common behavior of the four arithmetic operations *ADD, ADDU, SUB* and *SUBU* in the decoding stage and their common assembly syntax and binary encoding.

To capture the complete behavior of an operation, its behavior definition in other pipeline stages has to be taken into account. As shown in the example, an operation declaration can contain several parts:

- *DECLARE*, where local identifiers are specified
- *CODING*, where the binary encoding of the operation is described
- *SYNTAX*, where the assembly syntax of the operation is declared
- *BEHAVIOR*, where the exact instruction semantics including side effects are specified in a C-like syntax
- *ACTIVATION*, where the dependence relationship of the other *OPERATION*s is specified

An extra *SEMANTICS* part can be defined too. Although both *SEMANTICS* and *BEHAVIOR* define the function performed by an operation, *SEMANTICS* is reserved for use by the code generator during mapping and *BEHAVIOR* is for use in simulation. Recall that similar redundancy can be found in *EXPRESSION* in which both operation subsection and mapping subsection are dedicated to semantics specification.

ACTIVATION is part of the execution model of LISA. In LISA, one *OPERATION* can activate another *OPERATION*. The firing time of the activated *OPERATION* depends on the distance between the two *OPERATION*s in the pipeline path. The preceding example activates one of the opcodes that is declared at the *EX* stage. Because *EX* directly follows *ID*, it executes in the following cycle. One opcode *ADD* can be declared as follows [10]:

```
OPERATION ADD IN pipe.EX
{
     CODING {0b001000}
     SYNTAX {''ADD''}
     BEHAVIOR {alu = reg_a + reg_b;}

}
```

Each LISA description contains a special main *OPERATION* that can be activated in every cycle. It serves as a kernel loop in simulation. Pipeline control functions including shift, stall and flush can be used in the main loop.

One advantage of LISA is that the user can describe detailed control path information with the activation model. Control path description is important in generating a cycle accurate simulator. A fast and accurate simulator for the Texas Instrument (TI) TMS320C6201 VLIW DSP [28] based on LISA has been reported [10].

To use LISA for retargetable code generation, the instruction set has to be extracted. This is an easier job than instruction extraction from RT-level languages. The semantics of most arithmetic, logical and memory operations can be directly obtained by combining their *OPERATION* definitions at different stages. For complicated and hard-to-map instruction behaviors, the *SEMANTICS* keyword can be used as a supplement.

14.2.3.6 Miscellaneous

A language similar to LISA is RADL [11]. It was developed at Rockwell, Inc. as a follow-up of some earlier work on LISA. The only goal of RADL is to support cycle accurate simulation. Control-related APIs such as stall and kill can also be used in RADL descriptions. However, no information is available on the simulator utilizing RADL.

Beside RADL, another architecture description language developed in the industry is PRMDL [13]. PRMDL is intended to cover a range of VLIW architectures, including clustered VLIW architectures in which incomplete bypass networks and functional units are shared among different issues slots [13]. A PRMDL description separates the software view (the virtual machine) and the hardware view (the physical machine) of the architecture. The separation ensures code portability by keeping application programs independent of changes in hardware. However, information is not available on how PRMDL maps elements in the virtual machine to the physical machine.

14.3 Analysis of Architecture Description Languages

14.3.1 Summary of Architecture Description Languages

ADLs as a research area are far from mature. In contrast to the situation with other computer languages such as programming languages or hardware description languages, neither any kind of standard nor any dominating ADL exists. Most of the ADLs are designed specifically for software tools under development by the same group of designers and are thus tightly affiliated with the tools. The only languages that have been used by multiple compilers are MIMOLA and nML. However, they have limited popularity, and have no recent adopters. The stage is wide open for new efforts and it is not surprising that new ADLs are constantly emerging.

An apparent reason for the lack of an ADL standard is the lack of formalism in general computer architecture modeling. Modern computer architectures range from simple DSP and ASIPs with small register files and little control logic to complex out-of-order superscalar machines [26] with deep memory hierarchy and heavy control and data speculation. As semiconductor processes evolve, more and more transistors are squeezed onto a single chip. The drive for high performance leads to the birth of even more sophisticated architectures such as multithreading processors [26]. It is extremely hard for a single ADL to describe the vast range of computer architectures accurately and efficiently. A practical and common approach is for the designers to use a high-level abstraction committed to a small range of architectures. Usually one or more implicit architecture templates are behind a high-level ADL to bridge the gap between the abstraction and the physical architecture.

A second reason is probably the different design purposes of ADLs. Some ADLs were originally designed as a hardware description language (e.g., MIMOLA and UDL/I). The major design goal of those languages was accurate hardware modeling. Cycle accurate simulators and hardware synthesis tools are natural outcomes of such languages. It is nontrivial to extract instruction set information from these languages for use in a compiler. Some ADLs such as nML, LISA and EXPRESSION were initially developed to be high-level processor modeling languages. The important design considerations of these languages are general coverage over an architecture range and support for both compilers and simulators. Due to the divergent needs of compilers and simulators, a language usually can provide good description support for only one of them at a time.

Furthermore, the designers of the ADLs are often interested in the research value of the software tools. They are interested in different parts of the compilers or the simulators. As a result, the ADLs are usually biased toward the parts of interest. It is hard for a single ADL to satisfy the needs of all researchers. The remaining ADLs such as Maril, HMDES, TDL and PRMDL were developed as configuration systems for their accompanying software tools. In a sense these languages can be viewed as by-products of the software tools that were designed as compiler research infrastructure or as architecture exploration tools. The goal of these languages is flexibility within the configuration space. Generality is a secondary concern though it is desirable for extensibility considerations. However, if the generality of an ADL significantly exceeds that of its accompanying software tools, it may become a source of ambiguity. Moreover, generality often means less efficiency in modeling. There is little value in describing features that the software tools cannot support after all.

Thus, we see that ADLs can differ along many dimensions. Different ADLs are designed for different purposes, at different abstraction levels, with emphasis on different architectures of interest and with different views on the software tools. They reflect the designer's view of computer architectures and software tools.

The design of an ADL usually accompanies the development of software tools. The overall design effort on an ADL is often only a small fraction of the development effort on the software tools utilizing the ADL. So very often the developers of new software tools tend to design new ADLs. Though many ADLs have been introduced in the past decade, as mentioned earlier, this is not yet a mature research field. However, this is likely to change in the near future. Because programmable parts are playing an increasingly important role in the development of systems, this field is attracting the increased attention of researchers and engineers.

Table 14.1 summarizes and compares the important features of various ADLs. A few entries in the table were left empty, because information either is not available (no related publications) or is not applicable. The "( )" symbol in some entries indicates support with significant limitation.

TABLE 14.1 Comparison between ADLs

	MIMOLA	UDL/I	nML	ISDL	SLED/ λ-RTL	Maril	HMDES	TDL	EXPRESSION	LISA	RADL	PRMDL
category	HDL	HDL	behavioral	behavioral	behavioral	mixed	mixed	mixed	mixed	mixed	mixed	mixed
compiler	MSSQ, Record	COACH	CBC, CHESS	Aviv	Zephyr	Marion	IMPACT	PROPAN	EXPRESS			
simulator	MSSB/U	COACH	Sign/Sim, Checkers	GENSIM					SIMPRESS	LISA		
behavioral representation	RT-level	RT-level	RT-lists	RT-lists	RT-lists	RT-lists		RT-lists	RT-lists, mapping	RT-lists	RT-lists	mapping
hierarchical behavioral representation			yes	yes	yes	no	yes	yes	no	yes	yes	no
structural representation	netlist	netlist				resource	resource	resource	netlist	pipeline	pipeline	netlist
ILP compiler support			yes	yes		yes	yes	yes	yes	yes		yes
cycle simulation support	yes	yes	(yes)	(yes)	no	no	(yes)	no	yes	yes	yes	yes
control path	yes	yes	no	nos	no	no	no	no	no	yes	yes	no
constraint model				Boolean		resource	resource	resource, Boolean	resource			resource
other features							preprocessing support	preprocessing support	memory hierarchy	interrupt		

14.3.2 Basic Elements of an Architecture Description Language

Modern retargetable compiler back ends normally consist of three phases: code selection, register allocation and operation scheduling. The phases and various optimization steps [41] may sometimes be performed in different order and in different combinations. The first two phases are always necessary to generate working code. They require operation semantics and addressing mode information. Operation scheduling to minimize potential pipeline hazards in code execution, including data, structural and control hazards [26]. Scheduling also helps to pack operations into instruction-words for VLIW processors and tries to minimize unused empty slots. It is traditionally viewed as an optimization phase. As deeper pipelines and wider instruction-words are gaining more popularity in processor designs, scheduling is increasingly important to the quality of a compiler. A good ILP scheduler can improve performance and reduce code size greatly. Thus, we assume that the ILP scheduler is an essential part of the compiler back end in this chapter.

The data model of a common scheduler contains two basic elements: operand latency and reservation table. The first element models data and control hazards and is used to build a directed acyclic dependency graph. The second element models structural hazards between operations. The instruction-word packing problem does not always fit into the data model directly. A solution is to convert the packing conflicts to artificial resources and let conflicting operations require the use of such resources at the same time. By augmenting the reservation table with artificial resource usages, we prevent the scheduler from scheduling conflicting operations simultaneously. Thus, we can incorporate the packing problem into normal reservation table based scheduling [27].

Each machine operation performs some type of state transition in the processor. A precise description of the state transition has to include three elements: what, where and when. Correspondingly, information required by a compiler back end contains three basic elements: behavior, resource and time. In this context behavior means semantic actions including reading of source operands, calculation and writing of destination operands. Resource refers to abstract hardware resources used to model structural hazards or artificial resources used to model instruction packing. Common hardware resources include pipeline stages, register file ports, memory ports and data transfer paths. The last element, namely, time, is the cycle number when the behavior occurs and when resources are used. It is usually described relative to the operation fetch time or issue time. In some cases, phase number can be used for even more accurate modeling.

With the three basic elements, we can easily represent each machine operation in a list of triples. Each triple is in the form of (behavior, resource, time). For example, an integer Add operation in the pipeline depicted by Figure 14.4 can be described as:

```
(read operand reg[src1], through register file port a,
   at the 2nd cycle since fetch);
(read operand reg[src2], through register file port b,
   at the 2nd cycle since fetch);
(perform addition, in alu,
   at the 3rd cycle since fetch);
(write operand reg[dst], through register
   file write port, at the 5th cycle since fetch).
```

From the triple list, the compiler can extract the operation semantics and addressing mode by combining the first elements in order. It can also extract operand latency and reservation table information for the scheduler. Operand latency is used to schedule against data hazards, whereas reservation table can be used to schedule against structural hazards.

The triple list is a simple and general way of operation description. In practice, some triples may sometimes be simplified into tuples. For instance, most architectures contain sufficient register read ports so that no resource hazard can ever occur due to them. As a result, the scheduler does not need to take the ports into consideration. One can simplify the first two triples by omitting the resource elements. On the other hand, when resources exist that may cause contention and when no visible behavior is associated with the resource, the behavior can be omitted. In the same integer *Add* example, if for some reason the *MEM* stage becomes a source of contention, one may want to model it as a tuple of (*MEM* stage, at the 4th cycle since fetch).

The triple/tuple list can be found in all mixed languages in some form. For instance, the HMDES language is composed of these two types of tuples: the (behavior, time) tuple for operand latency and the (resource, time) tuple for reservation table. In LISA, operations are described in terms of pipeline stages. Each pipeline stage has an associated time according to its position in the path. Thus, a LISA description contains the triple list in some form too.

The final pieces of information left out from the triple list that are needed by the compilers are assembly syntax and binary encoding. These two can be viewed as attached attributes of the operation behavior. It is relatively straightforward to describe them.

14.3.3 Structuring of an Architecture Description Language

It is possible to turn the aggregation of triple lists directly into an ADL. However, describing a processor based on such an ADL can be a daunting task if it is to be written by humans. A typical processing element can contain around 50 $\sim$ 100 operations. Each of the operations may have multiple issue alternatives, proportional to the issue width of the architecture. Each issue alternative corresponds to one triple list whose length approximates the pipeline depth. As a result, the total number of tuples is about the product of operation count, issue width and pipeline depth, which may be of the order of thousands. Moreover, as mentioned earlier, artificial resources can be used to model instruction-word packing or irregular interoperation constraints. A raw triple list representation requires the user to do the artificial resource formation prior to the description. This process can be laborious and error prone for humans.

Thus, the task of an ADL design is to find a simple, concise and intuitive organization to capture the required information. Conceptually, a complete ADL should contain three parts.

14.3.3.1 Behavioral Part

This part contains operation addressing modes, operation semantics, assembly mnemonics and binary encoding. It corresponds to the first element in the triple. Behavior information is most familiar to compiler writers and can be found directly in architecture reference manuals. As a result, in many ADLs, if not all, behavioral information constitutes an independent section by itself.

In common processor designs, operations in the same category usually share many common properties. For instance, usually all three-operand arithmetic operations and logic operations share the same addressing modes and binary encoding formats. They differ only in opcode and semantics. Exploitation of the commonalities among operations can make the description compact, as has been demonstrated by nML and ISDL among others. Both languages adopt hierarchical description schemes under which common behaviors are described at the root of the hierarchy whereas specifics are kept in the leaves.

Beside commonality sharing, another powerful capability of hierarchical description is factorization of suboperations. A single machine operation can be viewed as the aggregation of several suboperations each of which has a few alternatives. Take, for example, the Load operation of TI TMS320C6200 family DSP [26]. The operation contains two suboperations: the load action and the addressing mode. Five versions of load action exist: load byte, load unsigned byte, load half word, load unsigned half word and load word. The addressing mode can further be decomposed into two parts: the offset mode and the address calculation mode. Two offset modes exist: register offset and constant offset. For the address calculation mode, six alternatives are possible: plus offset, minus offset, preincrement, predecrement, postincrement and postdecrement. Overall, the single Load operation contains 5*2*6 = 60 versions. Under a flat description scheme, it could be a nightmare when one finds a typographical error in the initial version after cutting, pasting and modifying 59 times.

In contrast to the geometric complexity of a flat description, a hierarchical description scheme factorizes suboperations and keeps the overall complexity linear. The resulting compact description is much easier to verify and modify. Conciseness is a very desirable feature for ADLs.

14.3.3.2 Structural Part

This part describes the hardware resources. It corresponds to the second element in the triple representation. Artificial resources may also be included in this part. The level of abstraction in this part can vary from fine-grained RT-level to coarse-grained pipeline stage level, though for use in a compiler, a coarse-grained description is probably more suitable. Two schemes exist for coarse-grained structural descriptions: resource based and netlist based.

Maril, TDL and HMDES utilize the resource-based description scheme. The advantage of the resource-based scheme is flexibility in creating resources. When creating a resource, the description writer does not have to worry about the corresponding physical entity and its connection with other entities. The resource may comfortably be an isolated artificial resource used for constraint modeling.

In contrast, EXPRESSION and PRMDL utilize the netlist-based scheme. A significant advantage of a netlist-based description is its intuitiveness. A netlist is more familiar to an architect than a list of unrelated abstract resource names. A netlist description scheme also enables friendly GUI input. A reservation table can be extracted from the netlist through some simple automated translation. The disadvantage is that the modeling capability may not be as flexible as the resource-based scheme. Architectures with complex dynamic pipeline behaviors are hard to model as a simple coarse-grained netlists. Also, netlists of commercial superscalar architectures may not be available to researchers.

Based on this comparison, the resource-based scheme seems more suitable as an approximation to complex high-end architecture descriptions, whereas the netlist-based scheme is better suited as an accurate model for simple ASIP/DSP designs for which netlists are available and control logic is minimal.

14.3.3.3 Linkage Part

This part completes the triple. The linkage information maps operation behavior to resources and time. It is usually not an explicit part of ADLs.

Linkage information is represented by different ADLs in different ways. No one way has an obvious advantage over another. In Maril, TDL or HMDES, linkage is described in the form of an explicit reservation table. For each operation, the resources it uses and the time of each use are enumerated. HMDES further exploits the commonality of resource uses through a hierarchical representation. In EXPRESSION, the linkage information is expressed in multiple ways: operations are mapped to pipeline stages by the *OPCODES* attribute associated with the pipeline stages, whereas data transfer resources are mapped to operations according to the operands and netlist connections. Grouping of operations into groups helps to simplify the mapping in EXPRESSION.

In summary, the desirable features of an architecture description include simplicity, conciseness and generality. These features may conflict with one another. A major task of the ADL design process is to find a good trade-off among the three for the architecture family in consideration. A good ADL should not be tied to internal decisions made by the software tools and it should minimize redundancy.

14.3.4 Difficulties

The triple list model looks simple and general. However, real-world architectures are never as simple and as straightforward to describe. ADL designers are constantly plagued with the trade-off of generality and abstraction level: low abstraction level brings high level of generality, whereas high abstraction level provides better efficiency. As for the software tools using the ADL, compilers prefer a high level of abstraction whereas cycle accurate simulators expect concrete detailed microarchitecture models. It is hard to find a clean and elegant representation satisfying both. Also, the idiosyncrasies of various architectures make general ADL design an even more agonizing process. When modeling a new processor, designers have to evaluate the ability of the ADL to accommodate the processor without disrupting the existing architecture model supported by the ADL. If this is not possible, then some new language constructs may need to be added. If the new construct is not orthogonal to the existing ones, the entire language may have to be revised. This also means that all written architecture descriptions and the parser need to be revised. The emergence of new "weird" architectural features keeps the designers busy struggling with evaluations, decisions and rewriting. Comparatively speaking, tool-specific ADL designers are much better off because they can comfortably exclude those architectures if the tools cannot handle them.

A few common problems faced by ADL designers and users are discussed next. Most of them do not have a clean solution and trade-offs have to be made depending on the specific needs in each case.

14.3.4.1 Ambiguity

The most common problem of an ADL is ambiguity, or inability to define precise behavior. As a side effect of abstraction, ambiguity exists in most ADLs, especially in modeling control related behavior. Take the common control behavior of interlocking, for example. Superscalar architectures have the capability to detect data and control hazards and stall the pipeline when a hazard exists. VLIW architectures, on the other hand, may not have the interlocking mechanism. The difference between the two obviously imposes different requirements on the simulator. It also results in different requirements for the operation scheduler: for superscalar, the scheduler only needs to remove as many read-after-write (RAW) data hazards as possible; whereas for VLIW, the scheduler should ensure the correctness of the code by removing all hazards. For two architectures differing in interlocking policy only, it is expected that the two would result in different structural descriptions. However, for many ADLs, actually no difference exists between the two due to the lack of control path specification. Among the mixed languages, only LISA and RADL can model the interlocking mechanism accurately because they put emphasis on accurate simulator generation.

Generally speaking, dynamic pipeline behaviors such as out-of-order issue and speculation are very hard to model cleanly. The use of microinstructions [26] in CISC machines makes modeling even harder. Such complicated behaviors can be ignored for compiler-oriented ADLs. However, an ADL with support for both compiler and cycle accurate simulator may need to address them.

Another common example of ambiguity is VLIW instruction-word packing. Some architectures allow the packing of only one issue bundle into an instruction-word (i.e., only operations scheduled to issue at the same cycle can appear in the same instruction-word) whereas other architectures allow multiple issue bundles in one instruction-word. The architecture dispatches the issue bundles at different cycles according to stop bits encoded in the instruction-word [28] or according to instruction-word templates [29]. Among the mixed ADLs, few can capture such instruction bundling behavior because it is related to the control path. The problem of code compression [47, 49] is more difficult than simple bundling and is not addressed by the ADLs at all.

Ambiguity is the result of abstraction. RT-level languages have the lowest level of ambiguity whereas highly abstract behavioral level ADLs have the highest level of ambiguity. Among mixed ADLs, simulator-oriented ADLs are less ambiguous than compiler-oriented ADLs. A good architecture description language design involves clever abstraction with minimal ambiguity.

In practice, ambiguity can be resolved by using an architecture template (i.e., the compiler or simulator presumes some architecture information left out from the ADL descriptions). This strategy

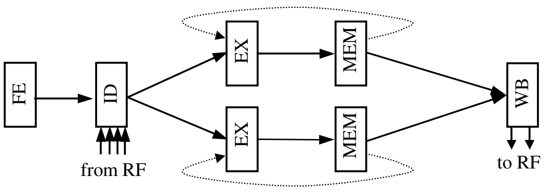


FIGURE 14.5 A two-issue integer pipeline with forwarding.

has been adopted by the tool-specific ADLs. The general-purpose ADLs can resolve the ambiguity while preserving generality by using multiple architecture templates.

14.3.4.2 Variable Latency

In many processors, operations or their operands may have variable latency. Many compiler-oriented ADLs can describe the nominal latency but not accurate variable latency. Consider again the example of an integer *Add* in Figure 14.4. By default, the source operands can be read at the *ID* stage and destination operand written at the *WB* stage. Thus, the source operand has a latency of 1, whereas the destination operand latency is 4, relative to the fetch time. However, if a forwarding path exists, which allows the operation to bypass its results from *MEM* to *EX*, then its destination operand latency is equivalent to 3. This is still fine if we fool the compiler by telling it the equivalent latency, though to the cycle accurate simulator we should tell the truth.

Now consider a multi-issue version of the same architecture in which interpipeline forwarding is forbidden, as is shown in Figure 14.5. Inside each pipeline, forwarding can occur and we have an equivalent destination operand latency of 3, whereas between pipelines, no forwarding is allowed and the destination latency is 4. Here we see a variable latency for the same operand. The variable latency is hard to describe explicitly in the triple list, unless the forwarding path and its implication become part of the ADL. Reservation table based ADLs normally do not capture forwarding paths. A common practice is to inform the scheduler about only the worst-case latency, which means no distinction exists for the compiler between the intra- and interpipeline case. As a result, some optimization opportunity is lost if the compiler has some control over operation issue.

Another type of variable latency resides in the operations themselves. Operations such as floating point division or square root can have a variable number of execution cycles depending on the exact source operands. In the triple list model, the variable operation latency means that the time element should be a variable, which could complicate the ADL greatly. To avoid the complication, usually a pessimistic latency is used in the description.

Memory operations can also result in variable latency in the presence of memory hierarchy. Most ADLs ignore the memory hierarchy by specifying a nominal latency for load and store operations because the emphasis of the ADLs is only on the processing elements. Memory hierarchy modeling is nontrivial due to the existence of various modules including ROM, SRAM, DRAM, nonvolatile memory and numerous memory hierarchy options and memory management policies. As the speed discrepancy between digital logic and memory keeps increasing, it will be more and more important for the compiler to be aware of the memory hierarchy. Thus far, only EXPRESSION and TDL provide a parameterized memory hierarchy model. Research has indicated that memory aware optimization in some cases yields an average profit of 24% reduction of execution time [30].

Overall, it is useful to specify structural information such as the pipeline diagram, forwarding path and memory hierarchy for use in the compiler.

14.3.4.3 Irregular Constraints

Constraints exist in computer architectures due to the limitation of resources including instruction-word width resources and structural resources. Common constraints include the range of constant operands and the number of issue slots. These constraints are familiar to compiler writers and can be handled with standard code generation and operation scheduling techniques.

ASIP designs often have extra-irregular constraints, because they are extra cost sensitive. Clustered and special register files with incomplete data transfer paths are commonly used to conserve chip area and power. Irregular instruction-word encoding is also used as a means to save instruction-word size, and therefore instruction memory size.

Typical operation constraints include both intra-operation and inter-operation constraints. An example of intra-operation constraint from a real proprietary DSP design is that in operation ADD D1, S0, S1, if D1 is AX, then S0 and S1 must be chosen from B0 and B1. If D1 is otherwise, no constraint is imposed on S0 and S1. Interoperation constraints occur similarly for operands in different operations. These operand-related constraints are hard to convert to artificial resource-based constraints. Special compiler techniques are needed to handle them, such as an extra constraint-checking phase in Aviv [19] or integer linear programming-based code optimization techniques [33].

The irregular constraints create new challenges for compilers as well as ADLs. Most existing ADLs cannot model irregular constraints. ISDL and TDL utilize Boolean expressions for constraint modeling, an effective model to supplement the triple list.

14.3.4.4 Operation Semantics

Both the code generator and the instruction set simulator need operation semantics information. For use in the code generator, simple treelike semantics would be most desirable due to the popularity of tree-pattern based dynamic programming algorithms [1]. Operation side effects such as the setting of machine flags are usually not part of the tree pattern and may be omitted. Similarly, operations unsuitable for code generation can be omitted too. In contrast, for the simulator, precise operation semantics of all operations should be provided. It is not easy to combine the two semantic specifications into one, while still keeping the result convenient for use in the compiler and simulator. ADLs such as LISA and EXPRESSION separate the two semantic specifications but result in redundancy. This leads to the additional problem of ensuring consistency between the redundant parts of the description.

For use in compilers, two schemes of operation semantics specification exist in ADLs. The first scheme is a simple mapping from machine-independent generic operations to machine-dependent operations. Both EXPRESSION and PRMDL use this scheme. The mapping scheme makes code generation a simple table lookup. This is a practical approach. The disadvantage is that the description cannot be reused for simulation purposes, and the description becomes dependent on compiler implementation.

The second description scheme defines for each operation one or more statements based on a predefined set of primitive operators. nML, ISDL and Maril all use this approach. The statements can be used for simulation, and for most operations, the compiler can derive tree pattern semantics for its use. However, difficulties exist because some of the compiler's intermediate representations (IRs) may fail to match any single operation.

For example, in lcc [31] IRs, comparison and branch are unified in the same node. An operator EQ compares two of its children and jumps to the associated label if they are equal. Many processors do not have a single operation performing this task. Take the x86 family, for example [32]. A comparison followed by a conditional branch operation is used to perform this task. The comparison operation sets machine flags and the branch operation reads the flags and alters control flow conditionally. The x86 floating point comparison and branch is even more complicated: the flags set by floating point comparison have to be moved through an integer register to the machine flags that the branch operation can read. As a result, a total of four operations need to be used to map the single EQ node. It would be a nontrivial job for the compiler to understand the semantics of each of the operations and the semantics associated with the flags to derive the matching. Some hints to the compiler should be provided for such cases. For these purposes, Maril [7] contains the “glue” and “*func” mechanisms. Glue allows the user to transform the IR tree for easier pattern matching and *func maps one IR node to multiple operations. However, such mechanisms expose the internals of the software tools to the ADL and make the ADL tool dependent.

14.3.4.5 Debugging ADL Descriptions

Debugging support is an important feature for any programming language because code written by humans invariably contains bugs. Retargetable compilers and simulators are difficult to write and debug themselves. The bugs inside an ADL description make the development process harder.

An initial ADL description often involves thousands of lines of code and may end up with hundreds of errors. Among the errors, syntax errors can be easily detected by ADL parsers. Some other forms of errors such as redefinition of the same operation can be detected by careful validity checking in the compiler. However, the trickiest errors can pass both tests and remain in the software for a long time. The experience of LISA developers shows that it takes longer to debug a machine description than to write the description itself [10].

Thus far, no formal methodology exists for ADL debugging. An ADL description itself does not execute on the host machine. It is usually interpreted or transformed into executable C code. In the interpretation case, debugging of the ADL description is actually the debugging of the retargetable software tool using the description. In the executable transformation case, debugging might be performed on the generated C, which probably looks familiar only to the tool writer.

The task of debugging would be easier if a usable target-specific compiler and a simulator for the architecture to be described exist. Comparison of emitted assembly code or simulation traces helps detect errors. Unfortunately, this is impossible for descriptions of new architecture designs. The ADL writer probably has to iterate in a trial-and-error process for weeks before getting a working description.

Though it takes a long time to get the right description, it is still worthwhile because it would take even longer to customize a compiler. After the initial description is done, mutation can then be performed on it for the purpose of design space exploration (DSE).

14.3.4.6 Others

Other issues in ADL design include handling of register overlap and mapping of intrinsics. These issues are not too hard and have been addressed by many ADL designs. They are worth the attention of all new ADL designers.

Register overlap, or alias, is common to many architectures. For example, in x86 architecture, the lower 16-b of the 32-b register EAX can be used independently as AX. The lower and upper half of AX can also be used as 8-b registers AH and AL. Register overlap also commonly exists in floating point register files of some architectures [45] where a single precision register is half of a double precision register.

Most architectures contain a few operations that cannot be automatically generated by compilers or that cannot be expressed by predefined RT- lists operators. For example, the TI TMS320C6200 family implements bit manipulation operations such as reverse operation BITR. Such operations are useful to special application families. However, no C operator can be mapped to such operations. Intrinsics mapping is a cleaner way to utilize these operations than in-line assembly. Thus, it is helpful for ADL to provide intrinsics support.

Description capability of operating system related operations may also be of interest to ADLs with accurate simulation support. Examples of such operations include the ARM branch and exchange operations that switches the processor between normal 32-b instruction width mode and the 16-b compressed instruction width mode [52]. Another example is the TI TMS320C6200 family MVC operation that updates the addressing mode or control status of the processor [28].

14.3.5 Future Directions

Driven by market demands, increasingly complex applications are implemented on semiconductor systems today. As both chip density and size increases, the traditional board-level functionality can now be performed on a single chip. As a result, processing elements are no longer privileged central components in electronic systems. A modern system-on-chip (SOC) may contain multiple processors, buses, memories and peripheral circuits. As electronic design automation (EDA) tools start to address SOC design issues, system-level modeling becomes a hot research area.

A processing element model now becomes part of the system model. It is desirable that the same ADL modeling the processing element can be extended to model the whole system. Hardware description language-based ADLs such as MIMOLA may be naturally used for this purpose. However, due to the large size of a typical SOC, description of the entire system at the RT-level inevitably suffers from poor efficiency. When analog or mixed-signal parts are involved in the system, it is impossible to model the entire system even at the RTL level. Abstraction at different levels for different parts of the system is a more practical approach.

To reuse the existing tools and ADLs for system-level description, an effective approach is to extend the ADLs with communication interface models. Communication interface models include the modeling of bus drivers, memory system interfaces, interrupt interfaces, etc.

The system-level interface is important for processing element simulators to interact with system simulators. It is also important for the design of advanced compilers. For instance, a system-level compiler can partition tasks and assign the tasks to multiple processing elements according to the communication latency and cost. Compilers can also schedule individual transactions to avoid conflicts or congestion according to the communication model.

Among the existing mixed ADLs, LISA is capable of modeling interrupts and EXPRESSION has a parameterized memory hierarchy model. Both of these can be viewed as important steps toward the specification of a system-level communication interface.

On the whole, systematic modeling of the communication interface for processing elements is an interesting though difficult research direction. It helps form the basis of a system-level retargetable compiler and simulator, which are desirable tools for modern SOC design.

14.4 Conclusion

ADLs provide machine models for retargetable software tools including the compiler and the simulator. They are crucial to DSE of ASIP designs.

In terms of requirements for an optimizing compiler, a tool-independent ADL should contain several pieces of information:

- Behavioral information in the form of RT lists. Hierarchical behavioral modeling based on attribute grammars is a common and effective means. For VLIW architectures, instruction-word formation should also be modeled.
- Structural information in the form of reservation table or coarse-grained netlists. Essential information provided by this section includes abstract resources such as pipeline stages and data transfer paths.
- Mapping between the behavioral and structural information. Information in this aspect includes the time when semantic actions take place and the resources used by the action.

In addition, irregular ILP constraint modeling is useful for ASIP-oriented ADLs. The desirable features of an ADL include simplicity for the ease of understanding, conciseness for description efficiency, generality for wide architecture range support, minimal ambiguity and redundancy and tool independence. Trade-off of abstraction levels needs to be made if a single ADL is used for both a retargetable compiler and a cycle-accurate simulator for a range of architectures. As ASIP design gains popularity in the SOC era, ADLs as well as retargetable software tool sets will become an important part of EDA tools. They will encompass not only single processors but also system interfaces.

References

[1] A.V. Aho, M. Ganapathi and S.W.K. Tjiang, Code generation using tree matching and dynamic programming, *ACM Trans. Programming Languages Syst.*, 11(4), 491–516, October 1989.

[2] B. Kienhuis, Design Space Exploration of Stream-Based Dataflow Architectures, Doctoral thesis, Delft University of Technology, Netherlands, January 1999.

[3] R. Leupers and P. Marwedel, Retargetable code generation based on structural processor descriptions, *Design Automation Embedded Syst.*, 3(1), 1–36, January 1998.

[4] H. Akaboshi, A Study on Design Support for Computer Architecture Design, Doctoral thesis, Department of Information Systems, Kyushu University, Japan, January 1996.

[5] A. Fauth, J. Van Praet and M. Freericks, Describing Instructions Set Processors Using nML, in Proceedings European Design and Test Conference, Paris, France, March 1995, pp. 503–507.

[6] G. Hadjiyiannis, S. Hanono and S. Devadas, ISDL: An Instruction Set Description Language for Retargetability, in Proceedings 34th Design Automation Conference (DAC 97), Anaheim, CA, June 1997, pp. 299–302.

[7] D.G. Bradlee, R.R. Henry and S.J. Eggers, The Marion System for Retargetable Instruction Scheduling, in Proceedings ACM SIGPLAN '91 Conf. on Programming Language Design and Implementation, Toronto, Canada, June 1991, pp. 229–240.

[8] J.C. Gyllenhaal, B.R. Rau and W.W. Hwu, Hmdes Version 2.0 Specification, Technical report IMPACT-96-3, IMPACT Research Group, University of Illinois, Urbana, IL, 1996.

[9] S. Pees et al., LISA — Machine Description Language for Cycle-Accurate Models of Programmable DSP Architectures, in Proceedings 36th Design Automation Conference (DAC 99), New Orleans, LA, June 1999, pp. 933–938.

[10] S. Pees, A. Hoffmann and H. Meyr, Retargetable compiled simulation of embedded processors using a machine description language, *ACM Trans. Design Automation Electron. Syst.*, 5(4), 815–834, October 2000.

[11] C. Siska, A Processor Description Language Supporting Retargetable Multi-Pipeline DSP program Development Tools, in Proceedings 11th International Symposium on System Synthesis (ISSS 98), Taiwan, China, December 1998, pp. 31–36.

[12] A. Halambi et al., Expression: A Language for Architecture Exploration through Compiler/Simulator Retargetability, in Proceedings Design Automation and Test in Europe (DATE 99), March 1999, Munich, Germany, pp. 485–490.

[13] A.S. Terechko, E.J.D. Pol and J.T.J. van Eijndhoven, PRMDL: A Machine Description Language for Clustered VLIW Architectures, in Proceedings European Design and Test Conference (DATE 01), Munich, Germany, March 2001, p. 821.

[14] R. Leupers and P. Marwedel, Retargetable Generation of Code Selectors from HDL Processor Models, in Proceedings European Design & Test Conference (ED&TC 97), Paris, France, 1997, pp. 144–140.

[15] IEEE Standard Description Language Based on the Verilog Hardware Description Language (IEEE std. 1364-1995), IEEE, Piscataway, NJ, 1995.

[16] A. Fauth and A. Knoll, Automatic generation of DSP program development tools, in Proceedings International Conference Acoustics, Speech and Signal Processing (ICASSP 93), Minneapolis, MN, April 1993, Vol. 1, pp. 457–460.

[17] F. Lohr, A. Fauth and M. Freericks, SIGH/SIM: An Environment for Retargetable Instruction Set Simulation, Technical report 1993/43, Department Computer Science, Tech. Univ. Berlin, Germany, 1993.

[18] D. Lanneer et al., Chess: Retargetable code generation for embedded DSP processors, *Code Generation for Embedded Processors*, Kluwer Academic, Boston, MA, 1995, pp. 85–102.

[19] S.Z. Hanono, Aviv: A Retargetable Code Generator for Embedded Processors, Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA, June 1999.

[20] G. Hadjiyiannis, P. Russo and S. Devadas, A Methodology for Accurate Performance Evaluation in Architecture Exploration, in Proceedings 36th Design Automation Conference (DAC 99), New Orleans, LA, June 1999, pp. 927–932.

[21] S.C. Johnson, Yacc: Yet Another Compiler-Compiler, *dinosaur.compilertools.net/yacc/index.html* (current November 2001).

[22] M. Bailey and J. Davidson, A Formal Model and Specification Language for Procedure Calling Conventions, in Conference Record of 22nd SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 95), New York, NY, January 1995, pp. 298–310.

[23] S. Aditya, V. Kathail and B. Rau, Elcor's Machine Description System: Version 3.0, Technical report, HPL-98-128, Hewlett-Packard Company, September 1999.

[24] A. Halambi et al., A Customizable Compiler Framework for Embedded Systems, in 5th International Workshop Software and Compilers for Embedded Systems (SCOPES 2001), St. Goar, Germany, 2001.

[25] A. Khare et al., V_SAT: A Visual Specification and Analysis Tool for System-on-Chip Exploration, Proceedings EUROMICRO-99, Workshop on Digital System Design (DSD99), Milan, Italy, 1999.

[26] J. Hennessy and D. Patterson, *Computer Architecture A Quantitative Approach*, 2nd ed., Morgan Kaufmann, San Francisco, 1995.

[27] S. Rajagopalan, M. Vachharajani and S. Malik, Handling Irregular ILP within Conventional VLIW Schedulers Using Artificial Resource Constraints, Proceedings International Conference Compilers, Architectures, Synthesis for Embedded Systems (CASES 2000), San Jose, CA, November 2000, pp. 157–164.

[28] TMS320C6000 CPU and Instruction Set Reference Guide, Texas Instrument Inc., October 2000.

[29] Intel IA-64 Software Developer's Manual, Vol. 3: Instruction Set Reference, Intel Corporation, July 2000.

[30] P. Grun, N. Dutt and A. Nicolau, Memory Aware Compilation through Accurate Timing Extraction, in Proceedings 37th Design Automation Conference, Los Angeles, CA, June 2000, pp. 316–321.

[31] C. Fraser and D. Hanson, *A Retargetable C Compiler: Design and Implementation*, Addison-Wesley, Menlo Park, CA, 1995.

[32] Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference Manual, Intel Corporation, 1997.

[33] D. Kästner, Retargetable Postpass Optimization by Integer Linear Programming, Ph.D. thesis, Saarland University, Germany, 2000.

[34] N. Ramsey and J. Davidson, Machine Descriptions to Build Tools for Embedded Systems, *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES '98), Lecture Notes in Computer Science*, Vol. 1474, Springer-Verlag, June 1998, pp. 172–188.

[35] N. Ramsey and M. Fernandez, Specifiying representation of machine instructions, *ACM Trans. Programming Languages Syst.*, 19(3), 492–524, May 1997.

[36] N. Ramsey and J. Davidson, Specifying Instructions' Semantics Using λ-RT, interim report, University of Virginia, Wise, VA, July, 1999.

[37] R. Stallman, Using and Porting the GNU Compiler Collection (GCC), *gcc.gnu.org/onlinedocs/gcc_toc.html* (current December 2001).

[38] *www.retarget.com* (current December 2001).

[39] J. Paakki, Attribute grammar paradigms — a high-level methodology in language implementation, *ACM Comput*. Surv., 27(2), 196–256, June 1995.

[40] R. Milner, M. Tofte and R. Harper, *The Definition of Standard ML*, MIT Press, Cambridge, MA, 1989.

[41] S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann, San Francisco, 1997.

[42] M. Fernandez, Simple and Effective Link-Time Optimization of Modula-3 Programs, in Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation, June 1995, pp. 103–115.

[43] N. Ramsey, A Retargetable Debugger, Ph.D. thesis, Department of Computer Science, Princeton University, Princeton, NJ, 1992.

[44] MC88100 RISC Microprocessor User's Manual, Motorola Inc., 1988.

[45] G. Kane, *MIPS R2000 RISC Architecture*, Prentice Hall, Englewood Cliffs, NJ, 1987.

[46] i860 64-bit Microprocessor Programmer's Reference Manual, Intel Corporation, 1989.

[47] S. Aditya, S.A. Mahlke and B. Rau, Code size minimization and retargetable assembly for custom EPIC and VLIW instruction formats, *ACM Trans. Design Automation Electron. Syst*., 5(4), 752–773, October 2000.

[48] E. Rohou, F. Bodin and A. Seznec, SALTO: System for Assembly-Language Transformation and Optimization, in Proceedings 6th Workshop on Compilers for Parallel Computers, Aachen, Germany, December 1996, pp. 261–272.

[49] A. Wolfe and A. Chanin, Executing Compressed Programs on an Embedded RISC Architecture, in Proceedings 25th International Symposium on Microarchitecture, Portland, OR, December 1992, pp. 81–91.

[50] D. Kastner, TDL: A Hardware and Assembly Description Languages, Technical report TDL 1.4, Saarland University, Germany, 2000.

[51] The SPARC Architecture Manual, Version 8, SPARC International, 1992.

[52] Arm Architecture Reference Manual, Advanced RISC Machines Ltd., 1996.

15

Instruction Selection Using Tree Parsing

Priti Shankar
Indian Institute of Science

15.1 Introduction and Background

15.1.1 Introduction

One of the final phases in a typical compiler is the instruction selection phase. This traverses an intermediate representation of the source code and selects a sequence of target machine instructions that implement the code. There are two aspects to this task. The first one has to do with finding efficient algorithms for generating an optimal instruction sequence with reference to some measure of optimality. The second has to do with the automatic generation of instruction selection programs from precise specifications of machine instructions. Achieving the second aim is a first step toward retargetabiltiy of code generators. We confine our attention to instruction selection for basic blocks. An optimal sequence of instructions for a basic block is called *locally optimal code*.

Early techniques in code generation centered around interpretive approaches where code is produced for a virtual machine and then expanded into real machine instructions. The interpretive approach suffers from the drawback of having to change the code generator for each machine. The idea of code generation by tree parsing replaced the strategy of virtual machine interpretation. The intermediate representation (IR) of the source program is in the form of a tree and the target machine instructions are represented as productions of a regular tree grammar augmented with semantic actions and costs. The code generator parses the input subject tree and, on each reduction, outputs target code. This is illustrated in Figure 15.1 for a subject tree generated by the grammar of Example 15.1.

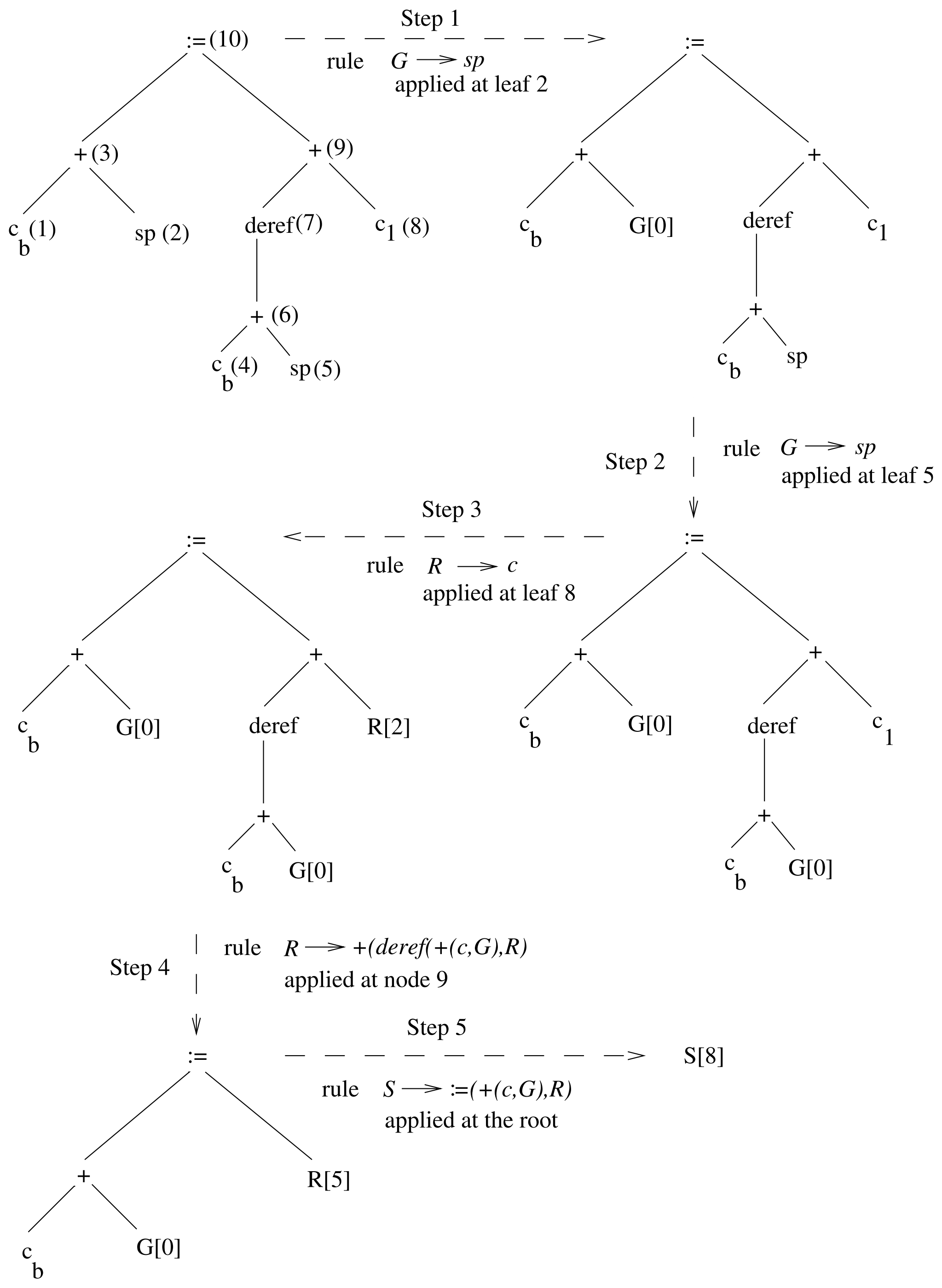

Step 1
rule $G \longrightarrow sp$ applied at leaf 2
:=(10)
+(3)
+(9)
c_b(1)
sp (2)
deref(7)
c_1 (8)
+ (6)
c_b(4)
sp(5)
Step 2
rule $G \longrightarrow sp$ applied at leaf 5
Step 3
rule $R \longrightarrow c$ applied at leaf 8
G[0]
R[2]
Step 4
rule $R \longrightarrow +(deref(+(c,G),R)$ applied at node 9
Step 5
rule $S \longrightarrow :=(+(c,G),R)$ applied at the root
R[5]
S[8]

FIGURE 15.1 A sequence of tree replacements for an IR tree for the grammar of Example 15.1.

Example 15.1

Consider the tree grammar that follows where the right-hand sides of productions represent trees using the usual list notation. Each production is associated with a cost and a semantic action enclosed in braces. The operator := is the assignment operator, *deref* is the deferencing operator that dereferences

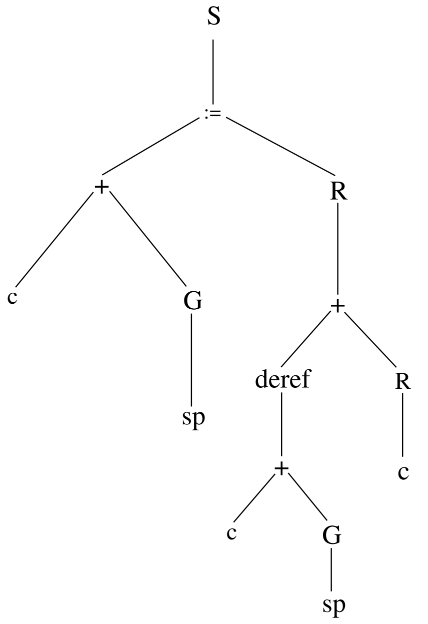


FIGURE 15.2 Derivation tree for the IR tree of Figure 15.1.

an address to refer to its contents. Subscripts on symbols are used to indicate attributes. For example, c_j indicates a constant with value j. Costs are assumed to be additive.

$S \rightarrow := (+(c_j, G_k), R_i)$	[3]	$\{emit(\text{“}store\ R\%s, \%s[\%s]\text{”}), i, j, k)\}$
$R \rightarrow +(deref(+(c_j, G_k), R_i)$	[3]	$\{emit(\text{“}add\ \%s[\%s], R\%s\text{”}), j, k, i\}$
$R_i \rightarrow c_j$	[2]	$\{i = allocreg();\ emit(\text{“}mov\ \#\%s, R\%s\text{”}), j, i\}$
$G_i \rightarrow sp$	[0]	$\{i = sp\}$

Consider the source level statement $b := b + 1$, where b is a local variable stored in the current frame pointed at by the stack pointer sp. The IR tree is shown as the first tree in Figure 15.1 with nodes numbered from 1 to 10. The IR tree is processed using the replacements shown in Figure 15.1. Each nonterminal replacing a subtree has its cost shown alongside.

The tree is said to have been *reduced* to S by a tree-parsing process, which implicitly constructs the derivation tree shown in Figure 15.2 , for the subject tree. The set of productions used is a cover for the tree. In general, there are several covers, given a set of productions, and we aim to obtain the best one according to some measure of optimality. The semantic actions also include a call to a routine to allocate a register. This can go hand in hand with the tree parsing and the selection of the register is independent of the parsing process. For the sequence of replacements shown, the code emitted is:

$$move\ \#1, R_0$$
$$add\ b[sp], R_0$$
$$store\ R_0, b[sp]$$

Fraser [19] and Cattell [12] employed tree-pattern matching along with heuristic search for code generation. Fraser used knowledge-based rules to direct pattern matching, whereas Cattell suggested a goal-directed heuristic search. Graham and Glanville [23] opened up new directions in the area of retargetable code generation. They showed that if the intermediate code tree is linearized and the target machine instructions are represented as context free grammar productions, then bottom-up parsing techniques could be used to generate parsers that parse the linearized intermediate code tree and emit machine instructions while performing reductions. This was a purely syntactic approach to the problem of instruction selection, and suffered from the drawback that the effective derivation has a left bias, in that the code for the subtree corresponding to the left operand is selected without considering the right operand. As a result, the code generated is suboptimal in many instances.

A second problem with the Graham–Glanville approach is that there are many architectural restrictions that have to be taken into account when generating code, such as register restrictions on addressing modes and so on. A purely syntactic approach to such semantics yields a very large number of productions in the specifications. Several implementations of the Graham–Glanville technique have been described, and the technique has been applied to practical compilers, [24, 30] Ganapathi and Fischer [22] suggested using attribute grammars instead of context-free grammars to handle the problem of semantics. Attributes are used to track multiple instruction results, for example, the setting of condition codes. Further, predicates are used to specify architectural restrictions on the programming model. Instruction selection is therefore performed by attributed parsing. Although this solves the problem of handling of semantic attributes, it is still not able to overcome the problem of left bias in the mode of instruction selection. A good survey of early work in this area is [21].

The seminal work of Hoffman and O'Donnell (HOD) [28] provided new approaches that could be adopted for retargetable code generation. They considered the general problem of pattern matching in trees with operators of fixed arity and presented algorithms for both top-down and bottom-up tree-pattern matching. The fact that choices for patterns that are available can be stored, and decisions can be deferred until an optimal choice can be made, overcomes the problem of left bias in the Graham–Glanville approach. The basic idea is that a tree automaton can be constructed from a specification of the machine instructions in terms of tree patterns during a preprocessing phase, and this can be used to traverse an intermediate code tree during a matching phase to find all matches, and finally generate object code. HOD showed that if tables encoding the automaton could be precomputed, then matching could be achieved in linear time. The size of the tables precomputed for bottom-up tree-pattern matching automata can be exponential in the product of the arity and the number of sub-patterns.

Chase [13] showed that table compression techniques that could be applied as the tables were under construction could greatly reduce auxiliary space requirements while performing pattern matching. This important observation actually made the HOD technique practically useful. Several bottom-up tools for generating retargetable code generators were designed. Some of these are hard-coded in that their control structure mirrors the underlying regular tree grammar [17, 20]. Such techniques have been employed in the tools **BEG** [17] and **iburg** [20], where the dynamic programming strategy first proposed by Aho and Johnson [4] is used in conjunction with tree parsing to obtain locally optimal code. Aho and Ganapathi [2] showed that top-down tree parsing combined with dynamic programming can be used for generating locally optimal code. Their technique is implemented in the code-generator generators **twig** [42] and **olive** [43].

The advantage of a top-down technique is that the tables describing the tree-parsing automaton are small. The disadvantage stems from the fact that cost computations associated with dynamic programming are performed at code generation time, thus slowing down the code generator. Dynamic programming using a top-down traversal was also used by Christopher, Hatcher and Kukuk [14]. Appel [7] has generated code generators for the VAX and Motorola 68020 using **twig**. Weisgerber and Wilhelm [44] describe top-down and bottom-up techniques to generate code. Henry and Damron [27] carried out an extensive comparison of Graham–Glanville style code generators and those based on tree parsing. Hatcher and Christopher [26] showed that costs could be included in the states of the finite state tree pattern matching automaton so that optimal instruction selection could be performed without incurring the extra overhead of the cost computations on-line. Static cost analysis exemplified in the approach of Hatcher and Christopher makes the code-generator generator more complex and involves large space overheads. However, the resultant code generator is simple and fast, which implies faster compilation. The technique of Hatcher and Christopher does not guarantee that the statically selected code can always be optimal and requires interaction from the user.

Pelegri-Llopart and Graham [37] combined static cost analysis with table compression techniques from Chase, and used term rewrite systems instead of tree patterns to develop a bottom-up rewrite system (BURS). A BURS is more powerful than a bottom-up pattern-matching system because it can incorporate algebraic properties of terms into the code generation system. However, systems based on BURS are generally more complex than those based on tree parsing. Balachandran, Dhamdhere and Biswas [9] used an extension of the work of Chase to perform static cost analysis and produce optimal code. Proebsting [39] used a simple and efficient algorithm for generating tables with static costs, in which new techniques called triangle trimming and chain rule trimming are used for state reduction. This technique is used in the bottom-up tool **burg**. Ferdinand, Seidl and Wilhelm [18] reformulated the static bottom-up tree-parsing algorithm based on finite tree automata. This generalized the work of Chase to work for regular tree grammars and included table compression techniques.

More recently, Nymeyer and Katoen [36] describe an implementation of an algorithm based on BURS theory, which computes all pattern matches, and which does a search that results in optimal code. Heuristics are used to cut down the search space. Shankar et al. [41] construct a linear regular (LR)-like parsing algorithm for regular tree parsing, which can be used for code generation with dynamic cost computation. The static cost computation technique of Balachandran et al. and the LR-like parsing approach of Shankar et al. have been combined into a technique for locally optimal code generation in [34]. A treatment of tree parsing for instruction selection is given in [45].

The bottom-up techniques mentioned earlier all require at least two passes over the intermediate code tree, one for labeling the tree with matched patterns and costs and the next for selecting the least-cost parse based on the information collected during the first pass. Thus, an explicit IR tree needs to be built. A technique that avoids the building of an explicit IR tree is proposed by Proebsting and Whaley [40]. The tool **wburg** generates parsers that can find an optimal parse in a single pass. An IR tree is not built explicitly, because the tree structure is mirrored in the sequence of procedure invocations necessary to build the tree in a bottom-up fashion. The class of grammars handled by this technique is a proper subset of the grammars that the two-pass system can handle. However, Proebsting and Whaley have claimed that optimal code can be generated for most major instruction sets including the SPARC, the MIPS R3000 and the x86.

We restrict our attention in this chapter to instruction selection techniques based on tree parsing. The techniques based on term-rewriting systems [15, 36, 37] are more powerful, but not as practical.

15.1.2 Dynamic Programming

Aho and Johnnson [4] used dynamic programming to generate code for expression trees. The algorithm presented by them generates optimal code for a machine with r interchangeable registers and instructions of the form $R_i := E$; R_i is one of the registers and E is any expression involving operators, registers and memory locations. The dynamic programming algorithm generates optimal code for evaluation of an expression contiguously. If T is an expression tree with op at its root and T_1 and T_2 as its subtrees, then a program is said to evaluate the tree contiguously if it first evaluates the subtrees of T that need to be computed into memory, and then evaluates the remainder of the tree either in the order T_1, T_2 and then the root, or T_2, T_1 and then the root. Aho and Johnson proved that for a uniform register machine, optimal code would always be generated by their algorithm. The implication of the property is that for any expression tree there is always an optimal program that consists of optimal programs for subtrees of the root followed by an instruction to evaluate the root. The original dynamic programming algorithm uses three phases. In the first bottom-up phase it computes a vector of costs for each node n of the expression tree, in which the ith component of the vector is the cost of computing the subtree at that node into a register, assuming i registers are available for the computation $0 \leq i \leq r$. The "zeroth" component of the vector is the minimal cost of computing the subtree into memory. In the second phase the algorithm traverses the tree top down to determine which subtrees should be computed into memory. In the third phase the algorithm traverses each tree using the cost vectors to generate the optimal code.

A simplified form of the dynamic programming algorithm is used in most code generator tools where what is computed at each node is a set of (rule, scalar cost) pairs. Register allocation is not part of the instruction selection algorithm though it can be carried out concurrently. The cost associated with a subtree is computed either at compile time (i.e., dynamically), by using cost rules provided in the grammar specification, or by simply adding the costs of the children to the cost of the operation at the root, or at compiler generation time (i.e., statically) by precomputing differential costs and storing them along with the instructions that match, as part of the state information of a tree pattern matching automaton. How exactly this is done will become clear in the following sections.

15.2 Regular Tree Grammars and Tree Parsing

Let A be a finite alphabet consisting of a set of operators *OP* and a set of terminals T. Each operator *op* in *OP* is associated with an *arity*, *arity(op)*. Elements of T have arity 0. The set *TREES*(A) consists of all trees with internal nodes labeled with elements of *OP*, and leaves with labels from T. Such trees are called *subject trees* in this chapter. The number of children of a node labeled *op* is *arity(op)*. Special symbols called *wild cards* are assumed to have arity of zero. If N is a set of wild cards, the set *TREES*$(A \cup N)$ is the set of all trees with wild cards also allowed as labels of leaves.

We begin with a few definitions drawn from [9] and[18]:

Definition 15.1. *A regular cost augmented tree grammar G is a four tuple (N, A, P, S) where:*

1. *N is a finite set of nonterminal symbols.*
2. $A = T \cup OP$ *is a ranked alphabet, with the ranking function denoted by arity. T is the set of terminal symbols and OP is the set of operators.*
3. *P is a finite set of production rules of the form* $X \rightarrow t\ [c]$ *where* $X \in N$ *and t is an encoding of a tree in TREES*$(A \cup N)$*, and c is a cost, which is a nonnegative integer.*
4. *S is the start symbol of the grammar.*

A tree pattern is thus represented by the right-hand side of a production of P in the preceding grammar. A production of P is called a *chain rule*, if it is of the form $A \rightarrow B$, where both A and B are nonterminals.

Definition 15.2. *A production is said to be in normal form if it is in one of the following three forms.*

1. $A \rightarrow op(B_1, B_2 \ldots, B_k)[c]$ *where* $A, B_i, i = 1, 2, \ldots, k$ *are all nonterminals, and op has arity k.*
2. $A \rightarrow B\ [c]$*, where A and B are nonterminals. Such a production is called a* chain rule.
3. $B \rightarrow b\ [c]$*, where b is a terminal.*

A grammar is in normal form if all its productions are in normal form.

Any regular tree grammar can be put into normal form by the introduction of extra nonterminals and zero-cost rules.

An example of a cost augmented regular tree grammar in normal form follows, with arities of symbols in the alphabet shown in parentheses next to the symbol:

Example 15.2

$$G = (\{V, B, G\}, \{a(2), b(0)\}, P, V)$$

P:

$$\begin{array}{lll} V \rightarrow & a(V, B) & [0] \\ V \rightarrow & a(G, V) & [1] \\ V \rightarrow & G & [1] \\ G \rightarrow & B & [1] \\ V \rightarrow & b & [7] \\ B \rightarrow & b & [4] \end{array}$$

DEFINITION 15.3. *For $t, t' \in$ TREES$(A \cup N)$, t directly derives t', written as $t \Rightarrow t'$ if t' can be obtained from t by replacement of a leaf of t labeled X by a tree p where $X \rightarrow p \in P$. We write $\Rightarrow_r$ if we wish to specify that rule r is used in a derivation step. The relations $\Rightarrow^+$ and $\Rightarrow^*$ are the transitive closure and reflexive–transitive closure respectively of $\Rightarrow$.*

An X-derivation tree, D_X, for G has the following properties:

- The root of the tree has label X.
- If X is an internal node, then the subtree rooted at X is one of the following three types (for describing trees we use the usual list notation):
  1. $X(D_Y)$ if $X \rightarrow Y$ is a chain rule and D_Y is a derivation tree rooted at Y.
  2. $X(a)$ if $X \rightarrow a, a \in T$ is a production of P.
  3. $X(op(D_{X_1}, D_{X_2}, \ldots, D_{X_k}))$ if $X \rightarrow op(X_1, X_2, \ldots, X_k)$ is an element of P.

The language defined by the grammar is the set:

$$L(G) = (t) \mid t \in \mathit{TREES}(A)$$

and

$$S \Longrightarrow^* t$$

With each derivation tree is associated a cost, namely, the sum of the costs of all the productions used in constructing the derivation tree. We label each nonterminal in the derivation tree with the cost of the subtree below it. Four cost augmented derivation trees for the subject tree $a(a(b, b), b)$ in the language generated by the regular tree grammar of preceding Example 15.2 are displayed in Figure 15.3.

DEFINITION 15.4. *A rule $r : X \rightarrow p$ matches a tree t if there exists a derivation $X \Rightarrow_r p \Rightarrow^* t$.*

DEFINITION 15.5. *A nonterminal X matches a tree t if there exists a rule of the form $X \rightarrow p$ that matches t.*

DEFINITION 15.6. *A rule or nonterminal matches a tree t at node n if the rule or nonterminal matches the subtree rooted at the node n.*

Each derivation tree for a subject tree thus defines a set of matching rules at each node in the subject tree (a set because there may be chain rules that also match at the node).

Example 15.3
For all the derivation trees of Figure 15.3 the rule $V \rightarrow a(V, B)$ matches at the root.

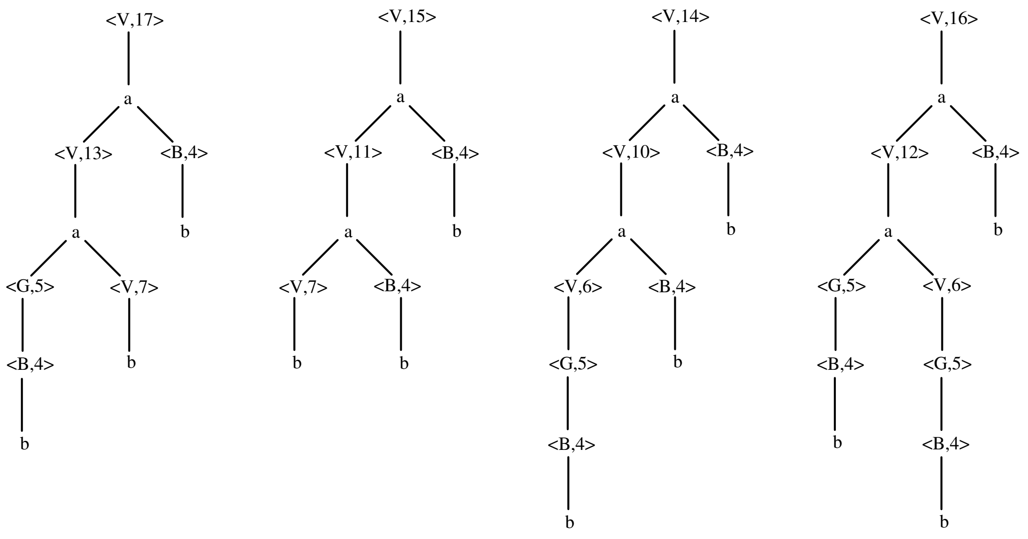


FIGURE 15.3 Four cost-augmented derivation trees for the subject tree $a(a(b, b), b)$ in the grammar of Example 15.2.

For a rule $r : X \rightarrow p$ matching a tree t at node n, where t_1 is the subtree rooted at node n, we define:

1. The cost of rule r matching t at node n is the minimum of the cost of all possible derivations of the form $X \Rightarrow_r p \Rightarrow^* t_1$.
2. The cost of nonterminal X matching t at node n is the minimum of the cost of all rules r of the form $X \rightarrow p$ that match t_1.

Typically, any algorithm that does dynamic cost computations compares the costs of all possible derivation trees and selects one with minimal costs while computing matches. To do this it has to compute for each nonterminal that matches at a node the minimal cost of reducing to that nonterminal (or equivalently, deriving the portion of the subject tree rooted at that node from the nonterminal.) In contrast to this are algorithms that perform static cost computations, precompute relative costs and store differential costs for nonterminals. Thus, the cost associated with a rule r at a particular node in a subject tree is the difference between the minimum cost of deriving the subtree of the subject tree rooted at that node using rule r at the first step, and the minimum cost of deriving it using any other rule at the first step. Figure 15.4 shows the matching rules with relative costs at the nodes of the subject tree for which derivation trees are displayed in Figure 15.3. By assuming such differences are bounded for all possible derivation trees of the grammar, they can be stored as part of the information in the states of a finite state tree-parsing automaton. Thus, no cost analysis need be done at matching time. Clearly, tables encoding the tree automaton with static costs tend to be larger than those without cost information in the states.

The tree-parsing problem we address in this chapter is:

Given a regular tree grammar $G = (N, T, P, S)$, and a subject tree t in *TREES(A)*, find (a representation of) all S-derivation trees for t.

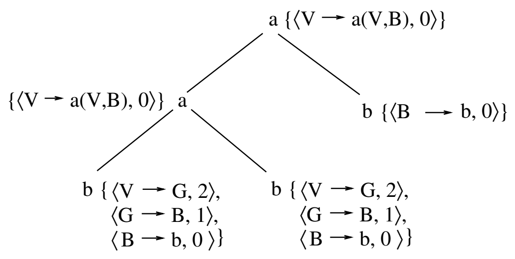

a {⟨V → a(V,B), 0⟩}
{⟨V → a(V,B), 0⟩} a
b {⟨B → b, 0⟩}
b {⟨V → G, 2⟩, ⟨G → B, 1⟩, ⟨B → b, 0⟩}
b {⟨V → G, 2⟩, ⟨G → B, 1⟩, ⟨B → b, 0⟩}

FIGURE 15.4 Subject tree of Figure 15.3 shown with ⟨matching rule, relative cost⟩ pairs.

The problem of computing an optimal derivation tree has to take into account costs as well. We discuss both top-down, as well as bottom-up strategies for solving this problem. All the algorithms we present can solve the following problem, which we will call the *optimal tree-parsing problem*:

Given a cost augmented regular tree grammar G and a subject tree t in *TREES(A)*, find a representation of a cheapest derivation tree for t in G.

Given a specification of the target machine by a regular tree grammar at the semantic level of a target machine, and an IR tree, we distinguish between the following two times when we solve the optimal tree-parsing problem for the IR tree:

1. *Preprocessing time.* This is the time required to process the input grammar, independent of the IR tree. It typically includes the time taken to build the matching automaton or the tables.
2. *Matching time.* This involves all IR tree-dependent operations, and captures the time taken by the driver to match a given IR tree using the tables created during the preprocessing phase.

For the application of code generation, minimizing matching time is important because it adds to compile time, whereas preprocessing is done only once at compiler generation time.

15.3 Top-Down Tree-Parsing Approach

The key idea with this approach is to reduce tree-pattern matching to string pattern matching. Each root to leaf path in a tree is regarded as a string in which the symbols in the alphabet are interleaved with numbers indicating which branch from father to son has been followed. This effectively generates a set of strings. The well-known Aho–Corasick multiple-keyword pattern-matching algorithm [1] is then adapted to generate a top-down, tree-pattern-matching algorithm. The Aho–Corasick algorithm converts the set of keywords into a trie; the trie is then converted into a string pattern-matching automaton that performs a parallel search for keywords in the input string. If K is the set of keywords, then each keyword has a root leaf path in the trie, whose branch labels spell out the keyword. This trie is then converted into a string pattern-matching automaton as follows. The states of the automaton are the nodes of the trie, with the root the start state. All states that correspond to complete keywords are final states. The transitions are just the branches of the trie with the labels representing the input symbols on which the transition is made. There is a transition from the start state to itself on every symbol that does not begin a keyword. The pattern-matching automaton has a failure function for every state other than the start state. For a state reached on input w, this is a pointer to the state reached on the longest prefix of a keyword that is a proper suffix of w. The construction of both the trie as well as the pattern-matching automaton has complexity linear in the sum of the sizes of the keywords. Moreover, matches of the keywords in an input string w are found in time linearly proportional to the length of w. Thus, the string pattern-matching problem can be solved in time

$O(|K| + |w|)$, where K is the sum of the lengths of the keywords and w is the length of the input string [1].

HOD generalized this algorithm for tree-pattern matching by noting that a tree can be defined by its root to leaf paths. A root-to-leaf path contains, alternately, root labels and branch numbers according to a left to right ordering. Consider the tree patterns on the right-hand sides of the regular tree grammar in Example 15.4. Arities of various terminals and operators are given in parentheses next to the operators and terminals, and rules for computing costs shown along with the productions. Actions to be carried out at each reduction are omitted.

Example 15.4
$G = (\{S, R\}, \{:= (2), +(2), deref(1), sp(0), c(0)\}, P, S)$ where P consists of the following productions:

1. $S \rightarrow := (deref\,(sp), R)$	$cost = 3 + cost(R)$
2. $R \rightarrow\ deref\,(sp)$	$cost = 2$
3. $R \rightarrow +(R, c)$	$cost = 1 + cost(R)$
4. $R \rightarrow +(c, R)$	$cost = 1 + cost(R)$
5. $R \rightarrow c$	$cost = 1$

Thus, the patterns on the right-hand sides of productions in Example 15.4 are associated with the following set of path strings:

1. $:=\ 1\ deref\ 1\ sp$
2. $:=\ 2\ R$
3. $deref\ 1\ sp$
4. $+\ 1\ R$
5. $+\ 2\ c$
6. $+\ 1\ c$
7. $+\ 2\ R$
8. c

By using the Aho–Corasick algorithm we can construct the pattern-matching automaton shown in Figure 15.5. The final states are enclosed in double circles and the failure function pointers that do not point to state 0 are shown as dashed lines. Path strings that match at final states are indicated by specifying the trees they belong to next to the state, t_i, indicating the right-hand side of rule i. Once the pattern-matching automaton is constructed the subject tree is traversed in preorder, computing automaton states as we visit nodes and traverse edges.

The top-down, tree-pattern-matching algorithm was adapted to the problem of tree parsing by Aho and Ganapathi, and the presentation that follows is based on [3]. First, the subject tree is traversed in depth-first order using the routine *MarkStates(n)* and the automaton state reached at each node is kept track of. This is displayed in Figure 15.6, where δ is the transition function of the underlying string pattern-matching automaton. The routine also determines the matching string patterns. The scheme described by HOD [28] using bit vectors to decide whether there is a match at a node is used here. With each node of the subject tree, a bit string b_i is associated with every right-hand side pattern t_i, $1 \leq i \leq m$, where m is the total number of patterns. At any node n of the subject tree, bit j of the bit string b_i is 1 iff every path from the ancestor of n at distance j through n to every descendant of n has a prefix which is a path string of the pattern we wish to match. The bit string does not need to be longer than the height of the corresponding pattern. To find a cover of the intermediate code tree, it is necessary to keep track of reductions that are applicable at a node. The routine *Reduce* shown in Figure 15.7 does this. Because we are looking for an optimal cover there is the need to maintain a cost for each tree t_i that matches at a node n. The implementation in [3] allows general cost computation rules to be used in place of simple additive costs. The function $cost(t_i, n)$

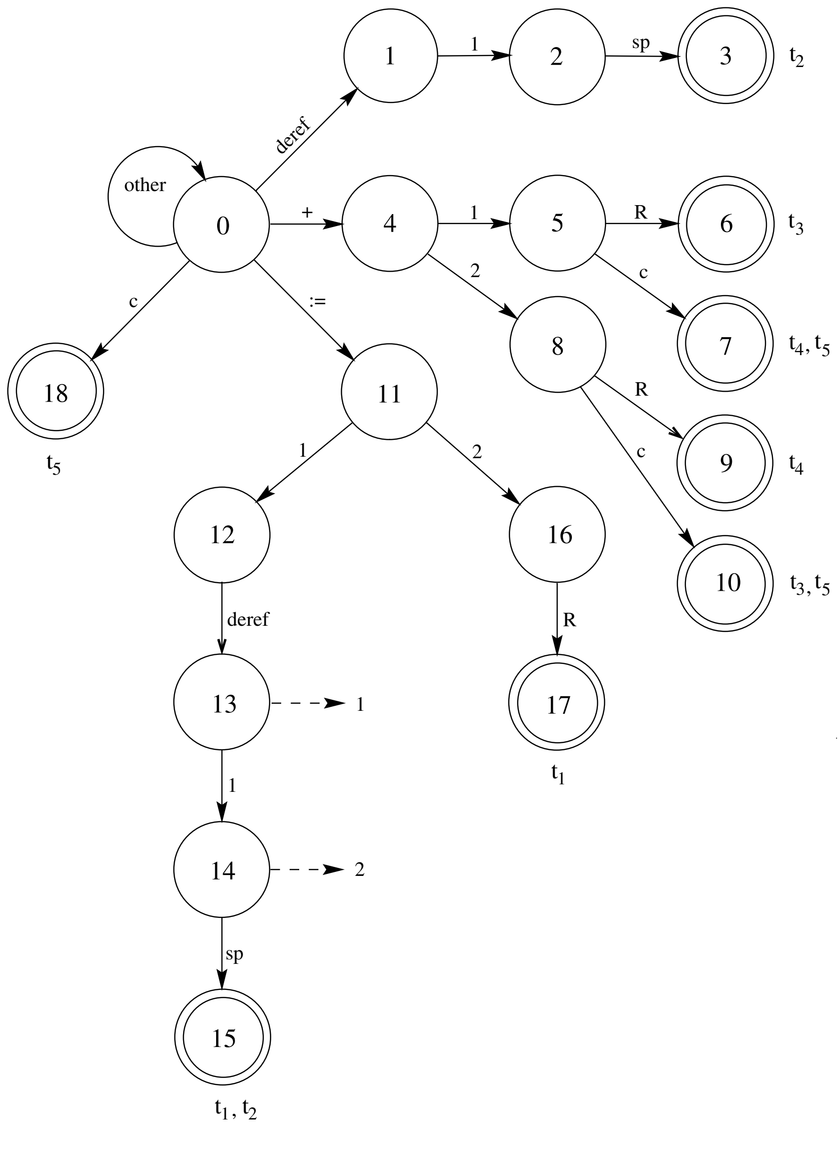


FIGURE 15.5 The tree-pattern-matching automaton for the tree grammar of Example 15.2.

computes this cost for each node n. For each node n, there is an array *n.cost* of dimension equal to the number of nonterminals. The entry corresponding to a nonterminal is the cost of the cheapest match of a rule with that nonterminal on the left-hand side. The index of that rule is stored in the array *n.match* that also has dimension equal to the number of nonterminals. Thus, after the parsing is over, the least cost covers at each node along with their associated costs are available. The result of applying *MarkStates* to the root of the subject tree shown in Figure 15.8 is shown in Figure 15.9.

We trace through the first few steps of this computation.

```
procedure MarkStates (n)
if n is the root then
    $n.state = \delta(0, n.symbol)$
else
    $n.state = \delta(\delta(n.parent.state, k), n.symbol)$
    where n is the $k^{th}$ child of $n.parent$
end if
for every child c of n do
    $MarkStates(c)$
end for
$n.b_i = 0$
if $n.state$ is accepting then
    for each path string of $t_i$ of length $2j + 1$ recognized at $n.state$ do
        $n.b_i = n.b_i$ or $2^j$
    end for
end if
for every righthandside tree pattern $t_i$ do do
    $n.b_i = n.b_i$ or $\Pi_{C \in C(n)} right\_shift(c.b_i)$
    where $C(n)$ is the set of all children of node n
end for
$Reduce(n)$
end procedure
```

FIGURE 15.6 The procedure for visiting nodes and computing states.

```
procedure $Reduce(n)$
$list$ = set of productions $l_i \rightarrow t_i$ such that the zeroth bit of $n.b_i$ is 1
while $list \neq \emptyset$ do
    choose and remove next element $l_i \rightarrow t_i$ from list
    if $cost(t_i, n) < n.cost[l_i]$ then
        $n.cost[l_i] = cost(t_i, n)$
        $n.match[l_i] = i$
        if n is the root then
            $q = \delta(0, l_i)$
        else
            $q = \delta(\delta(n.parent.state, k), l_i)$
            where n is the $k^{th}$ child of $n.parent$
        end if
        if q is an accepting state then
            for each path string of $t_k$ of length $2j + 1$ recognized at q do
                $n.b_k = n.b_k$ or $2^j$
                if the zeroth bit of $n.b_k$ is a 1 then
                    add $l_k \rightarrow t_k$ to list
                end if
            end for
        end if
    end if
end while
end procedure
```

FIGURE 15.7 Procedure for reducing IR trees.

Example 15.5

We use the subject tree of Figure 15.8 with the nodes numbered as shown, and the automaton of Figure 15.5 whose transition function is denoted by δ:

1. The start state is 0.
2. $node(1).state = \delta(0, :=) = 11.$
3. $node(2).state = \delta(\delta(11, 1), deref) = 13.$

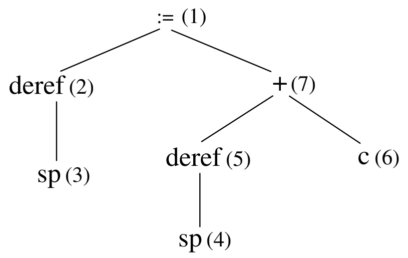


FIGURE 15.8 A subject tree generated by the tree grammar of Example 15.2.

4. $node(3).state = \delta(\delta(13, 1), sp) = 15$.
5. Path strings corresponding to patterns t_1 and t_z are matched at node(3). Bit strings $node(3).b_1 = 100$, node(3).$b_2 = 10$ and all the other bit strings are 0.
6. The routine *Reduce*(*node*(3)) does nothing because no reductions are called for.
7. We now return to *node*(2) and update the bit string $node(2).b_1 = 10$ and $node(2).b_2 = 1$ to reflect the fact that we have moved one level up in the tree.
8. The call to *Reduce*(*node*(2)) notes the fact that the zeroth bit of $node(2).b_2 = 1$. Thus, a reduction by the rule $R \rightarrow$ *deref*(*sp*) is called for. The cost of this rule is 2 and the rule number is 2; thus $Cost(R) = 2$ and $Match(R) = 2$ at *node*(2).
9. We return to *node*(1) and call *MarkState*(*node*(7)) ; *node*(7) is the second child of its parent, the failure function is invoked at state 16 on which a transition is made to state 0, and then from state 0 to state 4 on the symbol +. Thus, $node(7).state = 4$.
10. $node(5).state = \delta(\delta(4, 1), deref) = 1$. (Note, the failure function is invoked here again at state 5.)
11. $node(4).state = \delta(\delta(1, 1), sp) = 3$.
12. A path string corresponding to pattern t_2 is matched at *node*(4). The bit string $node(4).b_2 = 10$ and all other bit strings are 0.
13. The routine *Reduce*(*node*(4)) does nothing because no reductions are called for.
14. We return to *node*(5); the bit string $node(5).b_2 = 1$ to reflect the fact that we have moved one level up in the tree.
15. The call to *Reduce*(*node*(5)) notes the fact that the zeroth bit of $node(5).b_2$ is a 1. Thus, a reduction by the rule $R \rightarrow deref(sp)$ is called for. The cost of this rule is 2, and the number of the rule is 2. Thus $Cost(R) = 2$ and $Match(R) = 2$ at *node*(5).
16. The variable q at *node*(5) is updated to reflect the state after reduction. Thus, $q = \delta(\delta(4, 1), R) = 6$. The state 6 is an accepting state, which matches a string pattern of t_3. Therefore $node(5).b_3 = 10$.
17. Continuing in this manner we obtain the labels in Figure 15.9.

Once a cover has been found, the reductions are performed during which time the action parts of the productions constituting the cover are executed. As we observed earlier, a reduction may be viewed as replacing a subtree corresponding to the right-hand side of a production with the left-hand side nonterminal. In addition, the action part of the rule is also executed. Usually actions associated with reductions are carried out in depth-first order.

15.4 Bottom-Up Tree-Parsing Approaches

We begin with cost-augmented regular tree grammars in normal form. We first describe a strategy for the generation of tables representing a tree automaton with states that do not encode cost information.

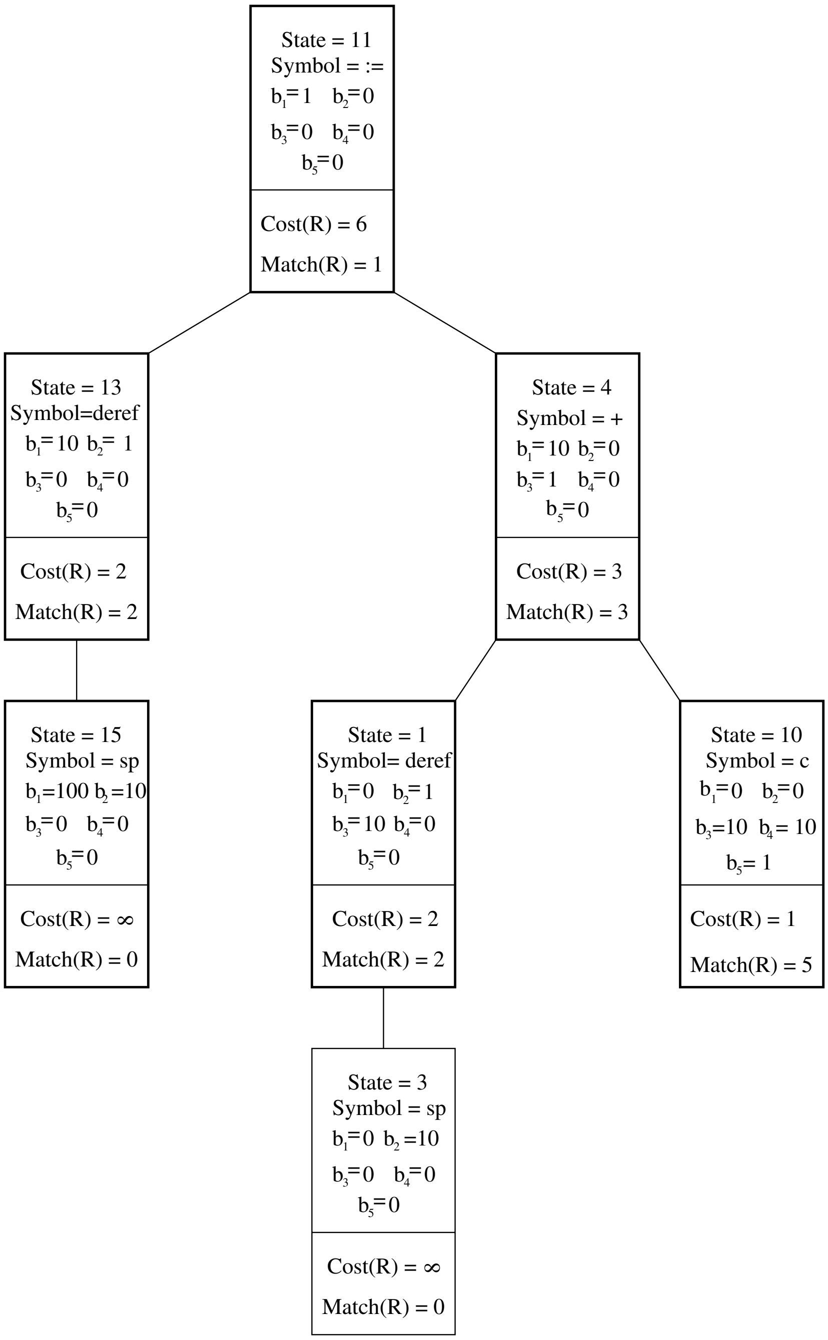


FIGURE 15.9 The information at each node after *MarkStates* is applied to the subject tree of Figure 15.8.

By using such an automaton, cost computations for generating locally optimal code have to be performed at code generation time (i.e., dynamically).

Our aim is to find at each node n in the subject tree, minimal cost rules for each nonterminal that matches at the node. We call such a set of nonterminals *matching nonterminals* at node n. If the intermediate code tree is in the language generated by the tree grammar, we expect one of the nonterminals that matches at the root to be the start symbol of the regular tree grammar. The information about rules and nonterminals that match as we go up the tree can be computed with the help of a bottom-up, tree-pattern-matching automaton built from the specifications. Thus, during the matching or code generation phase, we traverse the intermediate code tree bottom up, computing states and costs as we go along. For a given nonterminal in a set we retain only the minimal cost rule associated with that nonterminal. Finally, when we reach the root of the tree, we have associated with the start symbol the minimal cost of deriving the tree from the start nonterminal. Next in a top-down pass we select the nonterminals that yield the minimal cost tree and generate code as specified in the translation scheme.

We present both, an iterative algorithm and a work list-based algorithm drawing from the work of Balachandran, Dhamdhere and Biswas [9] and Proebsting [39].

15.4.1 Iterative Bottom-Up Preprocessing Algorithm

Let $G = (N, A, P, S)$ be a regular tree grammar in normal form. Before describing the algorithm, we describe some functions that we use in the computation. Let *maxarity* be the maximum arity of an operator in A. Let I_l be the set of positive integers of magnitude at most maxarity:

1. $\mathit{rules} : T \cup OP \mapsto 2^P$
   For $a \in T$, $\mathit{rules}(a) = \{r | r : n \to a \in P\}$
   For $op \in OP$, $\mathit{rules}(op) = \{r : | r : n \to op(n_1, n_2, \ldots n_k) \in P\}$
   The set *rules*(a) contains all production rules of the grammar whose right-hand side tree-patterns are rooted at a.
2. $\mathit{child\_rules} : N \times I_l \mapsto 2^P$
   $\mathit{child\_rules}(n, i) \ = \{r | r : n_l \to op(n_1, n_2, \ldots, n_k)$ and $n_i = n\}$
   The set *child_rules*(n, i) contains all those productions such that the ith nonterminal on the right-hand side is n. The function can be extended to a set of nonterminals N_1 as follows:

$$\mathit{child\_rules}(N_1, i) = \bigcup_{n \in N_1} \mathit{child\_rules}(n, i)$$

3. $\mathit{child\_NT} : OP \times I_l \mapsto 2^N$
   $\mathit{child\_NT}(op, j) \ = \{n_j | r : n_l \to op\, n_1 n_2, \ldots, n_j, \ldots, n_k \in P\}$
   In other words *child_NT*(op, j) is the set of all nonterminals that can appear in the jth position in the sequence of nonterminals on the right-hand side of a production for operator *op*. (If j exceeds *arity*(*op*) the function is not defined.)
4. $\mathit{nt} : P \mapsto 2^N$
   $\mathit{nt}(r) = \{n | r : n \to \alpha \in P\}$
   The set *nt*(r) contains the left-hand side nonterminal of the production. The function can be extended to a set R of rules as follows:

$$nt(R) = \bigcup_{r \in R} nt(r)$$

5. $\mathit{chain\_rule\_closure} : N \mapsto 2^P$
   $\mathit{chain\_rule\_closure}(n) = \{r | r \in P, r : n_1 \to n_2, n_1 \Rightarrow n_2 \Rightarrow^* n\}$
   The set *chain_rule_closure* of a nonterminal is the set of all rules that begin derivation sequences that derive that nonterminal and that contain only chain rules. The function can be extended to a set of nonterminals as follows:

$$\mathit{chain\_rule\_closure}(N_1) = \bigcup_{n \in N_1} \mathit{chain\_rule\_closure}(n)$$

Given a regular tree grammar in normal form and a subject tree, function *Match* in Figure 15.10 computes the rules that match at the root. For the time being, we ignore the costs of the rules.

```
function $Match(t)$
begin
if $t = a \in T$ then
    $match\_rules = rules(a)$;
    $match\_NT = nt(match\_rules)$;
    $Match = match\_rules \cup chain\_rule\_closure(match\_NT)$;
else
    let $t = op(t_1, t_2, \ldots, t_k)$ where $arity(op) = k$
    for $i = 1$ to $k$ do
        $mr_i = Match(t_i)$
        $nt_i = nt(mr_i)$
    end for
    $match\_rules = rules(op) \cap child\_rules(nt_1, 1) \cap child\_rules(nt_2, 2) \ldots \cap$
    $child\_rules(nt_k, k)$
    $match\_NT = nt(match\_rules)$;
    $Match = match\_rules \cup chain\_rule\_closure(match\_NT)$
end if
end function
```

FIGURE 15.10 Function that computes matching rules and nonterminals.

```
function $TableMatch(t)$
begin
if $t = a \in T$ then
    $TableMatch = \tau_a$
else
    let $t = op(t_1, t_2, \ldots, t_k)$ where $arity(op) = k$
    $TableMatch = \tau_{op}(TableMatch(t_1), TableMatch(t_2) \ldots TableMatch(t_k))$
end if
end function
```

FIGURE 15.11 Function that computes matches using precomputed tables.

The function computes the matching rules at the root of the subject tree by recursively computing matching rules and hence matching nonterminals at the children. This suggests a bottom-up strategy that computes matching rules and nonterminal sets at the children of a node before computing the sets at the node. Each such set can be thought of as a state. The computation that finds the matching rules at a node from the nonterminals that match at its children does not need to be performed at matching time because all the sets under consideration are finite and can be precomputed and stored in the form of tables, one for each operator and terminal. This set of tables is actually an encoding of the transition function of a finite state bottom-up, tree-pattern-matching automaton, which computes the state at a node corresponding to an operator, from the states of its children. For a terminal symbol $a \in T$, the table τ_a contains just one set of rules. For an operator $op \in OP$, of arity k, the table τ_{op} is a k-dimensional table. Each dimension is indexed with indices corresponding to the states described earlier. Assume that such tables are precomputed and stored as auxiliary information to be used during matching. Function *TableMatch* shown in Figure 15.11 can find the matching rules at the root of a subject tree.

The function *TableMatch* computes transitions in a bottom-up fashion performing a constant amount of computation per node. Thus, matching time with precomputed tables is linear in the size of the subject tree. The size of the table for an operator of arity k is $O(2^{|N| \times \, maxarity})$. The table sizes computed in this manner are huge and can be reduced in the following way.

Assume that *States* is the set of states that is precomputed and indexed by integers from the set I. Let us call each element of *States* an *itemset*. Each such set consists of a set of rules, satisfying the property that there is some subject tree matched by exactly this set of rules. Let *itemset*(i) denote the set indexed by i. We define for each operator *op* and each dimension j, $1 \leq j \leq$ *arity*(*op*) an equivalence relation R_{op}^{j} as follows: for i_p and $i_q \in I$, $i_p\ R_{op}^{j}\ i_q$ if $nt(itemset(i_p)) \cap child\_NT(op, j) = nt(itemset(i_q)) \cap child\_NT(op, j)$. In other words, two indices are put into the same equivalence class of operator *op* in dimension j if their corresponding nonterminal sets project onto the same sets in the jth dimension of operator *op*. If i_p and i_q are in the same equivalence class of R_{op}^{j}, then it follows that for all $(i_1, i_2, \ldots, i_{j-1}, i_{j+1}, \ldots, i_k)$, $\tau_{op}(i_1, i_2, \ldots, i_{j-1}, i_p, i_{j+1}, \ldots, i_k) = \tau_{op}(i_1, i_2, \ldots, i_{j-1}, i_q, i_{j+1}, \ldots, i_k)$. For the case $k = 2$ what this means is that i_p and i_q correspond to the indices of identical rows or columns. This duplication can be avoided by storing just one copy. We therefore use index maps as follows. The mapping from the set of indices in I to the set of indices of equivalence classes of R_{op}^{j} denoted by I_{op}^{j} is denoted by μ_{op}^{j}. Thus we have the mapping:

$$\mu_{op}^{j} : I \mapsto I_{op}^{j},\ 1 \leq j \leq k,\ arity(op) = k$$

The table τ_{op} is now indexed by elements of I_{op}^{j} in dimension j instead of those of I. At matching time one extra index table lookup is necessary in each dimension to obtain the resulting element of *States*. This is expressed by the following relation that replaces the table lookup statement of function *TableMatch*:

$$\tau_{op}(i_1, i_2, \ldots, i_k) = \theta_{op}(\mu_{op}^{1}(i_1), \mu_{op}^{2}(i_2), \ldots, \mu_{op}^{k}(i_k))$$

where θ_{op} is the compressed table.

The next step is the generation of compressed tables directly. Informally, the algorithm works as follows. It begins by finding elements of *States* for all symbols of arity 0. It then finds elements of *States* that result from derivation trees of height increasing by one at each iteration until there is no change to the set *States*. At each iteration, elements of *States* corresponding to all operators that contribute to derivation trees of that height are computed. For each operator *op* and each dimension j of that operator, only nonterminals in $child\_NT(op, j)$ that are also members of a set in *States* computed so far can contribute to new sets associated with *op*. Such a collection of subsets for *op* in the jth dimension at iteration i is called *repset*(*op, j, i*). Thus, choices for a sequence of children of *op* are confined to elements drawn from a tuple of sets in the Cartesian product of collections at each iteration. Each such tuple is called a *repset_tuple*. (In fact, iteration is confined only to elements drawn from new tuples formed at the end of every iteration.) At the end of each iteration, the new values of *repset*(*op, j, i*) are computed for the next iteration. The computation is complete when there is no change to *repset*(*op, j, i*) for all operators in all dimensions, for then no new tuples are generated. The procedure for precomputing compressed tables is given in Figure 15.12.

We illustrate with the help of an example (adapted from [20]).

Example 15.6

Let the following be the rules of a regular tree grammar. The rules and nonterminals are numbered for convenience:

1.	$stmt \rightarrow := (addr, reg)$	[1]
2.	$addr \rightarrow +(reg, con)$	[0]
3.	$addr \rightarrow reg$	[0]
4.	$reg \rightarrow +(reg, con)$	[1]
5.	$reg \rightarrow con$	[1]
6.	$con \rightarrow CONST$	[0]

The nonterminals are numbered as follows: *stmt* $= 1$, *addr* $= 2$, *reg* $= 3$ and *con* $= 4$.

```
procedure MainNoCost()
$States = \emptyset$
$itemset = \emptyset$
for each $a \in T$ do
    $mrules = rules(a)$
    $mnonterminals = nt(mrules(a))$
    $match\_rules = mrules \cup chain\_rule\_closure(mnonterminals)$
    $itemset = match\_rules$
    $States = States \cup \{itemset\}$
end for
generate $child\_NT(op, j)$ for each $op \in OP$ and $j, 1 \leq j \leq arity(op)$;
generate $repset(op, j, 1)$ for each $op \in OP$ and $j, 1 \leq j \leq arity(op)$ and
update index maps;
$i = 1$; $repset\_product_0 = \emptyset$
repeat
    for each $op \in OP$ do
        let $repset\_product_i = \Pi_{j=1 \ldots arity(op)} repset(op, j, i)$
        for each $repset\_tuple = (S_1, S_2, \ldots S_k) \in repset\_product_i - repset\_product_{i-1}$
        do
            $itemset = \emptyset$
            for each $(n_1, n_2 \ldots n_k)$ with $n_i \in S_i, 1 \leq i \leq k$ do
                $mrules = \{r : n \rightarrow op(n_1, n_2 \ldots n_k) \in P\}$
                $mnonterminals = nt(mrules)$
                $match\_rules = mrules \cup chain\_rule\_closure(mrules)$
                $itemset = itemset \cup match\_rules$
            end for
            $\theta_{op}(S_1, S_2 \ldots S_k) = itemset$
            $States = States \cup \{itemset\}$
        end for
    end for
    $i = i + 1$;
    generate $repset(op, j, i)$ for each $op \in OP$ and $j, 1 \leq j \leq arity(op)$ and
    update index maps
until $repset(op, j, i) = repset(op, j, i-1) \forall op \forall j, 1 \leq j \leq arity(op)$
end
```

FIGURE 15.12 Algorithm to precompute compressed tables without costs.

The operators are := and $+$ both of arity 2, and there is a single terminal *CONST* of arity 0. The results of the first iteration of the algorithm follow:

1. There is only one symbol of arity 0, namely, *CONST*.
   $mrules = \{con \rightarrow CONST\}$.
   $mnonterminals = \{con\}$.
   $match\_rules = \{con \rightarrow CONST, reg \rightarrow con, addr \rightarrow reg\}$.
   Thus after processing symbols of arity zero
   $States = \{con \rightarrow CONST, reg \rightarrow con, addr \rightarrow reg\}$. Assume this set has index 1. Thus $I = \{1\}$.
   Referring to the state by its index, $nt(1) = \{con, reg, addr\}$.
2. Consider the operator $+$.
   The set $child\_NT(+, 1) = \{reg\}$ and $child\_NT(+, 2) = \{con\}$.
   Thus, $repset(+, 1, 1) = child\_NT(+, 1) \cap nt(1) = \{\{reg\}\}$, $\mu_+^1(1) = 1$, $I_+^1 = \{1\}$.
   Here the projection onto the first dimension of operator $+$ gives the set containing a single set $\{reg\}$ assigned index 1. For ease of understanding, we use the indices and the actual sets they represent interchangeably.
   $repset(+, 2, 1) = child\_NT(+, 2) \cap nt(1) = \{\{con\}\}$, $\mu_+^2 = 1$, $I_+^2 = \{1\}$.
   Thus, for $i = 1$ *repset_product* for $+ = \{\{\{reg\}\}, \{\{con\}\}\}$.

For $repset\_tuple = (\{reg\}, \{con\}), i = 1$.
$mrules = \{reg \rightarrow +(reg, con), addr \rightarrow +(reg, con)\}$.
$mnonterminals = \{reg, addr\}$.
$match\_rules = \{reg \rightarrow +(reg, con), addr \rightarrow +(reg, con), addr \rightarrow reg\}$.
This set *match_rules* is added as a new element of *States* with index 2.
Thus, $\theta_+(1, 1) = 2$.
There are no more states added due to operator + at iteration 1.

3. Consider the operator :=.
   The *set* $child\_NT(1, :=) = \{addr\}$ and $child\_NT(2, :=) = \{reg\}$.
   Thus, $repset(:=, 1, 1) = \{\{addr\}\}$ and $\mu^1_{:=} = 1, I^1_{:=} = \{1\}$.
   $repset(:=, 2, 1) = \{\{reg\}\}$ and $\mu^2_{:=} = 1, I^1_{:=} = \{1\}$.
   Thus, for $i = 1$ *repset_product* for *operator* := = $\{ \{\{addr\}\}, \{\{reg\}\}\}$.
   For $repset\_tuple = (\{addr\}, \{reg\})$ and $i = 1$.
   $mrules = \{stmt \rightarrow := (addr, reg)\}, mnonterminals = \{stmt\}$.
   $match\_rules = \{stmt \rightarrow := (addr, reg)\}$.
   A new state corresponding to *match_rules* is added to *States* with index 3. Thus, $\theta_{:=}(1, 1) = 3$.
   There are no more states added due to *operator* := at iteration 1.
4. At the end of the iteration for $i = 1$, $States = \{1, 2, 3\}$.
5. It turns out that no more states can be added to *States*.

We next show how costs can be included in the states of the bottom-up, tree-pattern-matching automaton. The information we wish to capture is the following. Supposing we had a subject tree t and we computed all matching rules and nonterminals as well as minimal costs for each rule and each nonterminal that matched at a node. If we now compute the difference between the cost of each rule and that of the cheapest rule matching at the same node in the tree, we obtain the differential cost. If these differential costs are bounded, they can be precomputed and stored as part of the item in the itemset. Likewise, we can store differential costs with each nonterminal:

Let as before $match\_rules(t)$ be the set of rules matching at the root of a subject tree t. We now define the set of (rule,cost) pairs, *itemset* matching the root of t:

$$itemset = \{(r, \Delta_r) | r \in match\_rules(t), \Delta_r = cost(r) - min\{cost(r') | r' \in match\_rules(t)\}\}$$

If the costs are bounded for all such pairs, we can precompute them by augmenting the procedure in Figure 15.12.
The function that performs the computation for arity zero symbols is given in Figure 15.13.

Given this procedure we present the algorithm for precomputing tables with costs. We note that $repset(\text{op}, i, j)$ is a collection of sets whose elements are $\langle$*nonterminal, cost*$\rangle$ pairs. The iterative procedure *IterativeMain*, for precomputation of itemsets, first calls *IterativeArityZeroTables* to create the tables for symbols of arity zero. It then iterates over patterns of increasing height until no further items are generated. Procedure *IterativeComputeTransitions* creates the new states for each operator at each iteration and updates *States*.

We illustrate the procedure for the grammar of Example 15.6.

Example 15.7
The following steps are carried out for the only symbol *CONST* of arity zero.

1. $mrules = \{6 : con \rightarrow CONST\}, mnonterminals = \{con\}$.
2. $match\_rules = \{6 : con \rightarrow CONST, 5 : reg \rightarrow con, 3 : addr \rightarrow con\}, match\_NT = \{con, reg, addr\}$.
3. $\Delta_6 = \infty, \Delta_5 = \infty, \Delta_3 = \infty$.

```
procedure IterativeArityZeroTables
States = ∅
for each $a \in T$ do
    $itemset = \emptyset$
    $mrules = rules(a)$
    $mnonterminals = nt(rules(a))$
    $match\_rules = mrules \cup chain\_rule\_closure(mnonterminals)$
    $match\_NT = nt(match\_rules)$
    $\Delta_r = \infty, r \in match\_rules$
    $D_n = \infty, n \in match\_NT$
    $COST_{min} = min\{rule\_cost(r) | r \in mrules\}$
    for each $r$ in mrules do
        $\Delta_r = COST_r - COST_{min}$
    end for
    for each $n$ in mnonterminals do
        $D_n = min\{\Delta_r | \exists r \in mrules, n \in nt(r)\}$
    end for
    repeat
        for each $r : n \rightarrow n_1$ such that $r \in chain\_rule\_closure(mnonterminals)$
        do
            $D_n = min\{D_n, D_{n_1} + rule\_cost(t)\}$
            $\Delta_r = min\{\Delta_r, D_{n_1} + rule\_cost(r)\}$
        end for
    until no change to any $D_n$ or $\Delta_r$
    $itemset = \{(r, \Delta_r)\} | r \in match\_rules\}$
    $\tau_a = itemset$
    $States = States \cup \{itemset\}$
end for
end procedure
```

FIGURE 15.13 Computation of arity zero tables with static costs.

4. $D_4 = \infty, D_3 = \infty, D_2 = \infty$.
5. $COST_{min} = min\{rule\_cost(con \rightarrow CONST)\} = 0$.
6. $\Delta_6 = rule\_cost(con \rightarrow CONST) - COST_{min} = 0$.
7. $D_4 = 0$.
8. After the first iteration of the repeat-until loop $D_4 = 0, D_3 = 1, D_2 = \infty, \Delta_6 = 0, \Delta_5 = 1, \Delta_3 = \infty$.
9. After the second iteration of the repeat-until loop $D_4 = 0, D_3 = 1, D_2 = 1, \Delta_6 = 0, \Delta_5 = 1, \Delta_3 = 1$.
10. There is no change at the next iteration so $States = \{\{\langle con \rightarrow CONST, 0\rangle, \langle reg \rightarrow con, 1\rangle, \langle addr \rightarrow reg, 1\rangle\}\}$.

We next consider operator + of arity 2. $child\_NT(+, 1) = \{reg\}$, $child\_NT(+, 2) = \{con\}$. $repset(+, 1, 1) = \{\{\langle reg, 0\rangle\}\}$, $repset(+, 2, 1) = \{\{\langle con, 0\rangle\}\}$. The following steps are then carried out for operator + at the first iteration:

1. $repset\_product_1 = \{\{\langle reg, 0\rangle\}\} \times \{\{\langle con, 0\rangle\}\}$.
2. $repset\_tuple = (\{\langle reg, 0\rangle\}, \{\langle con, 0\rangle\})$.
3. $mrules = \{2 : addr \rightarrow +\ (reg\ \ con), 4 : reg \rightarrow +\ (reg\ \ con)\}$.
4. $C_{rhs,2} = 0, C_{rhs,4} = 0$.
5. $mnonterminals = \{addr,\ reg\}$.
6. $match\_rules = \{2 : addr \rightarrow +\ (reg\ \ con), 4 : reg \rightarrow +\ (reg\ \ con), 3 : addr \rightarrow reg\}$.
7. $match\_NT = \{addr,\ reg\}$.
8. $\Delta_2 = \Delta_4 = \Delta_3 = \infty$.
9. $D_3 = D_2 = \infty$.

10. $COST_2 = 0 + 0 = 0$, $COST_4 = 0 + 1 = 1$, $COST_{min} = 0$.
11. $\Delta_2 = 0$, $\Delta_4 = 1$, $D_2 = 0$, $D_3 = 1$.
12. There is no change to these sets during the first iteration of the while loop, hence the value of *itemset* after discarding more expensive rules for the same nonterminal is: $itemset = \{\langle addr \rightarrow + (reg\ \ con), 0\rangle, \langle reg \rightarrow + (reg\ \ con), 1\rangle\}$.
13. Thus $States = \{\{\langle con \rightarrow CONST, 0\rangle, \langle reg \rightarrow con, 1\rangle, \langle addr \rightarrow reg, 1\rangle\}\} \cup \{\{\langle addr \rightarrow + (reg\ \ con), 0\rangle, \langle reg \rightarrow + (reg\ \ con), 1\rangle\}\}$.

After processing the operator := in a similar manner we get the following three itemsets in *States*:

1. $\{\langle con \rightarrow CONST, 0\rangle, \langle reg \rightarrow con, 1\rangle, \langle addr \rightarrow reg, 1\rangle\}$.
2. $\{\langle addr \rightarrow + (reg\ \ con), 0\rangle, \langle reg \rightarrow + (reg\ \ con), 1\rangle\}$.
3. $\{\langle stmt \rightarrow := (addr,\ reg), 0\rangle\}$.

15.4.2 Work List-Based Approach to Bottom-Up Code-Generator Generators

Proebsting [39] employs a work list approach to the computation of itemsets; the presentation that follows is based on [39]. A state is implemented as a set of tuples, each tuple containing:

1. A nonterminal that matches a node
2. The normalized cost of this nonterminal
3. The rule that generated this nonterminal at minimal cost

A tuple structured as shown earlier is called an *item*; a collection of such items is termed an *itemset*. Each itemset represents a state of the underlying cost augmented tree-pattern-matching automaton whose set of states is *States*. Each itemset is represented as an array of (rule,cost) pairs indexed by nonterminals. Thus *itemset*[*n*].*cost* refers to the normalized cost of nonterminal *n* of the itemset, and *itemset*[*n*].*rule* gives a rule that generates that nonterminal at minimal cost. A cost of ∞ in any position indicates that no rule derives the given nonterminal. The empty state ($\emptyset$) has all costs equal to infinity.

```
procedure IterativeMain()
IterativeArityZeroTables
generate $repset(op, j, 1)$ for each $op \in OP$ and $j, 1 \leq j \leq arity(op)$
$i = 1; repset\_product_0 = \emptyset$
repeat
    for each $op \in OP$ do
        $IterativeComputeTransition(op, i)$
    end for
    $i = i + 1$
    generate $repset(op, j, i)$ for each $op \in OP$ and $j, 1 \leq j \leq arity(op)$ and
    update index maps
until $repset(op, j, i) = repset(op, j, i - 1) \forall op \forall j, 1 \leq j \leq arity(op)$
end
```

FIGURE 15.14 Procedure to precompute reduced tables with static costs.

```
procedure IterativeComputeTransition(op, i)
let $repset\_product_i = \Pi_{j=1...arity(op)} repset(op, j, i)$
for each $repset\_tuple = (S_1, S_2, \ldots S_k) \in repset\_product_i - repset\_product_{i-1}$
do
    $itemset = \emptyset$; $mrules = \emptyset$
    for each $(\langle n_1, D_{n_1}\rangle, \langle n_2, D_{n_2}\rangle \ldots \langle n_k, D_{n_k}\rangle), \langle n_i, D_i\rangle \in S_i$
    do
        if $r : n \rightarrow op(n_1, n_2, \ldots n_k) \in p$ then
            $C_{rhs,r} = D_{n_1} + D_{n_2} + \ldots D_{n_k}$
            $mrules = mrules \cup \{r\}$
        end if
    end for
    $mnonterminals = nt(mrules)$
    $match\_rules = mrules \cup chain\_rule\_closure(mrules)$
    $mtach\_NT = nt(match\_rules)$
    $\Delta_r = \infty, r \in match\_rules$; $D_n = \infty, n \in match\_NT$
    for each $r$ in $mrules$ do
        $COST_r = C_{rhs,r} + rule\_cost(r)$
    end for
    $COST_{min} = min\{COST_r | r \in mrules\}$
    for each $r$ in $mrules$ do
        $\Delta_r = COST_r - COST_{min}$
    end for
    for each $n$ in $mnonterminals$ do
        $D_n = min\{\Delta_r | n \in nt(r)\}$
    end for
    repeat
        for each $r : n \rightarrow n_1$ such that $r \in chain\_rule\_closure(match\_NT)$
        do
            $D_n = min\{D_n, D_{n_1} + rule\_cost(r)\}$
            $\Delta_r = min\{\Delta_{n_1} + rule\_cost(r)\}$
        end for
    until no change to any $D_n$ or $\Delta_n$
    $itemset = itemset \cup \{(r, \Delta_r)\} | r \in match\_rules, \Delta_r \leq \Delta_{r'}$ if $nt(r) =$
    $nt(r')\}$
    $\theta_{op}(S_1, S_2 \ldots S_k) = itemset$
    $States = States \cup \{itemset\}$
end for
end procedure
```

FIGURE 15.15 Procedure for computing transitions on operators.

```
procedure WorklistMain()
$States = \emptyset$
$WorkList = \emptyset$
WorklistArityZeroTables
while $WorkList \not= \emptyset$ do
    $itemset =$ next itemset from $WorkList$
    for $op \in OP$ do
        $WorklistComputeTransition(op, itemset)$
    end for
end while
end procedure
```

FIGURE 15.16 Work list processing routine.

The procedure *WorklistMain*() in Figure 15.16 manipulates a work list that processes itemsets. Assume that *States* is a table that maintains a one-to-one mapping from itemsets to nonnegative integers. The routine *WorklistArityZeroTables* in Figure 15.17 computes the tables for all terminals in T.

```
procedure WorklistArityZeroTables
for $a \in T$ do
    $itemset = \emptyset$
    for each $r \in rules(a)$ do
        $itemset[nt(r)] = (r, rule\_cost(r))$
    end for
    // normalize costs
    for all $n \in N$ do
        $itemset[n].cost = itemset[n].cost - min_i\{itemset[i].cos\}$
    end for
    // compute chain rule closure
    repeat
        for all $r$ such that $r : n \rightarrow m$ is a chain rule do
            $cost = rule\_cost(r) + itemset[m].cost$
            if $cost < itemset[n].cost$ then
                $itemset[n] = (r, cost)$
            end if
        end for
    until no changes to $itemset$
    Append $itemset$ to $WorkList$
    $States = States \cup \{itemset\}$
    $\tau_a = itemset$
end for
end procedure
```

FIGURE 15.17 The computation of tables of arity zero.

The routine *WorklistComputeTransition* shown in Figure 15.18 augments the operator tables with a new transition computed from an itemset in the work list. The itemset is projected in each dimension of each operator and combined with other representer sets for that operator to check whether the combination leads to a new state. The closure is computed only if this is a new state. Finally, the itemset is added to the work list and the set of states, and the appropriate transition table is updated.

Proebsting has shown that an optimization he calls *state trimming* considerably reduces table sizes. We briefly explain one of the optimizations called *triangle trimming*. Consider the two derivation trees shown in Figure 15.19. Both these have the same root and leaves except for a single leaf nonterminal. Both trees use different rules for the operator *op* to reduce to A. Let $r_1 : A \rightarrow op(X, Q)$ and $r_2 : B \rightarrow op(Y, R)$ with $A \rightarrow B$, $R \rightarrow Q$ and $Y \rightarrow Z$ the chain rules. Triangle trimming notes that both reductions to A involve different nonterminals for a left child state related to operator *op* that occur in the same state. Let that state be *state*. If *state*[X].*cost* exceeds or equals *state*[Z].*cost* in all contexts, then we can eliminate nonterminal X from all such states. Considerable savings in storage have been reported using this optimization.

15.4.3 Hard Coded Bottom-Up Code-Generator Generators

Hard coded code-generator generators are exemplified in the work of Fraser, Hanson and Proebsting [20] and Emmelmann, Schroer and Landwehr [17]. They mirror their input specifications in the same way that recursive descent parsers mirror the structure of the underlying *LL*(1) grammar. Examples of such tools are **BEG** [17] and **iburg** [20]. Code generators generated by such tools are easy to understand and debug because the underlying logic is simple. The code generator that is output typically works in two passes on the subject tree. In a first bottom-up, left-to-right pass it labels each node with the set of nonterminals that match at the node. Then in a second top-down pass, it visits each node, performing appropriate semantic actions, such as generating code. The transition tables used in the techniques described earlier in this section are thus encoded in the flow of control of the code generator, with cost computations performed dynamically.

```
procedure WorklistComputeTransition$(op, itemset)$
for $i = 1$ to $arity(op)$ do
    $repstate = \emptyset$
    for $n \in N$ do
        if $child\_rules(n, i) \cap rules(op) \neq \emptyset$ then
            $repstate[n].cost = itemset[n].cost$
        end if
    end for
    for all $n \in N$ do
        $repstate[n].cost = repstate[n].cost - min_i\{repstate[i].cost\}$
    end for
    $\mu^i_{op}(itemset) = repstate$
    if $repstate \notin I^i_{op}$ then
        $I^i_{op} = I^i_{op} \cup \{repstate\}$
        for each $repset\_tuple = (S_1, S_2, \ldots S_{i-1}, repstate, S_{i+1}, \ldots S_k)$ where $S_j \in$
        $I^j_{op}, j \neq i$ do
            $newitemset = \emptyset$
            for each rule $r$ of the form $n \rightarrow opn_1n_2 \ldots n_{arity(op)}$ in $rules(op)$ do
                $cost = rule\_cost(r) + repstate[n_i].cost + \Sigma_{j \neq i} S_j[n_j].cost$
                if $cost < newitemset[n].cost$ then
                    $newitemset[n] = (r, cost)$
                end if
            end for
            for all $n \in N$ do
                $newitemset[n].cost = newitemset[n].cost - min_i\{newitemset[i].cost\}$
            end for
            if $newitemset \notin States$ then
                repeat
                    for all $r$ such that $r : n \rightarrow m$ is a chain rule do
                        $cost = rule\_cost(r) + newitemset[m].cost$
                        if $cost < newitemset[n].cost$ then
                            $newitemset[n] = (r, cost)$
                        end if
                    end for
                until no change to $newitemset$
                append $newitemset$ to WorkList
                $States = States \cup \{newitemset\}$
            end if
            $\theta_{op}(S_1, S_2, \ldots, S_{i-1}, repset, S_{i+1}, \ldots S_k) = newitemset$
        end for
    end if
end for
end procedure
```

FIGURE 15.18 Procedure to compute transitions on operators.

15.4.4 Code Generation Pass

Following the first pass, where all the nodes of the IR tree are labeled with a state, a second pass over the tree generates the optimal code. Each rule has associated with it a case number that specifies a set of actions to be executed when the rule is matched. The actions could include allocation of a register, emission of code or computation of some attribute. During this phase each node n can be assigned a list of case numbers stored in *n.caselist* in the reverse order of execution. This is described in procedures *GenerateCode* and *TopDownTraverse* in Figures 15.20 and 15.21, respectively.

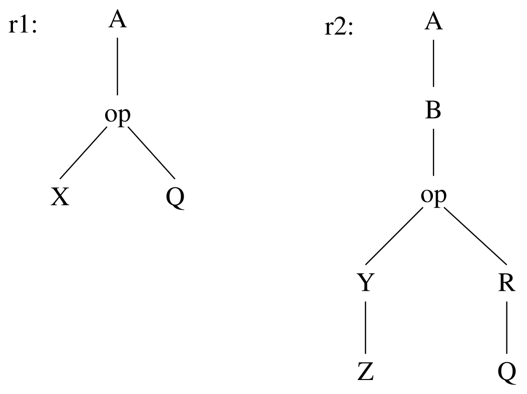


FIGURE 15.19 Two derivation trees to illustrate triangle trimming.

```
**procedure** *GenerateCode*
$top\_down\_traverse(root, S)$
execute actions in $root.caselist$ in reverse order of the list
**end procedure**
```

FIGURE 15.20 Code generation routine.

```
**procedure** $TopDownTraverse(node, nonterminal)$
**if** $node$ is a leaf **then**
    **if** $node.state[nonterminal].rule$ is $r : X \rightarrow Y, Y \in N$ **then**
        append case number of $r$ to $node.caselist$
        $TopDownTraverse(node, Y)$
    **else**
        **if** $node.state[nonterminal].rule$ is $r : X \rightarrow a, a \in T$ **then**
            append case number of $r$ to $node.caselist$
            execute actions in $node.caselist$ in reverse order
        **end if**
    **end if**
**else**
    **if** $node.state[nonterminal].rule$ is $r : X \rightarrow Y, Y \in N$ **then**
        append case number of $r$ to $node.caselist$
        $TopDownTraverse(node, Y)$
    **else**
        **if** $node.state[nonterminal].rule$ is $r : X \rightarrow op(X_1, X_2, \ldots X_k)$ **then**
            append case number of $r$ to $node.caselist$
            **for** $i = 1$ to $k$ **do**
                $TopDownTraverse(child(i, node), X_i)$
            **end for**
        **end if**
        execute actions in $node.caselist$ in reverse order
    **end if**
**end if**
**end procedure**
```

FIGURE 15.21 Top-down traversal for code generation.

15.5 Techniques Extending LR Parsers

The idea of LR-based techniques for table-driven code generation had been proposed earlier by Graham and Glanville [23]. However, their approach cannot be applied in general, to the problem of regular tree parsing for ambiguous tree grammars, because it does not carry forward all possible choices to be able to report all matches. The technique described here can be viewed as an extension of the *LR*(0) parsing strategy and is based on the work reported in Shankar et al. [41] and Madhavan et al. [34]. Let G' be the context-free grammar obtained by replacing all right-hand sides of productions of G by postorder listings of the corresponding trees in *TREES*$(A \cup N)$. Note that G is a regular tree grammar whose associated language contains trees, whereas G' is a context-free grammar whose language contains strings with symbols from A. Of course, these strings are just the linear encodings of trees.

Let *post*(t) denote the postorder listing of the nodes of a tree t. The following (rather obvious) claim underlies the algorithm:

> A tree t is in $L(G)$ if and only if *post*(t) is in $L(G')$. Also any tree α in *TREES*$(A \cup N)$ that has an associated S-derivation tree in G has an unique sentential form *post*(α) of G' associated with it.

The problem of finding matches at any node of a subject tree t is equivalent to that of parsing the string corresponding to the postorder listing of the nodes of t. Assuming a bottom-up parsing strategy is used, parsing corresponds to reducing the string to the start symbol, by a sequence of *shift* and *reduce* moves on the parsing stack, with a match of rule r reported at node j whenever r is used to reduce at the corresponding position in the string. Thus, in contrast with earlier methods that seek to construct a tree automaton to solve the problem, a deterministic pushdown automaton is constructed for the purpose.

15.5.1 Extension of the *LR*(0)-Parsing Algorithm

We assume that the reader is familiar with the notions of rightmost derivation sequences, handles, viable prefixes of right sentential forms and items valid for viable prefixes. Definitions may be found in [29]. The meaning of an *item* in this section corresponds to that understood in LR-parsing theory. By a viable prefix induced by an input string is the stack contents that result from processing the input string during an *LR*-parsing sequence. If the grammar is ambiguous, then several viable prefixes may be induced by an input string.

The key idea used in the algorithm is contained in the following theorem [41]:

Theorem 15.1. *Let G' be a normal form context-free grammar derived from a regular tree grammar. Then all viable prefixes induced by an input string are of the same length.*

To apply the algorithm to the problem of tree-pattern matching, the notion of matching is refined to one of matching in a left context.

Definition 15.7. *Let n be any node in a tree t. A subtree t_i is said to be to the left of node n in the tree, if the node m at which the subtree t_i is rooted occurs before n in a postorder listing of t. Subtree t_i is said to be a maximal subtree to the left of n if it is not a proper subtree of any subtree that is also to the left of n.*

Definition 15.8. *Let $G = (N, T, P, S)$ be a regular tree grammar in normal form, and t be a subject tree. Then rule $X \rightarrow \beta$ matches at node j in left context $\alpha, \alpha \in N^*$ if:*

1. *$X \rightarrow \beta$ matches at node j or equivalently, $X \Rightarrow \beta \Rightarrow^* t'$ where t' is the subtree rooted at j.*
2. *If α is not ϵ, then the sequence of maximal complete subtrees of t to the left of j, listed from left to right is $t_1, t_2, \ldots, t_k$, with t_i having an X_i-derivation tree, $1 \leq i \leq k$, where $\alpha = X_1X_2, \ldots, X_k$.*
3. *The string $X_1X_2, \ldots, X_kX$ is a prefix of the postorder listing of some tree in TREES$(A \cup N)$ with an S-derivation.*

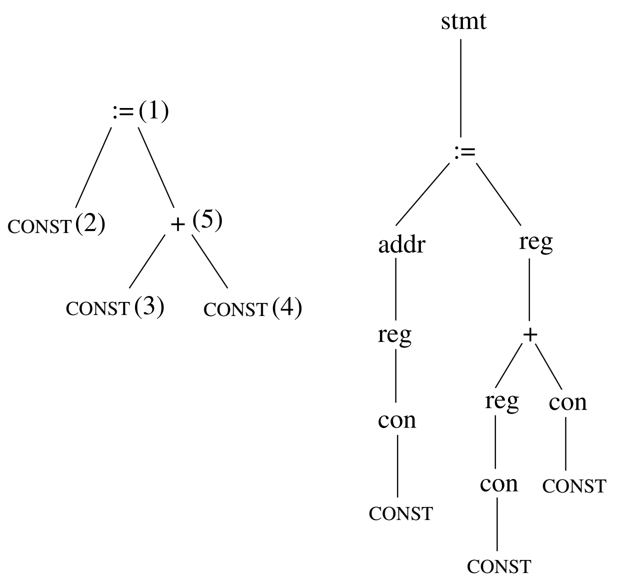


FIGURE 15.22 A derivation tree for a subject tree derived by the grammar of Example 15.8.

Example 15.8

We reproduce the tree grammar of Example 15.6 as a context-free grammar that follows:

1.	*stmt* $\rightarrow$ *addr reg* :=	[1]
2.	*addr* $\rightarrow$ *reg con* +	[0]
3.	*addr* $\rightarrow$ *reg*	[0]
4.	*reg* $\rightarrow$ *reg con*+	[1]
5.	*reg* $\rightarrow$ *con*	[1]
6.	*con* $\rightarrow$ *CONST*	[0]

Consider the subject tree of Figure **??** and the derivation tree alongside. The rule *con* $\rightarrow$ *CONST* matches at node 2 in left context ϵ. The rule *con* $\rightarrow$ *CONST* matches at node 3 in left context *addr*. The rule *reg* $\rightarrow$ *reg con* + matches at node 5 in left context *addr*.

The following property forms the basis of the algorithm. Let t a subject tree with postorder listing $a_1, \dots, a_j w, a_i \in A, w \in A^*$. Then rule $X \rightarrow \beta$ matches at node j in left context α if and only if there is a rightmost derivation in the grammar G' of the form:

$$S \Rightarrow^* \alpha X z \Rightarrow^* \alpha\, post(\beta) z \Rightarrow^* \alpha a_h, \dots, a_j z \Rightarrow^* a_1, \dots, a_j z, z \in A^*$$

where $a_h, \dots, a_j$ is the subtree rooted at node j.

Because a direct correspondence exists between obtaining rightmost derivation sequences in G' and finding matches of rules in G, the possibility of using an *LR*-like parsing strategy for tree parsing is obvious. Because all viable prefixes are of the same length a deterministic finite automaton (DFA) can be constructed that recognizes sets of viable prefixes. We call this device the *auxiliary automaton.* The grammar is first augmented with the production $Z \rightarrow S\$$ to make it prefix free. Next, the auxiliary automaton is constructed; this plays the role that a DFA for canonical set of LR items does in an *LR*-parsing process. We first explain how this automaton is constructed without costs. The automaton M is defined as follows:

$$M = (Q, \Sigma, \delta, q_0, F)$$

where each state of Q contains a set of items of the grammar, $\Sigma = A \cup 2^N$, $q_0 \in Q$ is the start state, F is the state containing the item $Z \longrightarrow S\$$ and $\delta : Q \times (A \cup 2^N) \mapsto Q$.

Transitions of the automaton are thus either on terminals or on sets of nonterminals. A set of nonterminals can label an edge iff all the nonterminals in the set match some subtree of a tree in the language generated by the regular tree grammar in the same left context. The precomputation of M is similar to the precomputation of the states of the DFA for canonical sets of $LR(0)$ items for a context-free grammar. However, there is one important difference. In the DFA for $LR(0)$ items, transitions on nonterminals are determined just by looking at the sets of items in any state. Here we have transitions on sets of nonterminals. These cannot be determined in advance because we do not know *a priori* which rules are matched simultaneously when matching is begun from a given state. Therefore, transitions on sets of nonterminals are added as and when these sets are determined. Informally, at each step, we compute the set of items generated by making a transition on some element of A. Because the grammar is in normal form, each such transition leads to a state, termed a *matchset* that calls for a reduction by one or more productions called *match_rules*. Because all productions corresponding to a given operator are of the same length (because operator arities are fixed and the grammar is in normal form), a reduction involves popping off a set of right-hand sides from the parsing stack, and making a transition on a set of nonterminals corresponding to the left-hand sides of all productions by which we have performed reductions, from each state (called an *LCset*) that can be exposed on stack after popping off the set of handles. This gives us, perhaps, a new state, which is then added to the collection if it is not present. Two tables encode the automaton. The first, δ_A, encodes the transitions on elements of A. Thus, it has, as row indices, the indices of the *LCsets*, and as columns, elements of A. The second, δ_{LC}, encodes the transitions of the automaton on sets of nonterminals. The rows are indexed by *LCsets*, and the columns by indices of sets of nonterminals. The operation of the matcher, which is effectively a tree parser, is defined in Figure 15.23. Clearly, the algorithm is linear in the size of the subject tree. It remains to describe the precomputation of the auxiliary automaton coded by the tables δ_A and δ_{LC}.

```
procedure TreeParser($a, M, matchpairs$)
// The input string of length $n + 1$ including the end marker is in array $a$
// M is the DFA (constructed from the context free grammar) which
controls the parsing process with transition functions $\delta_A$ and $\delta_{LC}$.
// matchpairs is a set of pairs $(i, m)$ such that the set of rules in $m$ matches
at node $i$ in a left context induced by the sequence of complete subtrees
to the left of $i$.
$stack = q_0$; $matchpairs = \emptyset$
$current\_state = q_0$
for $i = 1$ to $n$ do
    $current\_state := \delta_A(current\_state, a[i])$;
    $match\_rules = current\_state.match\_rules$
    // The entry in the table $\delta_A$ directly gives the set of rules matched.
    $pop(stack) arity(a[i]) + 1\ times$;
    $current\_state := \delta_{LC}(topstack, S_m)$;
    //$S_m$ is the set of nonterminals matched after chain rule application
    $match\_rules = match\_rules \cup current\_state.match\_rules$
    // add matching rules corresponding to chain rules that are matched
    $matchpairs = matchpairs \cup \{(i, match\_rules)\}$
    $push(current\_state)$
end for
end procedure
```

FIGURE 15.23 Procedure for tree parsing using bottom-up context-free parsing approach.

15.5.2 Precomputation of Tables

The start state of the auxiliary automaton contains the same set of items as would the start state of the DFA for sets of $LR(0)$ items. From each state, say q, identified to be a state of the auxiliary automaton, we find the state entered on a symbol of A, say a. (This depends only on the set of items in the first state). The second state, say m (which we refer to as a matchstate), may contain only complete items. We then set $\delta_A(q, a)$ to the pair $(match\_rules(m), S_m)$, where $match\_rules(m)$ is the set of rules that match at this point, and S_m is the set of left-hand side nonterminals of the associated productions of the context-free grammar. Next we determine all states that have paths of length $arity(a) + 1$ to q. We refer to such states as valid left context states for q. These are the states that can be exposed on stack while performing a reduction, after the handle is popped off the stack. If p is such a state, then we compute the state r corresponding to the itemset got by making transitions on elements of S_m augmented by all nonterminals that can be reduced to because of chain rules. These new item sets are computed using the usual rules that are used for computing sets of $LR(0)$ items. Finally, the closure operation on resulting items completes the new item set associated with r. The closure operation here is the conventional one used for constructing canonical sets of LR items [6].

Computing states that have paths of the appropriate length to a given state is expensive. A very good approximation is computed by the function *Validlc* in Figure 15.24. This function just examines the sets of items in a matchstate and a candidate left context state and decides whether the candidate is a valid left context state. For a matchstate m let $rhs(m)$ be the set of right-hand sides of productions corresponding to complete items in m.

For a matchstate m and a candidate left context state p, define:

$$NTSET(p, rhs(m)) = \{B \mid B \to .\alpha \in itemset(p), \alpha \in rhs(m)\}$$

Then a necessary, but not a sufficient condition for p to be a valid left context state for a matchstate corresponding to a matchset m is $NTSET(p, rhs(m)) = S_m$. (The condition is only necessary, because there may be another production that always matches in this left context when the others do, but which is not in the matchset.)

Before we describe the preprocessing algorithm, we have to define the costs that we can associate with items. The definitions are extensions of those used in Section 15.4 and involve keeping track of costs associated with rules partially matched (as that is what an item encodes) in addition to costs associated with rules fully matched.

DEFINITION 15.9. *The absolute cost of a nonterminal X matching an input symbol a in left context* ϵ *is represented by* $abscost(\epsilon, X, a)$. *For a derivation sequence d represented by* $X \Rightarrow X_1 \Rightarrow X_2, \ldots, \Rightarrow X_n \Rightarrow a$, *let* $(C_d = rulecost(X_n \to a) + \sum_{i=1}^{n-1} rulecost(X_i \to X_{i+1}) + rulecost(X \to X_1)$; *then* $abscost(\epsilon, X, a) = min_d(C_d)$.

```
function Validlc(p, m)
if NTSET(p, rhs(m)) = S_m then
    Validlc := true
else
    Validlc := false
end if
end function
```

FIGURE 15.24 Function to compute valid left contexts.

```
function $Goto(itemsets, a)$
$Goto = \{[A \to \alpha a., c] | [A \to \alpha .a, c'] \in itemset$ and
$c = c' + rule\_cost(A \to \alpha a) -$
$\min\{c'' + rule\_cost(B \to \beta a) | \; [B \to \beta .a, c''] \in itemset\}\}$
end function
```

FIGURE 15.25 The function to compute transitions on elements of A.

Definition 15.10. *The absolute cost of a nonterminal X matching a symbol a in left context* α *is defined as follows:*

$$abscost(\alpha, X, a) = abscost(\epsilon, X, a) \text{ if } X \text{ matches in left context } \alpha$$

$$abscost(\alpha, X, a) = \infty \text{ otherwise}$$

Definition 15.11. *The relative cost of a nonterminal X matching a symbol a in left context* α *is* $cost(\alpha, X, a) = abscost(\alpha, X, a) - min_{y \in N}\{abscost(\alpha, Y, a)\}$.

After defining costs for trees of height one we next look at trees of height greater than one. Let t be a tree of height greater than one.

Definition 15.12. *The cost* $abscost(\alpha, X, t) = \infty$ *if X does not match t in left context* α*. If X matches t in left context* α*, let* $t = a(t_1, t_2, \ldots, t_q)$ *and* $X \longrightarrow Y_1 Y_2, \ldots, Y_q a$ *where* Y_i *matches* $t_i, 1 \leq i \leq q$.

Let $abscost(\alpha, X \to Y_1 Y_2, \ldots, Y_q a, t) = rulecost(X \to Y_1, \ldots, Y_q a) + cost(\alpha, Y_1, t_1) + cost(\alpha Y_1, Y_2, t_2) + \cdots + cost(\alpha Y_1 Y_2, \ldots, Y_{q-1}, Y_q, t_q)$*. Hence define:*

$$abscost(\alpha, X, t) = min_{X \Rightarrow \beta \Rightarrow^* t}\{abscost(\alpha, X \Rightarrow \beta, t)\}$$

Definition 15.13. *The relative cost of a nonterminal X matching a tree t in left context* α *is* $cost(\alpha, X, t) = abscost(\alpha, X, t) - min_{Y \Rightarrow^* t}\{abscost(\alpha, Y, t)\}$.

We now proceed to define a few functions that can be used by the algorithm. The function *Goto* makes a transition from a state on a terminal symbol in A and computes normalized costs. Each such transition always reaches a match state because the grammar is in normal form.

The reduction operation on a set of complete augmented items $itemset_1$ with respect to another set of augmented items, $itemset_2$, is encoded in the function *Reduction* in Figure 15.26

The function *Closure* is displayed in Figure 15.27 and encodes the usual closure operation on sets of items. The function *ClosureReduction* is shown in Figure 15.28.

With these functions defined, we now present the routine for precomputation in Figure 15.29. The procedure *LRMain* produces the auxiliary automaton for states with cost information included in the items. Equivalence relations that can be used to compress tables are described in [34]. We next look at an example with cost precomputation. The context-free grammar obtained by transforming the grammar of Example 15.2 is displayed in Example 15.9.

Example 15.9

$$G = (V, B, G, a(2), b(0), P, V)$$

P:

$V \to$	VBa	[0]
$V \to$	GVa	[1]
$V \to$	G	[1]
$G \to$	B	[1]
$V \to$	b	[7]
$B \to$	b	[4]

The automaton is shown in Figure 15.30.

```
function $Reduction(itemset_2, itemset_1)$
// First compute costs of nonterminals in matchsets
$\mathcal{S} = \mathcal{S}_{itemset_1}$
$cost(X) = \min\{c_i | [X \rightarrow \alpha_i., c_i] \in itemset_1\}$ if $X \in \mathcal{S} \infty$ otherwise
// Process chain rules and obtain updated costs of nonterminals
$temp = \bigcup\{[A \rightarrow B., c] | \exists[A \rightarrow .B, 0] \in itemset_2 \wedge [B \rightarrow \gamma., c_1] \in$
$itemset_1 \wedge c = c_1 + rule\_cost(A \rightarrow B)\}$
repeat
    $\mathcal{S} = \mathcal{S} \cup \{X | [X \rightarrow Y., c] \in temp\}$
    for $X \in \mathcal{S}$ do
        $cost(X) = \min(cost(X), min\{c_i | \exists[X \rightarrow Y_i., c_i] \in temp\})$
        $temp = \{[A \rightarrow B., c] | \exists[A \rightarrow .B, 0] \in itemset_2 \wedge [B \rightarrow Y., c_1] \in$
        $temp \wedge c = c_1 + rule\_cost(A \rightarrow B)\}$
    end for
until no change to $cost$ array or $temp = \phi$
// Compute reduction
$Reduction = \bigcup\{[A \rightarrow \alpha B.\beta, c] | [A \rightarrow \alpha.B\beta, c_1] \in itemset_2 \wedge B \in \mathcal{S} \wedge$
$c = cost(B) + c_1$ if $\beta \not\Rightarrow \epsilon$ else
// This is a complete item corresponding to a chain rule
$c = rule\_cost(A \rightarrow B) - \min\{c_i | \exists[X \rightarrow .Y, 0] \in itemset_2, \wedge c_i =$
$rule\_cost(X \rightarrow Y)\}$
end function
```

FIGURE 15.26 Function that performs reduction by a set of rules given the LC state and the matchstate.

```
function $Closure(itemset)$
repeat
    $itemset = itemset \bigcup\{[A \rightarrow .\alpha, 0] | [B \rightarrow .A\beta, c] \in itemset\}$
until no change to itemset
$Closure = itemset$
end function
```

FIGURE 15.27 Function to compute the closure of a set of items.

```
function $ClosureReduction\ (itemset)$
$ClosureReduction = Closure(Reduction(itemset))$
end function
```

FIGURE 15.28 Function to compute *ClosureReduction* of a set of items.

Example 15.10
Let us look at a typical step in the preprocessing algorithm. Let the starting state be q_0, i.e., the first *LCset*.

$$q_0 = \{[S \rightarrow .V\$, 0], [V \rightarrow .\ V\ B\ a,\ 0], [V \rightarrow .\ G\ V\ a,\ 0], [V \rightarrow .\ G,\ 0], [G \rightarrow .\ B,\ 0], [V \rightarrow .\ b,\ 0], [B \rightarrow .\ b,\ 0]\}$$

By using the definition of the *Goto* operation, we can compute the matchset q_1 as:

$$q_1 = Goto(q_0, b) = \{[V \rightarrow b\ .,\ 3], [B \rightarrow b\ .,\ 0]\}$$

In the matchset, the set of matching nonterminals S_{q_1} is:

$$S_{q_1} = \{V,\ B\}$$

with costs 3 and 0, respectively.

```
procedure $LRMain()$
$lcsets := \emptyset$
$matchsets := \emptyset$
$list := Closure(\{[S \to .\alpha, 0] \mid S \to \alpha \in P\})$
while $list$ is not empty do
    delete next element $q$ from $list$ and add it to $lcsets$
    for each $a \in A$ such there is a transition on $a$ from $q$ do
        $m := Goto(q, a)$
        $\delta_A(q, a) := (match(m), S_m)$
        if $m$ is not in matchsets then
            $matchsets := matchsets \cup \{m\}$
            for each state $r$ in $lcsets$ do
                if $Validlc(r, m)$ then
                    $p := ClosureReduction(r, m)$
                    $\delta_{LC}(r, S_m) := (match(p), p)$
                    if $p$ is not in $list$ or $lcsets$ then
                        append $p$ to $list$
                    end if
                end if
            end for
        end if
    end for
    for each state $t$ in $matchsets$ do
        if $Validlc(q, t)$ then
            $s := ClosureReduction(q, t)$
            $\delta_{LC}(q, S_t) := (match(s), s)$
            if $s$ is not in $list$ or $lcsets$ then
                append $s$ to $list$
            end if
        end if
    end for
end while
end procedure
```

FIGURE 15.29 Algorithm to construct the auxiliary automaton.

Now, we can compute the set *ClosureReduction*(q_0, q_1). First, we compute *Reduction*(q_0, q_1):

Initialization

$\mathcal{S} = \mathcal{S}_{q_1} = \{V,\ B\}$
cost(V) $= 3$
cost(B) $= 0$
cost(G) $= \infty$
temp $= \{[G \to B\ ., 1]\}$

Processing chain rules

Iteration 1

$\mathcal{S} = \mathcal{S} \cup \{G\} = \{V,\ B,\ G\}$
cost(V) $= 3$
cost(B) $= 0$
cost(G) $= 1$
temp $= \{[V \to G\ ., 2]\}$

Iteration 2

$\mathcal{S} = \mathcal{S} \cup \{V\} = \{V,\ B,\ G\}$
cost(V) $= 2$
cost(B) $= 0$

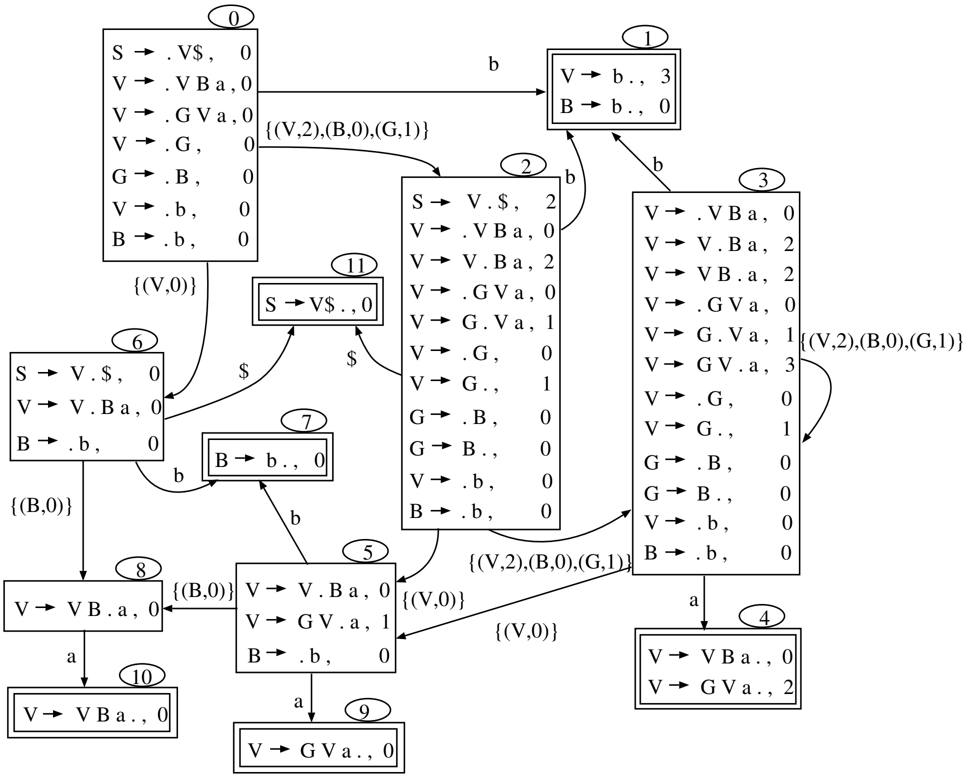


FIGURE 15.30 Auxiliary automaton for grammar of Example 15.9.

$cost(G) = 1$
$temp = \phi$

Computing reduction

Reduction $= \{[S \rightarrow V.\$, 2], [V \rightarrow V\,.\,B\,a,\ 2], [V \rightarrow G\,.\,V\,a,\ 1], [V \rightarrow G\,.,\ 1],$
$[G \rightarrow B\,.,\ 0]\}$

Once we have Reduction(q_0, q_1), we can use the function *Closure* and compute *ClosureReduction*. Therefore:

$$\begin{aligned}
q_2 &= \mathit{ClosureReduction}(q_0, q_1)\\
&= \mathit{Closure}(\mathit{Reduction}(q_0, q_1))\\
&= \mathit{Closure}(\{[S \rightarrow V.\$, 2], [V \rightarrow V\,.\,B\,a,\ 2], [V \rightarrow G\,.\,V\,a,\ 1],\\
&\quad [V \rightarrow G\,.,\ 1], [G \rightarrow B\,.,\ 0], \})\\
&= \{[S \rightarrow V.\$, 2], [V \rightarrow .\,V\,B\,a,\ 0], \{[V \rightarrow V\,.\,B\,a,\ 2], [V \rightarrow .\,G\,V\,a,\ 0],\\
&\quad [V \rightarrow G\,.\,V\,a,\ 1], [V \rightarrow .\,G,\ 0], [V \rightarrow G\,.,\ 1], [G \rightarrow .\,B,\ 0],\\
&\quad [G \rightarrow B\,.,\ 0], [V \rightarrow .\,b,\ 0], [B \rightarrow .\,b,\ 0]\}
\end{aligned}$$

The auxiliary automaton for the grammar of Example 15.6 is shown in Figure 15.31. Though the number of states for this example exceeds that for the conventional bottom-up, tree-pattern-matching automaton, it has been observed that for real machines, the tables tend to be smaller than those for conventional bottom-up, tree-pattern-matching automata [34]. This is perhaps because separate tables do not need to be maintained for each operator. An advantage of this scheme is that it allows the machinery of attribute grammars to be used along with the parsing.

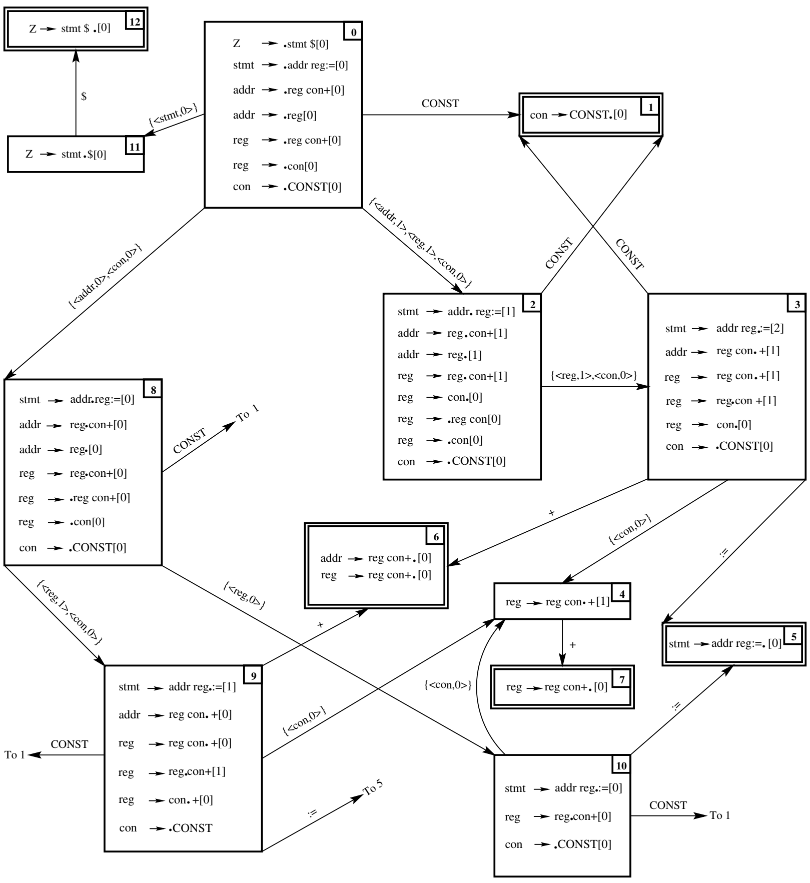


FIGURE 15.31 Auxiliary automaton for the grammar of Example 15.6.

15.6 Related Issues

A question that arises when generating a specification for a particular target architecture is the following: can a specification for a target machine produce code for every possible intermediate code tree produced by the front end? (We assume here, of course, that the front end generates a correct intermediate code tree.) This question has been addressed by Emmelmann [16], who refers to the property that is desired of the specification as the completeness property. The problem reduces to one of containment of the language $L(T)$ of all possible intermediate code trees in $L(G)$ the language of all possible trees generated by the regular tree grammar constituting the specification. Thus, the completeness test is the problem of testing the subset property of two regular tree grammars, which is decidable. An algorithm is given in [16].

A second question has to do with whether a code-generator generator designed to compute normalized costs statically terminates on a given input. If the grammar is such that relative costs diverge, the code-generator generator cannot halt. A sufficient condition for ensuring that code-generator generators based on extensions of LR-parsing techniques halt on an input specification is given in [34]. However, it is shown that there are specifications that can be handled by the tool but fail the test.

An important issue is the generation of code for a directed acyclic graph (DAG) where shared nodes represent common subexpressions. The selection of optimal code for DAGs has been shown to be intractable [5], but heuristics exist that can be employed to generate code [6]. The labeling phase of a bottom-up tree parser can be modified to work with DAGs. One possibility is that the code generator could, in the top-down phase, perform code generation in the normal way but count visits for each node. For the first visit it could evaluate the shared subtree into a register and keep track of the register assigned. On subsequent visits to the node it could reuse the value stored in the register. However, assigning common subexpressions to registers is not always a good solution, especially when addressing modes in registers provide free computations associated with offsets and immediate operands. One solution to this problem involves adding a DAG operator to the intermediate language [10].

15.7 Conclusion and Future Work

We have described various techniques for the generation of instruction selectors from specifications in the form of tree grammars. Top down, bottom up and LR-parser based techniques have been described in detail. Instruction selection using the techniques described in this chapter is useful and practical, but the formalism is not powerful enough to capture features like pipelines, clustered architectures and so on. Although it would be useful to have a single formalism for specification from which a complete code generator can be derived, no commonly accepted framework exists as yet. Instruction selection is important for complex instruction set computer (CISC) architectures, but for reduced instruction set computer (RISC) architectures there is a shift in the functional emphasis from code selection to instruction scheduling. In addition, because computation must be done in registers and because the ratio of memory access time to cycle time is high, some form of global register allocation is necessary. The interaction between instruction scheduling and register allocation is also important. Bradlee [11] has implemented a system that integrates instruction scheduling and global register allocation into a single tool. The advent of embedded processors with clustered architectures and very long instruction word (VLIW) instruction formats has added an extra dimension to the complexity of code-generator generators. The impact of compiler techniques on power consumption has been the subject of research only recently. Considerable work on retargetable compilers for embedded processors is already available, including MSSQ [35], RECORD [32], SPAM [8], CHESS [31, 38], CodeSyn [33] and AVIV [25]. Much more work is needed to address related problems on a sound theoretical basis.

Acknowledgments

Portions of this chapter were excerpted from M. Madhavan et al., *ACM Trans. Programming Language Syst.*, 22(6), 973–1001, 2000; P. Shankar et al. *Theor. Comput. Sci.*, 242, 125–142, 2000. With permission.

The author gratefully acknowledges the assistance of J. Surya Kumari, Balvinder Singh and V. Suresh in the preparation of the figures.

References

[1] A.V. Aho and M.J. Corasick, Efficient string matching: An aid to bibliographic search, *Commun. ACM*, 18(6), 333–340, 1975.

[2] A.V. Aho and M. Ganapathi, Efficient Tree Pattern Matching: An Aid to Code Generation, in Proceedings 12th ACM Symposium on Principles of Programming Languages, 1985, pp. 334–340.

[3] A.V. Aho, M. Ganapathi and S.W. Tjiang, Code generation using tree matching and dynamic programming, *ACM Trans. Programming Languages and Syst.*, 11(4), 491–516, 1989.

[4] A.V. Aho and S.C. Johnson, Optimal code generation for expression trees, *J. ACM*, 23(3), 146–160, 1976.

[5] A.V. Aho, S.C. Johnson and J.D. Ullman, Code generation for expressions with common subexpressions, *J. ACM*, 24(1), 21–28, 1977.

[6] A.V. Aho, R. Sethi and J.D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison Wesley, Reading, MA, 1986.

[7] A.W. Appel, Concise Specifications of Locally Optimal Code Generators, Technical report CS-TR-080-87, Department of Computer Science, Princeton University, Princeton, NJ, 1987.

[8] G. Araujo, S. Malik and M. Lee, Using Register Transfer Paths in Code Generation for Heterogeneous Memory-Register Architectures, in Proceedings of the 33rd Design Automation Conference (DAC), 1996.

[9] A. Balachandran, D.M. Dhamdhere and S. Biswas, Efficient retargetable code generation using bottom up tree pattern matching, *Comput. Languages*, 3(15), 127–140, 1990.

[10] J. Boyland and H. Emmelmann, Discussion: Code generator specification techniques, in *Code-Generation — Concepts, Tools, Techniques, Workshops in Computing Series*, R. Giegerich and S.L. Graham, Eds., Springer-Verlag, 1991, pp. 66–69.

[11] D.G. Bradlee, Retargetable Instruction Scheduling for Pipelined Processors, Ph.D. thesis, University of Washington, Seattle, WA, 1991.

[12] R.G.G. Cattell, Automatic derivation of code generators from machine descriptions, *ACM Trans. Programming Languages and Syst.*, 2(2), 173–190, 1980.

[13] D. Chase, An Improvement to Bottom up Tree Pattern Matching, in *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, 1987, pp. 168–177.

[14] T.W. Christopher, P.J. Hatcher and R.C. Kukuk, Using Dynamic Programming to Generate Optimised Code in a Graham-Glanville Style Code Generator, in Proceedings of the ACM SIGPLAN 1984 Symposium on Compiler Construction, 1984, pp. 25–36.

[15] H. Emmelmann, Code selection by regularly controlled term rewriting, in *Code-Generation-Concepts, Tools, Techniques, Workshops in Computing Series*, R. Giegerich and S.L. Graham, Eds., Springer-Verlag, 1991, pp. 3–29.

[16] H. Emmelmann, Testing completeness of code selector specifications, in *Proceedings of the 4th International Conference on Compiler Construction, CC '92, Lecture Notes in Computer Science*, Springer-Verlag, Vol. 641, 1992, pp. 163–175.

[17] H. Emmelmann, F. Schroer and R. Landwehr, BEG — A Generator for Efficient Back Ends, in Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation, SIGPLAN Notices, 1989, pp. 227–237.

[18] C. Ferdinand, H. Seidl and R. Wilhelm, Tree automata for code selection, *Acta Inf.*, 31, 741–760, 1994.

[19] C. Fraser, Automatic Generation of Code Generators, Ph.D. thesis, Yale University, New Haven, CT, 1977.

[20] C.W. Fraser, D.R. Hanson and T.A. Proebsting, Engineering a simple, efficient code-generator generator, *ACM Lett. Programming Languages and Syst.*, 1(3), 213–226, 1992.

[21] M. Ganapathi, C.N. Fischer and J.L. Hennessy, Retargetable compiler code generation, *Comput. Surv.*, 14(4), 1982.

[22] M. Ganapathi and C.W. Fischer, Affix grammar driven code generation, *ACM Trans. Programming Languages Syst.*, 7(4), 560–599, 1985.

[23] S.L. Graham and R.S. Glanville, A New Method for Compiler Code Generation, in Proceedings of the 5th ACM Symposium on Principles of Programming Languages, 1978, pp. 231–240.

[24] S.L. Graham, R. Henry and R.A. Schulman, An Experiment in Table Driven Code Generation, in Proceedings of the SIGPLAN '82 Symposium on Compiler Construction, SIGPLAN Notices, 1982.

[25] S. Hanono and S. Devadas, Instruction Selection, Resource Allocation and Scheduling in the AVIV Retargetable Code Generator, in Proceedings of the 35th Design Automation Conference (DAC), 1998.

[26] P. Hatcher and T. Christopher, High-Quality Code Generation via Bottom-up Tree Pattern Matching, in Proceedings of the 13th ACM Symp. on Principles of Programming Languages, 1986, pp. 119–130.

[27] R.R. Henry and P.C. Damron, Performance of Table Driven Generators Using Tree Pattern Matching, Technical report 89-02-02, Computer Science Department, University of Washington, Seattle, WA, 1989.

[28] C. Hoffman and M.J. O'Donnell, Pattern matching in trees, *J. ACM*, 29(1), 68–95, 1982.

[29] J.E. Hopcroft and J.D. Ullman, *An Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, Reading, MA, 1979.

[30] R. Landwehr, H.-St. Jansohn and G. Goos, Experience with an Automatic Code Generator Generator, in Proceedings of the SIGPLAN '82 Symposium on Compiler Construction, SIGPLAN Notices, 1982.

[31] D. Lanneer, J. Van Praet, A. Kifli, K. Schoofs, W. Geurts, F. Thoen and G. Goossens. CHESS: Retargetable code generation for embedded DSP processors, in Chap. 5. *Code Generation for Embedded Processors*, P. Marwedel and G. Goosens, Eds., 1995.

[32] R. Leupers and P. Marwedel, Retargetable Generation of Code Selectors from HDL Processor Models, in Proceedings of the European Design and Test Conference (ED & TC), 1997.

[33] C. Liem, *Retargetable Compilers for Embedded Core Processors*, Kluwer Academic, Dordrecht, 1997.

[34] M. Madhavan, P. Shankar, S. Rai and U. Ramakrishna, Extending Graham-Glanville techniques for optimal code generation, *ACM Trans. Programming Languages Syst.*, 22(6), 973–1001, 2000.

[35] P. Marwedel, Tree-Based Mapping of Algorithms to Predefined Structures, in Proceedings of the International Conference on Computer-Aided Design (ICCAD), 1993.

[36] A. Nymeyer and J.P. Katoen, Code generation based on formal BURS theory and heuristic search, *Acta Inf.*, 34(8), 597–636, 1997.

[37] E. Pelegri-Llopart and S.L. Graham, Optimal Code Generation for Expression Trees, in Proceedings of the 15th ACM Symposium on Principles of Programming Languages, 1988, pp. 119–129.

[38] J. Van Praet, D. Lanneer, W. Geurts and G. Goossens, Processor modeling and code selection for retargetable compilation, *ACM Trans. Design Automation Digital Syst.*, 6(3), 277–307, 2001.

[39] T.A. Proebsting, BURS automata generation, *ACM Trans. Programming Languages Syst.*, 17(3), 461–486, 1995.

[40] T.A. Proebsting and B.R. Whaley, One-Pass Optimal Tree Parsing-with or without Trees, in International Conference on Compiler Construction, 1996, pp. 294–308.

[41] P. Shankar, A. Gantait, A.R. Yuvaraj and M. Madhavan, A new algorithm for linear regular tree pattern matching, *Theor. Comput. Sci.*, 242, 125–142, 2000.

[42] S.W.K. Tjiang, Twig Reference Manual, Technical Report Computing Science Technical report 120, AT&T Bell Labs, Murray Hills, NJ, 1985.

[43] S.W.K. Tjiang, An Olive Twig, Technical report, Synopsis Inc., 1993.

[44] B. Weisgerber and R. Willhelm, Two tree pattern matchers for code selection, in *Compiler Compilers and High Speed Compilation, Lecture Notes in Computer Science* Vol. 371, Springer-Verlag, 1988, pp. 215–229.

[45] R. Wilhelm and D. Maurer, *Compiler Design*, International Computer Science Series, Addison-Wesley, Reading, MA, 1995.

16

Retargetable Very Long Instruction Word Compiler Framework for Digital Signal Processors

Subramanian Rajagopalan
Princeton University

Sharad Malik
Princeton University

16.1 Introduction

Digital signal processors (DSP) are used in a wide variety of embedded systems ranging from safety critical flight navigation systems to common electronic items such as cameras, printers and cellular phones. DSPs are also some of the more popular processing element cores available in the market today for use in system-on-a-chip (SOC)-based design methodology for embedded processors. These systems not only have to meet the real-time constraints and power consumption requirements of the application domain, but also need to adapt to the fast-changing applications for which they are used. Thus, it is very important that the target processor be well matched to the particular application to meet the design goals. This in turn requires that DSP compilers need to produce good quality code and be highly retargetable to enable a system designer to quickly evaluate different architectures for the application on hand.

Unlike their general-purpose counterparts, an important requirement for embedded system software is that it has to be sufficiently dense so as to fit within the limited quantity of silicon area, either random-access memory (RAM) or read-only memory (ROM), dedicated to program memory

0-8493-1240-7/03/$0.00+$1.50

on the chip. This requirement arises because of the limited and expensive on-chip program memory. To achieve this goal, and at the same time not sacrifice dynamic performance, DSPs have often been designed with special architectural features such as address generation units, special addressing modes useful for certain applications, special computation units, accumulator-based data paths and multiple memory banks. Hence, to produce good quality code, DSP compilers must also incorporate a large set of optimizations to support and exploit these special architectural features.

In this chapter, we take a look at DSP architectures from a compiler developer's view. The organization of this chapter is as follows. The different types of DSP architectures are detailed in Section 16.2. Section 16.3 provides an overview of the different constraints imposed by DSP architectures using an illustrative example. The need for retargetable methodologies and those proposed thus far are discussed in Section 16.4. Some of the retargetable code generation and optimization techniques that have been developed to exploit the special DSP features are discussed in Section 16.5. Finally, summary and conclusions are provided in Section 16.6.

16.2 Digital Signal Processor Architectures

A close examination of DSP architectures and the requirements for a DSP compiler suggests that DSPs can be modeled as very long instruction word (VLIW) processors. A VLIW processor is a fully statically scheduled processor capable of issuing multiple operations per cycle. Figure 16.1 shows a simplified overview of the organization of instruction set architectures (ISAs) of VLIW processors.

The ISA of a VLIW processor comprises multiple instruction templates that define the set of operations that can be issued in parallel. Each slot in an instruction can be filled with one operation from a set of possible operations. Each operation in turn consists of an opcode that defines the operation's resource usage when it executes, and a set of operands each of which can be a register, a memory address or an immediate operand. The register operand in an operation can be a single machine word register, a part of a machine word register or a set of registers. Although the memory address is shown as a single operand, in general, the address itself can be multiple operands depending on the addressing mode used in the operation. Compilers for VLIW architectures also need optimizations to exploit the application's instruction level parallelism (ILP) to primarily obtain good dynamic performance and this is often achieved at the expense of increased static code size. For DSPs, however, static code size is equally or even more important than the dynamic performance.

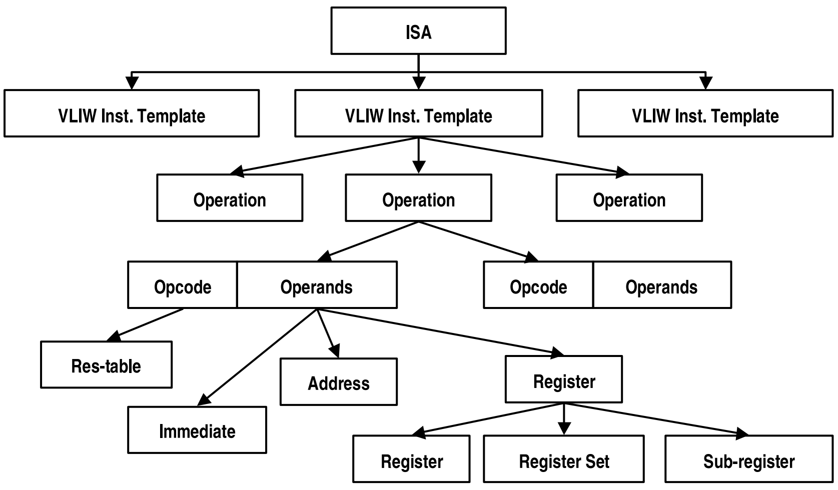


FIGURE 16.1 VLIW instruction set architecture.

Because both VLIW processors and DSPs rely more on compilers and less on hardware for performance and VLIW compilation is well known, the limitations of DSPs and the constraints posed by them for the compiler can be better understood by modeling the DSP architectures as VLIW architectures. Some specific examples of the architectural constraints in DSPs are presented in Section 16.3.

This section gives an overview of the features of different classes of DSP architectures from a compiler perspective. The basic architecture of DSPs is a Harvard architecture with separate data and program memories. Some of the architectural features of DSPs are motivated by the applications:

- Because DSP applications often work with multiple arrays of data, DSPs feature multiple data memory banks to extract ILP from memory operations.
- Because vector products and convolutions are common computations in signal processing, a fast single cycle multiply and multiply-accumulate based data paths are common.
- Multiple addressing modes, including special modes such as circular addressing, are used to optimize variable access.
- Fixed or floating point architectures are used depending on the application's requirements such as precision, cost and scope.
- Hardware to support zero overhead loops is present for efficient stream-based iterative computation.

For the discussion in the rest of this chapter, we define a VLIW instruction in an ISA as a set of operations allowed by the ISA to be issued in parallel in a single cycle. ISAs of DSPs can be classified based on the following attributes:

- *Fixed operation width ISA or a variable operation width ISA.* A fixed operation width ISA is an ISA in which all the operations are encoded using a constant number of bits. A variable operation width ISA is any ISA that does not use a fixed operation width for all operations.
- *Fixed operation issue ISA or variable operation issue ISA.* A fixed operation issue ISA is an ISA in which the number of operations issued in each cycle remains a constant. A variable operation issue ISA can issue a varying number of operations in each cycle.
- *Fixed instruction width ISA or a variable instruction width ISA.* A fixed instruction width ISA is an ISA in which the number of bits used to encode all the VLIW instructions is a constant. A variable instruction width ISA is any ISA whose instruction encoding size is not a constant.

Of these three attributes, operation width and the instruction width are more important because the issue width of a DSP is usually dependent on these two attributes. Hence, programmable DSP architectures can be broadly classified into four categories as shown in Figure 16.2, namely:

1. ISAs with fixed operation width (FOW) and fixed instruction width, Figure 16.2(a)
2. ISAs with FOW and variable instruction width, Figure 16.2(b)
3. ISAs with variable operation width (VOW) and fixed instruction width, Figure 16.2(c), and
4. ISAs with VOW and variable instruction width, Figure 16.2(d)

Because the architectural features of the first two categories of DSPs are more due to the FOW attribute, we combine the two categories into a single FOW category. Similarly, because compiling for architectures in the last category is similar to VOW and fixed instruction width architectures, we assume VOW architectures to include the last two categories.

16.2.1 Fixed Operation Width Digital Signal Processor Architectures

The ISA of an FOW architecture consists of reduced instruction set computer (RISC) style operations, all of which are encoded using a constant number of bits, typically 16 or 32 b. From Figure 16.2(a)

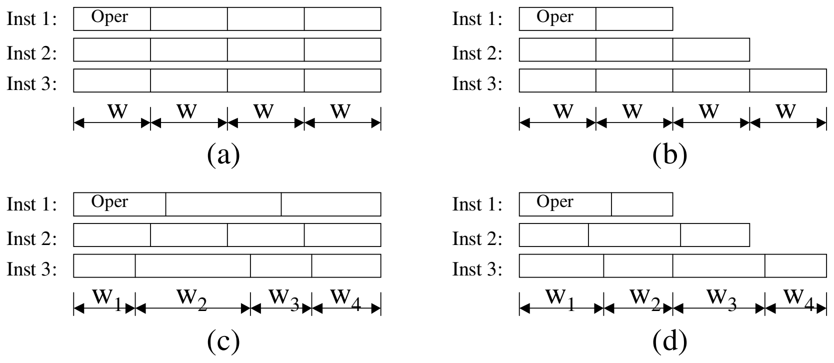


FIGURE 16.2 Different classes of DSP architectures: (a) fixed operation width and fixed instruction width; (b) fixed operation width and variable instruction width; (c) variable operation width and fixed instruction width; (d) variable operation width and variable instruction width.

and 16.2(b), we can see that the issue width and instruction width of these processors are tied to one another. As the issue width is increased to get more ILP, the instruction width to be decoded also increases proportionally. Examples in this category include the Texas Instruments (TI) TIC6X series DSPs. Most of the high-end DSPs, and, in particular, most floating point DSP architectures also fall into this category. Some of the features of these architectures are described in this section:

- These architectures are regular with less instruction level constraints on operands and operations when compared with the other categories of DSPs. The primary reason for this regularity is the encoding of the instructions. Not only is the encoding of each operation in an instruction independent but also the encoding of opcode and operands within an operation are also separate.
- A large general-purpose register set, which is a key to exploiting ILP, is an important feature of these architectures. This set may even be organized into multiple register files.
- The regularity and the large register set together make these DSPs more compiler friendly and easier to program than their VOW counterparts. This also stems from the fact that these architectures tend to have an orthogonal set of operands, operations and instructions.
- Due to the general nature of operations and increased ILP, these processors are capable of handling a wider variety of applications and the applications themselves can be large. Specialized functional units are sometimes used to improve a group of applications.
- To extract more ILP, there is a trend in these architectures to support speculation and predication, which are features in general-purpose processors. Because a discussion of speculation and predication is beyond the scope of this chapter, we refer the readers to [6, 42].
- One of the drawbacks of fixed instruction-width FOW DSPs is that of code size. For every VLIW slot in an instruction that the compiler is not able to fill with a useful operation, no-operations (NO-OPs) need to be filled in. This often leads to an increase in code size because the instruction width remains constant. To overcome this problem, some processors allow a compiler to signal the end of an instruction in each operation by turning on a flag in the last operation of an instruction. This prevents any unnecessary NO-OPs from being inserted and it is the responsibility of the hardware to separate the instructions before or during issue. With this feature enabled, fixed instruction width FOW processors transform into variable instruction width FOW processors.

Because the features of these processors are very similar to VLIW processors, the compilers of these processors mostly leverage the work done in VLIW compilation. Thus, the DSP compiler techniques and optimizations described in this chapter may not be applicable or useful to most processors in this category.

16.2.2 Variable Operation Width Digital Signal Processor Architectures

The features of VOW architectures shown in Figures 16.2(c) and 16.2(d) are in sharp contrast to those of FOW-style architectures. These DSPs are used in applications such as cellular telephones. Hence, the binary code size of applications running on these processors needs to be extremely dense because cost of on-chip program memory increases nonlinearly with code size. Also, a conflicting requirement of good dynamic performance exists because these applications usually run on battery. Most of the features of VOW DSPs described in this chapter reflect the decisions made by hardware architects to meet these conflicting requirements. Examples in this category include Fujitsu Hiperion, TI TMS320C25, Motorola DSP56000, etc.

- To contain the code size of applications, but still exploit ILP, VOW DSP instructions are encoded such that the instruction width does not change with issue width. This leads to a situation where all the operations in an instruction are encoded together and operands within an operation are also tied together in an encoding that leads to irregular architectures with restricted ILP support.
- These DSPs feature only a small number of registers with many constraints on how each register can and cannot be used.
- The irregular architecture and the heterogeneous register set pose a challenge to DSP compilers that have to tackle the numerous constraints and yet meet the conflicting requirements. This is primarily due to the irregular operation and instruction encoding in VOW DSPs. This leads to nonorthogonal operands within operations, nonorthogonal operations within instructions and a nonorthogonal instruction set.
- The DSPs in this class have traditionally been programmed in assembly. This drastically restricts the domain of applications to small programs that are usually kernels of larger applications that can be accelerated using DSPs.
- To compensate for the lack of a rich register set and to utilize the data memory banks effectively, these DSPs have an address register set and an address generation unit that enable address computations to be performed in parallel with other operations.

Section 16.3 discusses how some of these features affect the development of compilers for DSPs. For a more detailed discussion on DSP architectures, we refer the readers to *DSP Processor Fundamentals* [23].

16.3 Compilation for Digital Signal Processors

In this section, we use examples to describe some of the constraints posed by DSP architectures to compilers due to the features of VOW DSPs described in Section 16.2.2.

Some of the common DSP ISA constraints are shown in 16.3. The constraints between the different entries are represented in dotted lines. Figure 16.3(a) shows operation ILP constraints in the instruction template that restrict the set of operations that can be performed in parallel. Figure 16.3(b) shows operand constraints that exist between operands within an operation. This can be of different types such as register-register constraints or register-addressing mode or register-immediate size constraints and so on. Finally, Figure 16.3(c) extends the operand constraints across operations within an instruction.

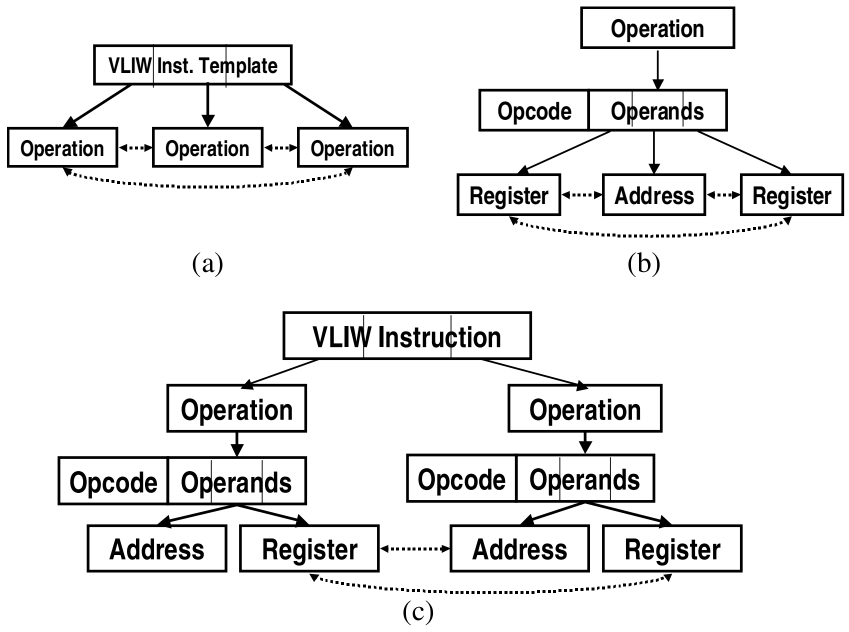
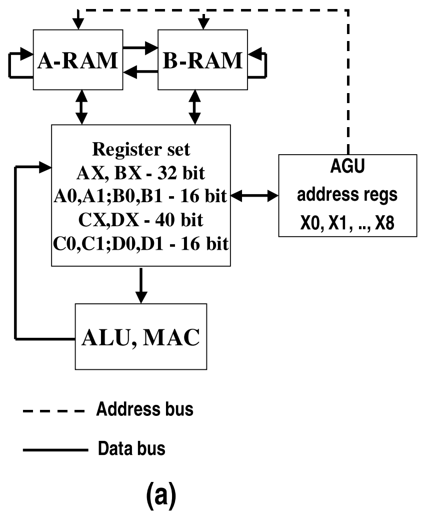


FIGURE 16.3 DSP constraints.



Restricted ILP

ALU & 1 MEMop
SFT & 1 MEMop
MAC & 1 MEMop
MAC & 2 LOADs
2 LOADs
1 LOAD & 1 STORE
MEMops & AGU

(b)

FIGURE 16.4 (a) A DSP datapath; (b) some operation ILP constraints.

The DSP data path shown in Figure 16.4(a) is a fixed point DSP core. It is a fixed instruction width VOW DSP. Its data path consists of dual memory banks, an arithmetic and logic unit (ALU) comprising a shift unit, an add unit, a multiply unit and an address generation unit. There are 8 address registers (ARs) and 4 accumulators each with 32 b, each of which can also be accessed as 2 general-purpose registers, each with 16 b. The two memory banks allow two memory access operations to be issued in parallel. In addition to instructions with parallel memory access operations, restricted ILP also exists between arithmetic and memory operations. However, this ILP requires several conditions to be satisfied due to the small fixed instruction width. Some of the constraints are briefly described and explained with examples, as follows:

- Because addressing modes occupy more instruction bits, many restrictions exist on the addressing mode that a memory operation can use in different instructions. This includes the set of ARs that can be used by a particular addressing mode in a particular instruction.
- The fixed instruction width and the VOW together enforce several constraints on what set of operations can be performed in parallel as explained in Section 16.3.1.
- Operations in a single instruction share operands due to lack of bits to encode the operations independently. This may or may not affect data flow as explained in Section 16.3.2.
- Operations in a single instruction have to split an already small number of instruction bits to encode their operands leading to register allocation constraints that arise not due to lack of registers, but lack of bits to encode more registers, as pointed out in the second example in Section 16.3.2.
- Large immediate operands require multiple instruction words to encode them leading to larger code size and a potentially extra operation execution time.

The constraints described in this section are representative of constraints found in most commercial DSPs and are difficult to capture in both behavior description-based machine descriptions (MDs) (MDs based on ISA descriptions) and structural description-based MDs (MDs that extract ISA information from description of a processor data path). It is also important to note that the internal data paths of a processor are seldom transparent to the application developer; thus, the latter may not even be an option.

16.3.1 Operation Instruction Level Parallelism Constraints

These constraints describe what set of operations can and cannot be issued in parallel. Figure 16.4(b) shows the set of ILP constraints. *MEMop* stands for either a *LOAD* or a *STORE* operation, *ALU* stands for operations that use the ALU such as *ADD*, *SUB* and *Shift* operations and *MAC* stands for all operations associated with the MAC unit such as *Multiply* and *Multiply-Accumulate*. Although two *LOAD* operations can be issued in parallel with a *MAC* operation, only one *LOAD* can be issued in parallel with an *ALU* operation. Hence, these constraints do not need to be limited by physical resources alone.

16.3.2 Operand Instruction Level Parallelism Constraints

These constraints describe how registers should be assigned to operands of operations issued in parallel. These constraints may or may not affect data flow.

Example 16.1: Constraint Affecting Data Flow
Constraint *MUL dest, source*1, *source*2; *LOAD dest*1, [*M*1] ; *LOAD dest*2, [*M*2] is a valid single-cycle schedule only if *dest*1 and *dest*2 are assigned the same registers as *source*1 and *source*2. It is the responsibility of the compiler to ensure that the two sources of the multiply operation are not used after this instruction. Hence, these constraints are data flow affecting constraints. In addition, there can also be restrictions on what registers can be assigned to each operand in an operation or instruction.

Example 16.2: Constraint Not Affecting Data Flow
Constraint: *ADD dest*1, source1, *source*2; *LOAD dest*2, $[M]$ is a valid single cycle schedule only if the following conditions are satisfied. If *dest*1 is assigned the register *CX*, then:

1. Condition: *dest*2 can be assigned a register only from the following set of registers $\{A0, A1, B0, B1, D0, D1, CX, DX\}$.
2. Condition *source*1 can be assigned only one of $\{CX, A0\}$.
3. Condition *source*2 must be assigned register $A1$.

Although the first constraint is an example of operand constraints across operations, the other two are examples of operand constraints within an operation.

Most of these constraints can be attributed to the nonorthogonality property of VOW DSP architectures. Apart from the lack of orthogonality in operand, operation and instruction encodings, this also includes difficulty in classifying arithmetic operations into similar classes, for example, an *ADD* and *SUB* may have completely different sets of constraints. Because the same operation has different constraints under different instances, we define each such instance as an *operation version* and the problem of picking the right version for an operation as the *operation versioning problem.*

16.4 Retargetable Compilation

Section 16.2 described the architectural features of DSPs that were designed by hardware architects with the intent of meeting application demands and Section 16.3 described the constraints posed by the architectures for compiler developers that are in addition to traditional compiler issues. From these two discussions, we can see that the compiler developers need a clean abstraction of the architecture that allows them to develop reusable or synthesizable compilers that do not need to know the actual architecture they are compiling for, and the hardware architects need a configurable abstraction of the compiler that allows them to quickly evaluate the architectural feature that they have designed without much knowledge of how the feature is supported by the compiler. Traditionally, the compiler has been split into two phases, namely, the front end that converts the application program into a semantically equivalent common intermediate representation and the back end that takes the intermediate representation as input and emits the assembly code for the given target. This partially eases the job of the compiler developer and the hardware designer of having to match the application to the target.

However, this coarse grain reusability is not sufficient, because the growing number of applications and the increasing number of processor designs both require an efficient fine grain reusability of components that is often referred to as retargetability. This is shown in Figure 16.5 where a retargetable interface captures the hardware artifacts, exports them to the compiler developer, captures the ability to parameterize the compiler (masking the algorithms, complexity, etc.) and exports it to the architecture designer. In this design, each phase of the compiler may be individually configurable or each phase itself may be divided into configurable components as shown by dotted lines within each phase in Figure 16.5. An interesting side effect of such a retargetable design is that the abstraction does not require the user to be an expert in both architectures and compilers.

For an SOC design, an architect may be faced with many options for the cores that meet the demands of the application on hand. The designer then needs the software tool set such as a compiler and a simulator to quickly evaluate the set of DSP cores available, pick the optimum core and reduce the time to market the design. This set of DSP cores includes programmable cores that may still have to be developed for a set of applications, therefore also requiring the development of the necessary

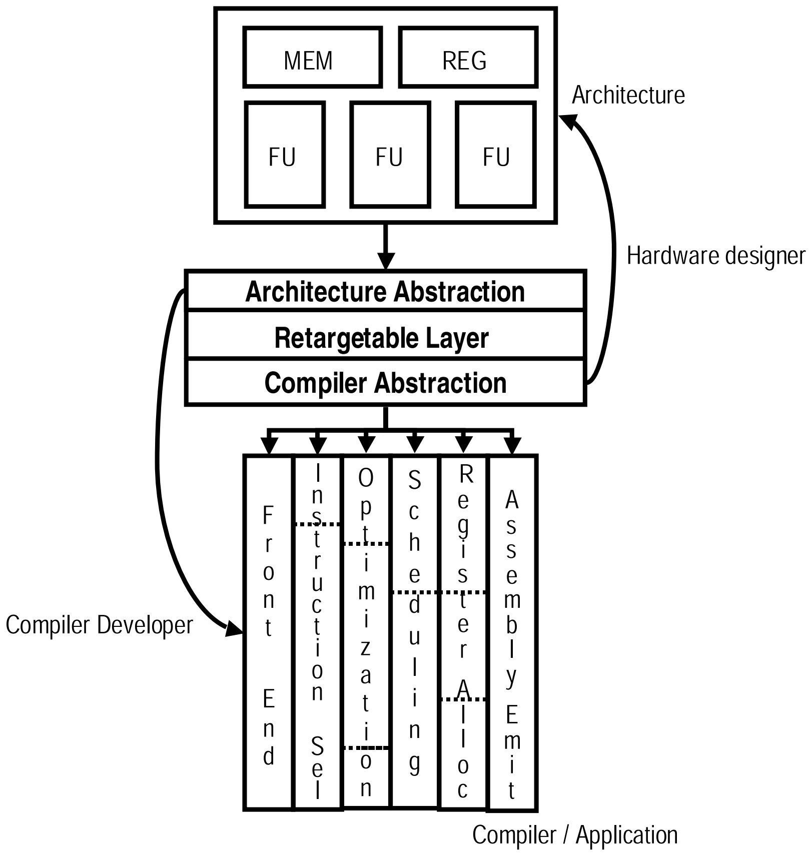


FIGURE 16.5 Retargetable interface.

tool set; and off-the-shelf processor cores that are readily available with the tool set. Although in the second case, the architectures are predefined and the necessary compilers for the specific processors may be readily available, in the former, the architecture itself is not defined and it is not practically possible to develop an architecture-specific compiler for every potential architecture solution. Hence, clearly a need exists to design the compiler framework so that it is easy either to synthesize a new compiler for every potential architecture or to reuse parts of the compiler to the maximum extent possible by minimizing the amount of architecture-specific components. Based on this need for retargetability, different types of retargetability have been defined with respect to the extent of reuse [53].

16.4.1.1 Automatic Retargetability

An automatically retargetable compiler framework has built-in support for all potential architectures that meet the demands of the applications on hand. Compilers for specific architectures can then be generated by configuring a set of parameters in the framework. Although this is an attractive solution for retargetability, it is limited in scope to a regular set of parameterizable architectures and is not capable of supporting a wide variety of specialized architectural features found in DSPs.

16.4.1.2 User Retargetability

A user retargetable compiler framework relies on an architectural description to synthesize a compiler for a specific architecture. Although support exists for a wide variety of machine-independent optimizations, this type of framework also suffers from the drawback of not having the ability to automatically generate machine-dependent optimizations. This level of retargetability has been achieved only in the instruction selection phase of the back ends. Some examples in this category include Twig [1] and Iburg [14], both of which automatically generate instruction selectors from a given specification.

16.4.1.3 Developer Retargetability

A developer is an experienced compiler designer and a developer retargetable framework is based on a library of parameterized optimizations for architectural features. A compiler for a specific architecture can be synthesized in this framework by stringing together the set of relevant optimizations from the developer's library with the correct set of parameters. If the library lacks an optimization that can be applied to a certain new architectural feature, then the developer is responsible for developing a parameterized optimization module, the associated procedure interface and the insertion of it into the library. The new optimization module can then be used for designs in the future that have the new architectural feature. The success of this framework lies in the extent of code reuse or optimization module reuse. Hence, this type of retargetability is best suited for evaluation of architectures in a single family.

Figure 16.6 shows a potential retargetable compiler development methodology that can be used to design DSP compilers. The methodology starts either by hand-coding DSP kernels in assembly, or, if possible, by using a simple machine specific compiler that produces correct code with no optimizations. For example, this compiler can be obtained from a simple modification of a compiler developed for a closely related processor, or it can be a first cut of the compiler or one of the publicly available retargetable research compilers that fits the domain of the applications. The developer then examines the assembly code for inefficiencies and suggests optimizations for the various regular and irregular parts of the DSP architecture. This includes efficient capture of the processor ISA and microarchitecture using machine descriptions and general-purpose performance optimizations for the regular parts. For the special architectural features, a parameterized optimization library is developed and the optimizations are added to this library as and when they are developed. An optimizing compiler is then assembled together using the different components and the cycle is repeated until satisfactory code quality is achieved. Conventionally, retargetability has always been achieved with some kind of a MD that describes the processor ISA and/or structure. This database is then queried by the different phases of compilation to generate code specific to the target processor.

The success of a retargetable methodology can be evaluated or measured based on several qualities such as *nature*, parameterizable or synthesizable components; *modularity*, fine or coarse grain capture of hardware artifacts; *accuracy*, accurate modeling of architecture; *efficiency*, quality of code generated for a set of processors, time taken to produce the code in the first place; *visibility*, what part and how much of the compiler is visible to the user; *usability*, how easily one can use the methodology; *extent*, the processor space that is covered; *extensibility*, ability to add new modules or optimizations and ability to extend the framework to other domains; *independence*, what fraction of the code is compiler specific and what fraction is architecture domain specific. These are not all independent or arranged in any order of preference.

16.5 Retargetable Digital Signal Processor Compiler Framework

In this section, we describe the flow of a retargetable DSP compiler for an architecture with various constraints mentioned in Section 16.3. The DSP code generator flow is shown in Figure 16.7.

Because the compiler flow shown in Figure 16.7 is very similar to a general-purpose compiler flow, we address only the issue of DSP-specific optimizations in this section. In general, the wide variety

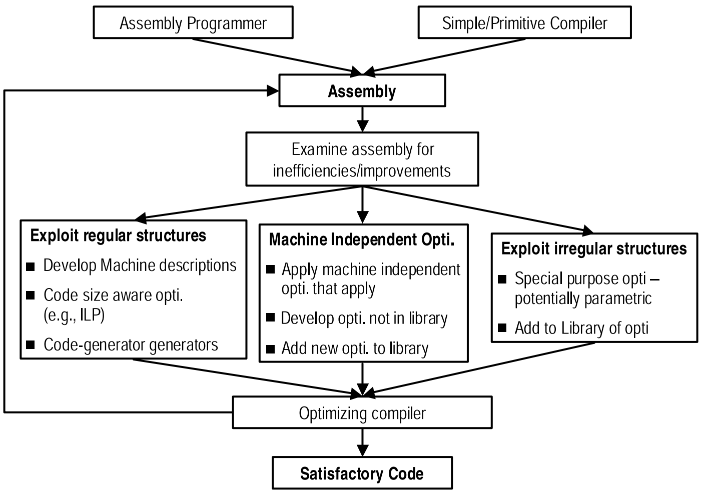

Assembly Programmer
Simple/Primitive Compiler
Assembly
Examine assembly for inefficiencies/improvements
Exploit regular structures
Develop Machine descriptions
Code size aware opti. (e.g., ILP)
Code-generator generators
Machine Independent Opti.
Apply machine independent opti. that apply
Develop opti. not in library
Add new opti. to library
Exploit irregular structures
Special purpose opti – potentially parametric
Add to Library of opti
Optimizing compiler
Satisfactory Code

FIGURE 16.6 Retargetable compiler methodology.

of machine-independent compiler optimizations used in compilers for general-purpose processors can also be applied to DSP compilers [2]. However, certain precautions need to be taken before directly applying an optimization due to the nature of the DSP architectures. For example, in most VOW-style DSPs, an extra instruction word is needed to store large constants if the operation uses one. Because this may also cause a one-cycle fetch penalty, constant propagation[1] must be applied selectively. Other examples that can potentially have a negative effect include common subexpression elimination[2] and copy propagation[3]. These sets of optimizations tend to increase the register pressure by extending live ranges that can lead to negative results in architectures with small register sets. However, some of the most beneficial optimizations for DSPs are the loop optimizations such as loop invariant code motion,[4] induction variable elimination,[5] strength reduction,[6] etc. Optimizations such as loop unrolling[7] increase static code size and some loop optimizations also tend to increase register pressure, but the trade-off in either case is the gain in reduction of execution time. Most DSP applications involve some kind of kernel loops that account for the majority of their execution time. Hence, optimizing loops is an important task in a successful DSP compiler framework. However, optimizations such as loop induction variable elimination can prevent efficient utilization of zero overhead looping features in DSPs.

[1]If a MOV operation occurs that moves a constant c to a virtual register u, then the uses of u may be replaced by c.

[2]If a program computes an expression more than once with exactly the same sources, it may be possible to remove redundant expressions.

[3]If a MOV operation occurs that moves one virtual register v to another virtual register u, then the uses of u may be replaced by use of v.

[4]Loop invariant code motion is moving loop invariant computations outside the loop.

[5]Induction variable elimination is removing the loop induction variable from inside the loop.

[6]Strength reduction is replacing expensive computations such as multiply and divide with simpler operations such as shift and add.

[7]Loop rolling is placing two or more copies of the loop body in a row to improve efficiency.

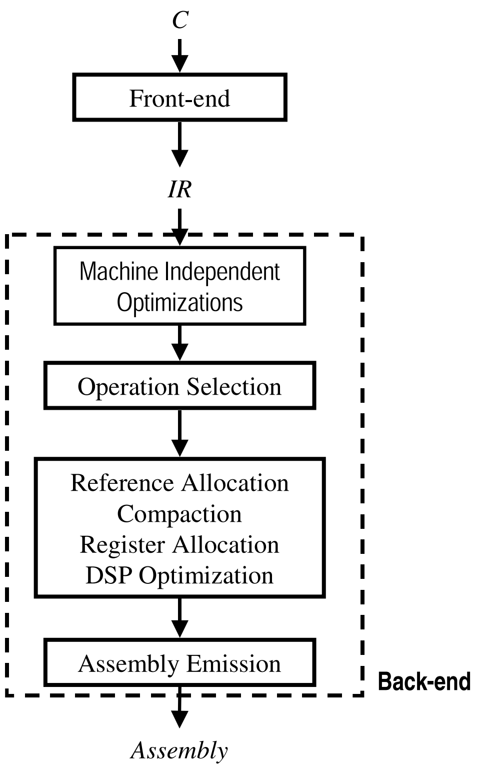


FIGURE 16.7 DSP code generator flow.

In general, aggressive machine-independent optimizations can also destroy information needed by DSP optimizations such as array reference allocation and offset assignment. Similar to general-purpose compilation, DSP compilation also suffers from the interplay between the various optimizations that is commonly known as the *phase-ordering* problem. For example, the most commonly used example of the phase-ordering problem is that between scheduling and register allocation. If scheduling is done prior to register allocation, then the variable live ranges can be long, which can increase register pressure and cause excessive spills (stores to memory). This may require another scheduling phase to be introduced after the allocation to take care of the spill (store to memory) and fill (load from memory) code. On the other hand, if register allocation is performed first, then several false dependencies are introduced that can lead to extended static schedules. The situation is no different in the case of DSP compilation. In the examples given in Sections 16.3.1 and 16.3.2, the constraints can be viewed as the problems of the scheduler, thereby creating unnecessary constraints for the register allocator; or they can be viewed as constraints for the register allocator, leading to bad schedules that do not exploit the ILP features of DSPs. Although the different phases in DSP compilation are listed in Figure 16.7, they may not necessarily be performed in the order shown.

Another concern with regard to optimizations is in terms of the compilation time. Most optimizations described in this section have been shown to be NP-complete. Given this, only a few options are available to a user — sacrifice optimality and convert the problem to a problem that can be solved in polynomial (preferably linear) time, or attempt to solve smaller versions of the problem optimally or use good heuristics. Hence, the user often needs to make a time vs. performance trade-off. This is an important issue in a retargetable design space exploration environment where faster techniques can be used to eliminate large chunks of the search space and longer near-optimal solutions can be applied to arrive at the final choice. Due to the large number of constraints in DSP compilation, some of the techniques use methods such as simulated annealing and linear programming, to solve the problems, and more often attempt to solve multiple problems or phases at a time. This leads to an increase in complexity and potentially increase in compilation time. In such cases, the trade-off is between optimality with some generality within an architecture class; and compilation time, complexity and scalability. Heuristics can result in completely different behavior even for small changes in the constraints, whereas an exact method can still produce robust solutions. This is another reason why a compiler framework and an MD need to be tied to one another.

In this section, we describe some of the common DSP optimizations performed to either exploit specialized architectural features or merely to satisfy data path constraints such as those mentioned in Section 16.3. For details on the conventional phases of general-purpose compilation such as scheduling and register allocation, we refer the readers to the corresponding chapters in this book.

16.5.1 Instruction Selection

Instruction selection corresponds to the task of selecting the sequence of appropriate target architecture opcodes from the intermediate representation of the application program. The output of most front ends of compilers is a forest of directed acyclic graphs (DAGs) semantically equivalent to the input program. It has been shown that the problem of covering the DAG with target opcodes is an NP-complete problem even for a single register machine [9, 15, 51]. However, optimal solutions exist when the intermediate representation is in the form of a sequence of expression trees (e.g., Iburg [14] and Twig [1]). Some of the DSP compilers use heuristics that convert a forest of DAGs to expression trees, potentially losing overall optimality, followed by locally (per tree) optimal pattern-matching algorithms to do instruction selection. Araujo et al. proposed a heuristic to transform a DAG into a series of expression trees [3] for acyclic architectures, classifying the ISA of DSPs as either cyclic or acyclic based on a register transfer graph (RTG) model.

The RTG model of an ISA describes how the different instructions utilize the transfer paths between the various storage locations to perform computations. The registers transparent to the developer through the processor ISA are divided into register classes, where each class has a specific function in the data path. For example, a typical DSP data path consists of an address register class, an accumulator class and a general-purpose register class.

16.5.1.1 Register Transfer Graph

An RTG is a directed acyclic multigraph where each node represents a register class and an edge between nodes r_i and r_j is labeled with instructions in the ISA that take operands from location r_i and store the result in location r_j. Memory is assumed to be infinitely large and not represented in the RTG. However, an arrowhead is added to the register class nodes for memory transfer operations with the arrowhead indicating the direction of memory transfer.

Figure 16.8(a) shows a simple processor data path and the corresponding RTG for this data path is shown in Figure 16.8(b). For clarity, the operations are not shown in the RTG. The accumulator ACC, register set R_i and register set R_j are the three register classes in the data path and hence have a vertex each in the RTG. Because the data path allows the accumulator to be both a source operand and a destination operand of some ALU operations, a self-edge exists around ACC in Figure 16.8(b). Because some ALU operations have R_i and R_j as sources and ACC as the destination, a directed edge exists from R_i and R_j each to ACC. The remaining arrowheads in the RTG represent memory operations. Load operations can be performed only with R_i or R_j as destinations and store operations can be performed using only the accumulator.

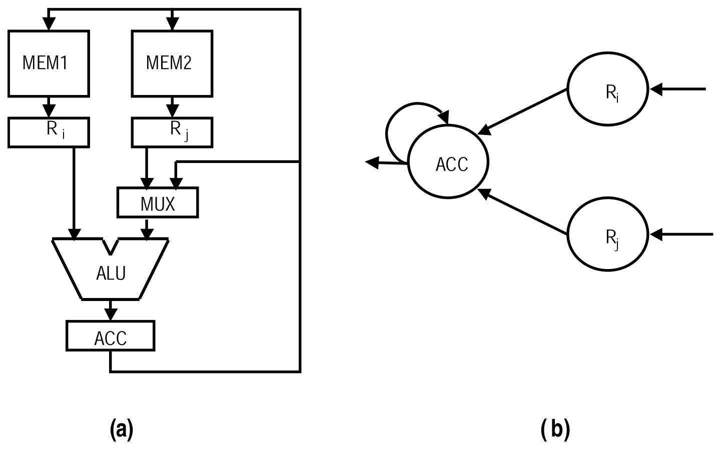


FIGURE 16.8 (a) Processor data path; (b) RTG for the data path.

An architecture is said to be cyclic (acyclic) if its RTG has (no) cycles where a cycle comprises at least two distinct vertices in the RTG. Acyclic ISAs have the property that any data path cycle, which is not a self-loop, including a path between two nodes r_i and r_j in the RTG, goes through memory. Hence, the data path shown in Figure 16.8(a) is acyclic because the RTG in Figure 16.8(b) does not have a cycle with two distinct vertices. Araujo and Malik show that spill-free code can be generated from expression trees for architectures with acyclic RTGs and provide a linear time algorithm for optimal scheduling for such architectures [4].

To transform the DAG into a series of expression trees, they use a heuristic with four phases:

- Partial register allocation is done for operands of operations for which the ISA itself clearly specifies the allocation. For example, for the data path in Figure 16.8(a), the results of all the ALU operations are always written to the accumulator.
- The architectural information and the set of constraints imposed by ISA are used to determine the set of edges in the DAG that can be broken without loss of optimality in the subsequent phases. For example, for the data path in Figure 16.8(a), the result of an ALU operation cannot be used as the left operand of a dependent operation. Such dependencies have to go through memory and thus the corresponding DAG edge, called *natural edge*, can be broken without any loss. Similarly, if an ALU operation is dependent on the results of two other ALU operations, then at least one of the dependencies has to go through memory. Such edges are called *pseudo-natural* edges and breaking such an edge does not guarantee optimality, but there is a reasonable probability that this is the case.
- Selective edges of the DAG are then marked and disconnected from the DAG to generate a forest of trees, while preserving the data flow constraints of the DAG.
- Optimal code is finally generated and scheduled for each expression tree.

Because the conversion of a DAG to expression trees potentially sacrifices optimality, Liao et al. proposed a technique using a combinatorial optimization algorithm called binate covering for instruction selection of DAGs. This can be solved either exactly or heuristically using branch and bound techniques [31]. This method, briefly summarized here, primarily involves the following steps for each basic block:

- Obtain all matching patterns of each node in the subject program graph and their corresponding costs. This phase does not take into account the data path constraints and assumes that the associated data transfers necessary are for free.

- Construct the covering matrix with a Boolean variable for each of the matched patterns as columns and the clauses and conditions that need to be satisfied for a legal cover of the program graph using the patterns as rows. This set of conditions are twofold, namely:
  - Each node must be covered by at least one pattern. This is represented as a set of clauses, one for each node in the program DAG, each of which is the inclusive OR of the Boolean variables that cover the node. For each row, or clause, an entry of 1 is made in the columns of Boolean variables included in the clause.
  - For each match, all the nonleaf inputs to the match must be outputs of other matches. This is represented by a set of implication clauses. For each nonleaf input, I, of each match, M, the set of all matches, M_I, that can generate I are determined. If M is chosen to cover a node or a set of nodes in the program DAG, then at least one of the matches from the set M_I must be chosen to obtain the input, I, to M. This needs to be true for each of the inputs of M and for all such matches chosen to cover the program graph. In the covering matrix, this is captured by entering a 0 in the column corresponding to M for all the implication clauses generated by M, and in each row corresponding to an implication clause of M, a 1 is entered in the columns of the variables included in the clause apart from M.
- A binate covering of the covering matrix that minimizes the cost and satisfies all the covering constraints — every row either has a 1 in the entry corresponding to a selected column or a 0 in the corresponding unselected column. The cost of a cover is the total cost of the selected columns. The main purpose of this covering is to generate complex instructions in the ISA from the program DAG that may not be possible by transforming to expression trees.
- The program graph is then modified into a new graph based on the obtained covering to reflect the complex instructions that have been generated.
- Additional clauses and costs corresponding to the irregular data path constraints, such as register class constraints of the selected instructions, and the consequent data transfer constraints and costs are added.
- A new binate covering of the modified graph using the new set of clauses is then obtained. The two-phase approach is used to contain the size of the code generation problem by solving for a smaller number of clauses.

A two-phase linear programming based method was developed by Leupers and Marwedel for DSPs with complex instructions [27, 29]. In the first phase, a tree-pattern matcher is used to obtain a cover on expression trees. In the second phase, complex instructions, which include a set of operations that can be issued in parallel, are generated from the cover (i.e., code scheduling is performed using linear programming). A constraint logic programming based code selection technique was developed by Leupers and Bashford for DSPs and media processors [24]. This technique addresses two potential drawbacks of tree-based methods. The first is that splitting DAGs into trees can potentially prevent generation of chained operations. For example, a multiply-accumulate operation cannot be generated if the result of the multiply is used by multiple operations, and covering expression trees with operation patterns does not take into account the available ILP in the ISA. Hence, the instruction selection algorithm takes the data flow graphs as input, keeps track of the set of all alternatives (both operation and instruction patterns that are possible) and then uses constraint logic programming to arrive at a cover. In addition, the authors also incorporate support for single instruction multiple data (SIMD)-style operations where a SIMD operation is essentially multiple-independent operations operating on independent virtual subregisters of a register.

In CodeSyn, Paulin et al. use a hierarchical treelike representation of the operations in the ISA to cover a control and data flow graph (CDFG) [39, 40]. If an operation in the tree does not match, then the search space for the matching patterns is reduced by pruning the hierarchical tree at the operation node. Once all the matching tree patterns are found, dynamic programming is used to select the cover of the CDFG.

Van Praet et al. use their instruction set graph (ISG) model of the target architecture, along with the control and data flow graph representation of the program in a branch-and-bound-based instruction selection algorithm in the CHESS compiler [22, 44]. The ISG is a mixed model of structural and behavioral models that captures the connectivity, ILP and constraints of the target processor. CHESS also performs instruction selection on the program DAG by covering the DAG with patterns called *bundles*. Bundles are partial instructions in the target processor that are generated dynamically for a DAG from the ISG as and when required. This is a key difference between other techniques that statically determine all possible bundles. Initially nodes in the DAG may be covered by more than one bundle that is then reduced to the minimum number of bundles required to cover the DAG by a branch and bound strategy.

In addition, there are also systems that solve multiple code generation problems in a combined manner. Hanono and Devadas provide a branch-and bound-based solution for performing instruction selection, and partial register allocation in a parallelism aware manner on DAGs [18]. At the core of this method is the split-node DAG representation of the program DAG. The split-node DAG is used to represent the program DAG with the set of all alternatives available for the nodes in the program DAG along with the corresponding data transfers. The algorithm then searches for a low-cost solution with the maximal parallelism among nodes in the split-node DAG while at the same time looking for the minimal number of instructions that cover the program DAG, on which graph coloring based register allocation is then performed. The valid operations, instructions and various constraints of the ISA are represented using the ISDL description.

Novack et al. developed the mutation scheduling framework, which is a unified method to perform instruction selection, register allocation and scheduling [36, 37]. This is achieved by defining a *mutation set* for each expression generating a value. The mutation set keeps track of all the possible machine-dependent methods of implementing an expression and this set is dynamically updated as scheduling is performed. For example, the references to a value depend on the register or memory assigned to it and the expression that records this information is added into the corresponding mutation set. During scheduling, an expression used to implement the corresponding value may be changed to better adapt to the processor resource constraints by replacing it with another expression in the mutation set.

Wilson et al. provide a linear programming-based integrated solution that performs instruction selection, register allocation and code scheduling on a DFG [62] and Wess provides another code generation technique based on trellis diagrams, an alternate form of representing instructions [60]. For more work in code generation, we refer readers to [8, 11, 19, 21, 30, 35, 43, 45, 48, 52].

16.5.2 Offset Assignment Problem

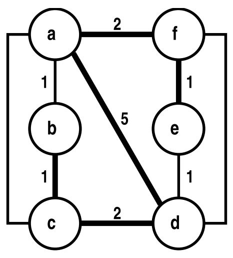
As stated in Section 16.2, DSPs provide special ARs, various addressing modes and address generation units. The address generation units are used by auto increment or auto decrement arithmetic operations, which operate on ARs used in memory accesses. The offset assignment (OA) problem addresses the issue of finding an ordering of variables within a memory bank that can reduce the amount of address computation code to a minimum by optimally utilizing the auto increment and auto decrement feature. This is shown in the example in Figure 16.10 taken from [32, 55]. Figure 16.10(b) and 16.10(d) shows the simple OA (SOA) optimized code sequence and the SOA unoptimized code sequence, respectively, for the piece of code in Figure 16.9(a). There are five additional address modification operations that are needed by the unoptimized version that places

```
c = a + b;
f = d + e;
a = a + d;
c = d + a;
d = d + f + a;
```

(a)

a b c d e f a d a d a c d f a d

(b)



(c)

FIGURE 16.9 (a) Code sequence; (b) access sequence; (c) access graph [32, 55].

```
b (0)
c (1)
d (2)
a (3) - AR
f (4)
e (5)
```

(a)

```
MOV AR, &a
LD [AR]          ; (a)
AR -= 3
LD [AR]++        ; (b)
c = a + b
ST [AR]++        ; (c)
LD [AR]          ; (d)
AR += 3
LD [AR]--        ; (e)
f = d + e
ST [AR]--        ; (f)
LD [AR]--        ; (a)
LD [AR]++        ; (d)
a = a + d
ST [AR]--        ; (a)
LD [AR]++        ; (d)
LD [AR]          ; (a)
c = d + a
AR -= 2
ST [AR]++        ; (c)
LD [AR]          ; (d)
AR += 2
LD [AR]--        ; (f)
d = d + f
LD [AR]--        ; (a)
d = d + a
ST [AR]          ; (d)
```

(b)

```
a (0) - AR
b (1)
c (2)
d (3)
e (4)
f (5)
```

(c)

```
MOV AR, &a
LD [AR]++        ; (a)
LD [AR]++        ; (b)
c = a + b
ST [AR]++        ; (c)
LD [AR]++        ; (d)
LD [AR]++        ; (e)
f = d + e
ST [AR]          ; (f)
AR -= 5
LD [AR]          ; (a)
AR += 3
LD [AR]          ; (d)
a = a + d
AR -= 3
ST [AR]          ; (a)
AR += 3
LD [AR]          ; (d)
AR -= 3
LD [AR]          ; (a)
c = d + a
AR += 2
ST [AR]++        ; (c)
LD [AR]          ; (d)
AR += 2
LD [AR]          ; (f)
d = d + f
 AR -= 5
LD [AR]          ; (a)
d = d + a
AR += 3
ST [AR]          ; (d)
```

(d)

FIGURE 16.10 (a) Optimized memory placement; (b) optimized code sequence; (c) unoptimized memory placement; (d) unoptimized code sequence.

variables in the order in which they are accessed in the code. Whereas simple offset assignment (SOA) addresses the offset assignment problem with one AR and increments and decrements by 1, the multiple AR and the *l* increment and decrement variant are addressed by general offset assignment (GOA) and *l*-SOA, respectively.

The following sequence of steps summarizes the work done by Liao et al. to solve the SOA problem [32] at the basic block level:

- For the sequence of arithmetic computations shown in Figure 16.9(a), the first step in SOA is to construct the access sequence shown in Figure 16.9(b). This is done by adding the source variables in each operation, from left to right, to the access sequence followed by the destination operand to the access.
- The next step is the construction of the access graph shown in Figure 16.9(c). Each vertex in this graph corresponds to a variable in the access sequence. An edge with weight *w* exists between two vertices if and only if the corresponding two variables are adjacent to each other *w* times in the access sequence.
- From the access graph, the cost of an assignment is equal to the sum of the weights of all edges connecting pairs of vertices that are not assigned adjacent locations in memory. Hence, the variables should be assigned memory locations in such a way that the sum of the weights of all edges connecting variables not assigned contiguous locations in memory is minimized. Liao et al. [32] have shown that this problem is equivalent to finding the maximum-weighted path covering (MWPC) of the access graph [32]. Because MWPC is NP hard, heuristics are used to find a good assignment of variables to memory locations. In Figure 16.9(c), the dark edges form an MWPC of the access graph with a cost of 4. Figure 16.10(a) shows the memory placement using the SOA path cover shown in Figure 16.9(c) and the consequent pseudo-assembly for a data path like the one shown in Figure 16.8(a). Additional optimizations such as memory propagation can be applied to the pseudo-assembly to prevent some unnecessary loads and stores. The heuristic given by Liao to find the MWPC is a greedy approach that looks at the edges in the access graph in decreasing order of weight, and adds an edge to the cover if it does not form a cycle with the edges already in the cover and if it does not increase the degree of a node in the cover to more than two.
- A heuristic to solve the general offset assignment problem with *k* address registers based on the simple offset assignment problem was also provided by Liao. This heuristic recursively partitions the accessed variables into two partitions, solves the simple assignment problem on each partition and then decides to either partition further or return the current partition based on the costs of each assignment. The total number of partitions generated is dependent on the number of address registers *k* in the architecture.

In the presence of control flow, the exact access sequence in a procedure can be known only during execution time. To perform offset assignment optimization at the procedure level instead of at the basic block level, they merge the access graphs of the basic blocks with equal weighting and the variables connected by control flow edges. After including the increments and decrements for each basic block, a separate phase decides whether to include an increment or a decrement across basic blocks.

Bartley was the first to address the SOA problem [7]. Leupers and Marwedel have also addressed the SOA problem [28]. Their work includes an improvement to the MWPC heuristic by introducing a tie-breaking function when the MWPC heuristic is faced with edges of equal weights and another heuristic to partition the vertices of the access graph for the GOA problem.

Sudarsanam, Liao and Devadas address the issue of l-SOA problem that corresponds to the case with one AR and an offset of $+/-\ l$ and the l, k-GOA problem with k address registers and a maximum offset of l [54]. In this work, the authors note that with an offset of more than 1, some of the edges in the access graph but not in the MWPC cover do not contribute to the cost of the cover. To identify these edges, called *induced edges*, they define an induced $(l + 1)$ clique of a cover C of the access graph G. Intuitively, if there is a path P of length l in G, then using the free 1-l autoincrement and decrement feature, it is possible to cover every edge that is a part of the complete subgraph induced by the $(l + 1)$ vertices of the path P in cover C on G. Hence, edges in G that are induced by a cover C are called induced edges; the sum of the weights of all edges in C and the edges induced by C is defined as the induced weight; and the sum of the weights of all edges that are in G, but are neither in C nor in the induced edges of C is defined as the l-induced cost of C. The l-SOA problem now reduces to finding a cover of an access graph G with the minimum l-induced cost. They define the (k, l)-GOA problem as given an access sequence L for a set of variables V, partition V into at most k partitions so that the total sum of the induced cost of each l-SOA and the associated setup costs is minimized.

Other work in offset assignment includes [26, 57, 61].

16.5.3 Reference Allocation

The DSP compiler phases of memory bank allocation and register allocation together constitute reference allocation. Memory bank allocation is performed to exploit the ILP between arithmetic operations and memory operations, and between memory operations themselves. The register allocator is similar to the general-purpose register allocator that decides which variables should be assigned to registers and which register each variable should be assigned to. As described in Section 16.3, DSPs can potentially have numerous operand and operation constraints that directly affect reference allocation. A general technique developed by Sudarsanam and Malik for reference allocation is presented here [55]. This is a simulated annealing-based technique that simultaneously performs memory bank allocation and register allocation phases while taking into account various constraints similar to those described in Section 16.3. The assumptions made by this technique include statically allocating all static and global variables to one memory bank and only compiling applications that are nonrecursive in nature. Also, the reference allocation technique described is performed after code compaction, or scheduling.

The first step involved in the algorithm is to generate the constraint graph. The constraint graph has a vertex for each symbolic register and variable. The problem of reference allocation is then transformed to a graph-labeling problem where every vertex representing a symbolic register must be labeled with a physical register and every vertex representing a variable must be labeled with a memory bank. The different types of constraint-weighted edges, where the weight corresponds to the penalty of not satisfying the constraint, are:

- A *red edge* is added between two symbolic registers if and only if they are simultaneously live [2]. The only exception to adding this edge is between two symbolic registers that are accessed in the same instruction by two memory operations. In this case, another type of edge ensures correctness. This edge ensures that the two symbolic registers connected by the edge are not assigned the same physical register. The cost of this edge is the amount of spill code that needs to be inserted if this constraint is violated. In their algorithm, this cost is assumed to be a large constant that reduces the chances of assigning two symbolic registers the same architectural register significantly. Hence, they do not compute the spill cost for each symbolic register.
- A *green edge* is added between two symbolic registers accessed by parallel memory operations to take into account any constraints that may exist between them. For example, for the instruction:

  $$MOV\ var_i, reg_j\ MOV\ var_k\ reg_l$$

  a green edge would be added in the constraint graph between the vertices corresponding to reg_j and reg_l to take care of any register constraints between the two symbolic registers. If a restriction exists on the two variables var_i and var_k concerning the memory banks to which they can be allocated, then this is captured by a pointer to var_i and var_k from reg_j and reg_l, respectively. The cost of this edge is the statically estimated instruction execution count because the two operations have to be issued in separate instructions if this constraint is not satisfied.

- Similar to green edges, *blue, brown, yellow edges* are added to the constraint graph with appropriate costs to represent constraints between parallel memory and register transfer operations. Blue edges are added for instructions involving a register transfer and a load operation. Brown edges are added for instructions involving an immediate load and a register transfer. Yellow edges are added for instructions involving a register transfer and a store operation.
- A *black edge* is added to represent operand constraints within an operation. These edges are added between symbolic registers and global vertices, which correspond to the register set of the DSP architecture. A black edge prevents the assignment of a symbolic register to the architectural register that it connects, due to the encodings of the ISA. Each black edge has cost ∞ because an unsatisfied black edge constraint is not supported by the hardware. For example, the accumulator cannot be a destination operand of load operations in the processor data path shown in Figure 16.9(a) and hence black edges would be added between symbolic registers representing the destination operands of load operations and the global vertex representing the accumulator.

Once the constraint graph has been constructed, the reference allocation algorithm uses simulated annealing to arrive at a low-cost labeling of the constraint graph. As an observation, they found that a greedy solution implemented to solve the constraint graph produced results that are very close to that produced by the simulated annealing algorithm [55]. The problem, however, with this approach is that for more varieties of constraints between operands, more complex formulations and additional colored edges would be needed.

Saghir, Chow and Lee have developed an algorithm to exploit the dual memory banks in DSPs using compaction-based partitioning and partial data duplication [49, 50]. In this method, an interference graph is constructed for each basic block with the variables accessed in the program as vertices and an edge is added between every pair of memory operations that can be legally performed, based on both the ISA and the data flow in the program. The edges are labeled with a cost that signifies the performance penalty if the corresponding two variables are not accessed simultaneously. The interference graph is then partitioned into two sets corresponding to the two memory banks such that the overall cost is minimum. During the interference graph construction, memory operations accessing the same variables and locations may be marked for duplication; these are placed in both memory banks and operations to preserve data integrity are inserted.

For more work on reference allocation, we refer the readers to [20, 41, 59].

16.5.4 Irregular Instruction Level Parallelism Constraints

Pressures to reduce code size and yet meet the power and execution time constraints of embedded applications often force designers to design processors, such as DSPs, with irregular architectures. As a consequence, DSPs are designed with small instruction widths and nonorthogonal opcode and operand encodings. This poses several problems to a compiler as illustrated in Section 16.3. In this section, we describe the artificial resource allocation (ARA) algorithm for the operation ILP problem developed by Rajagopalan, Vachharajani and Malik [47]. Conventional VLIW compilers use a reservation table-based scheduler that keeps track of the processor's resource usage as operations are scheduled. One of the highlights of the ARA algorithm is to allow compilers for irregular DSP architectures to use processor independent table-based schedulers instead of writing processor-specific schedulers. This is achieved by transforming the set of irregular operation ILP constraints to a set of artificial regular constraints that can subsequently be used by conventional VLIW schedulers.

The ARA algorithm takes as input the set of all possible combinations of operations that can be issued in parallel, and produces as output, an augmented resource usage of the MD such that the following constraints are satisfied:

Constraint 1: Every pair of operations that cannot be issued in parallel must share an artificial resource.
Constraint 2: Every pair of operations that can be issued in parallel must not share a resource.
Constraint 3: The total number of artificial resources generated must be minimum. This condition is used to reduce the size of the reservation table and potentially the scheduler time.

The ARA algorithm is explained next with an example from the *Fujitsu Hiperion* DSP ISA, as shown in Figure 16.11 taken from [46].

- The first step of ARA algorithm is construction of a *Compatibility graph* G. A vertex in the compatibility graph corresponds to an operation in the ISA. An edge exists between two vertices only if the two corresponding operations can be performed in parallel. For example, in Figure 16.11 there are five vertices in the compatibility graph, one each for *ADD*, *Shift*, *Multiply* and two *Loads*. The parallel combinations shown in the right of Figure 16.11 are captured by the straight-line edges in the compatibility graph.
- The complement G' of the compatibility graph shown in dotted lines in Figure 16.11 is then constructed. An edge between two vertices in G' implies that the two operations cannot be performed in parallel. An immediate solution that follows is to assign an artificial resource to each edge in G'. Because this solution can cause the size of reservation table to grow significantly, a better solution is obtained in the next step.
- The minimum number of artificial resources needed to satisfy the operation ILP constraints is obtained by performing the minimum edge clique cover on G' [47]. The problem of finding the minimum edge clique cover on G' is first converted to an equivalent problem of finding the minimum vertex clique cover on graph G_1. G_1 is obtained from G' as follows. A vertex in G_1 corresponds to an edge in G' and an edge exists between two vertices in G_1 if and only if the vertices connected by the corresponding edges in G' form a clique. Hence, a vertex clique cover of G_1 identifies larger cliques in G'. This problem in turn is transformed to graph coloring [15] on the complement of G_1. In the Figure 16.11 example, 3 cliques $C1$, $C2$ and $C3$ are required to obtain the minimum edge clique cover.
- The final step in the ARA algorithm translates the result of the minimum edge clique cover algorithm into the resource usage section of the processor machine description.

The advantages of using this algorithm are that it is a highly retargetable solution to the operation ILP problem and it helps avoid processor-specific schedulers when such irregular constraints exist. Eichenberger and Davidson have provided techniques to compact the size of machine descriptions [13] of general-purpose processors. Gyllenhaal and Hwu provided methods to optimize not

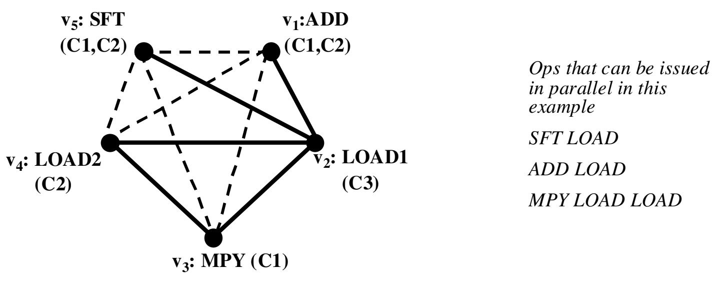


FIGURE 16.11 ARA Algorithm applied to a subset of *Hiperion* ISA [46].

only the size of the MDs but also the improvement of the efficiency of queries [17]. For more work on MDs, we refer the readers to the chapter on machine descriptions.

16.5.5 Array Reference Allocation

The ISA of DSPs provides special address modifying instructions that update address registers used by memory operations. Because many signal-processing applications contain kernels that operate on arrays, one of the most important DSP optimizations is to assign ARs to array references so that a majority of the address computation operations can be replaced by auto increment and decrement address modifying operations that are free. This can significantly affect the static code size and the dynamic performance of the assembly code because address computation for successive memory operations is performed in parallel with current memory operations. In this section, we describe the array reference allocation algorithm developed by Cintra and Araujo [10] and Ottani et al. [38].

Global reference allocation (GRA) is defined as the problem of allocating ARs to array references such that the number of simultaneously live ARs is kept below the maximum number of such registers available in the processor, and the number of instructions required to update them is minimized. The local version of this problem, called *local reference allocation* (LRA), has all references restricted to basic block boundaries. There are known efficient graph based solutions for LRA [16, 25]. Araujo, Sudarsanam and Malik proposed a solution for optimizing array references within loops based on an index graph structure [5]. The index graph is constructed as follows. A vertex exists for each array access in the loop. An edge exists between two accesses only if the indexing distance between the two accesses is less than the limit of the auto increment and decrement limit in hardware. The array reference allocation problem then deals with finding the disjoint path cycle cover that minimizes the number of paths and cycles. This is similar to the offset assignment algorithm explained in Section 16.5.2.

Whereas general-purpose register allocation concerns itself with allocating a fixed set of general-purpose registers to a potentially much larger set of virtual registers in the program, reference allocation pertains to assigning a fixed set of address registers to the various memory access references in the program. When two virtual registers are assigned the same general register, the allocator's responsibility is to insert the appropriate spill (store to memory) code and fill code (load from memory). Similarly, when reference allocation combines two references using a single AR, it is the reference allocator's responsibility to insert appropriate update operations.

We now describe the GRA algorithm developed by Cintra and Araujo [10] called *live range growth* (LRG) to assign ARs to array references:

- The set of array reference ranges is first initialized to all the independent array references in the program.
- The next step involves reducing the number of array reference ranges in the program to the number of ARs in the processor. This involves combining array reference ranges and at the same time minimizing the number of reference update operations. To quantify the cost of combining reference ranges, the notion of an indexing distance is defined as follows:

DEFINITION 16.1. *Let a and b be array references and s the increment of the loop containing these references. Let index(a) be a function that returns the subscript expression of reference a. The indexing distance between a and b is the positive integer:*

$$d(a,b) = \begin{cases} |index(b) - index(a)| & \text{if } a < b \\ |index(b) - index(a) + s| & \text{if } a > b \end{cases}$$

where $a < b$ $(a > b)$ if a (b) precedes b (a) in the schedule order.

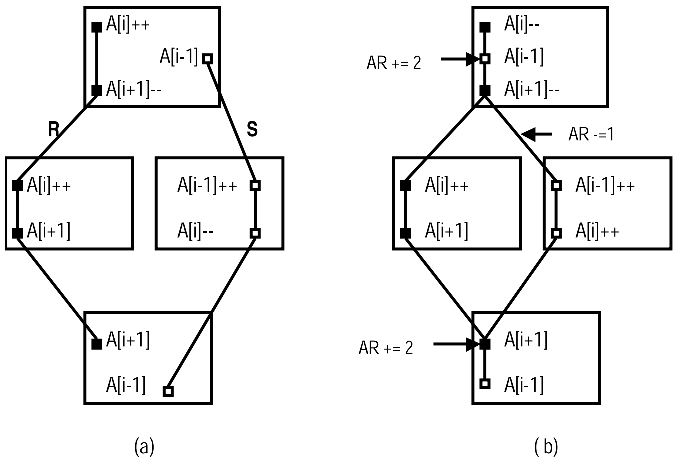


FIGURE 16.12 (a) Two array reference live ranges R, S; (b) after merging live ranges *R* and *S* into a single live range [46].

An update operation is required when combining two reference ranges whenever the indexing distance is greater (lesser) than the maximum (minimum) allowed automatic increment (decrement) value. Hence, the cost of combining two ranges is the number of update instructions that needs to be inserted. This is shown in the example in Figure 16.12(b) taken from [46] where the two live ranges *R* and *S* in Figure 16.12(a), with an indexing distance of 1, have been merged into a single live range. To maintain correctness, necessary AR update operations have been inserted both within basic blocks and along the appropriate control flow edges.

To facilitate the computation of the indexing distances, an important requirement is that the references in the control flow graph are in single reference form [10], a variation of *static single assignment* (*SSA*) *form* [12].

Although GRA is a typical DSP optimization, it is implemented along with machine-independent optimizations. Actually, it is performed before most of the classical optimizations. This is because techniques such as common subexpression elimination (CSE) may destroy some opportunities for applying GRA.

Leupers, Basu and Marwedel have proposed an optimal algorithm for AR allocation to array references in loops [25]. Liem, Paulin and Jerraya have developed an array reference analysis tool that takes, as input, the C program with array references and the address resources of the hardware, and produces an equivalent C program output in which the array references have been converted to optimized pointer references for the given hardware [33].

16.5.6 Addressing Modes

16.5.6.1 Paged Absolute Addressing

A consequence of reducing instruction width to reduce static code size, as noted in Sections 16.2 and 16.5.4, is that memory operations using absolute addressing mode (i.e., using physical address of memory location) to access data have to be encoded in more than one instruction word. To overcome this extra word penalty and potentially an extra cycle penalty, architectures such as the TI TMS320C25 [58] feature a paged absolute addressing mode. In these architectures, a single data memory space is partitioned into N nonoverlapping pages numbered 0 through $N - 1$ and a special page pointer register is dedicated to storing the ordinal of the page that is currently accessed. The absolute address can then be specified by loading the page pointer register with the appropriate page number and specifying the offset within the page in the memory operation. This can potentially save instruction memory words and reduce execution time for a sequence of memory operations accessing the same page. In addition, because a machine register is used instead of ARs or general-purpose registers, a better register assignment of the program code is also possible. Because DSP programs are nonrecursive in nature, the automatic variables are statically allocated in memory to exploit absolute addressing and relieve ARs. In this section, we describe some optimizations developed by Sudarsanam et al. to exploit absolute paged addressing modes [56].

The *LDPK* operation is used in the TI TMS320C25 DSP to load the page pointer register. To justify the use of absolute addressing for automatic variables, the *LDPK* operation overhead should be minimum. Hence, this algorithm tries to reduce the number of *LDPK* operations using data flow analysis [2]. The main steps in this algorithm are as follows:

- Assuming that code is generated on a per basic block basis, the algorithm conservatively assumes that the first operation to use absolute addressing in each basic block should be preceded by an *LDPK* operation.
- Another conservative assumption is that the value of the page pointer register is not preserved across procedure calls and hence must be restored after each procedure call.
- An *LDPK* operation can be suppressed before an operation using absolute addressing if the new value of the page pointer register is the same as the current value of the page pointer register.
- By using data flow analysis [2] on the assembly code, the set of unnecessary *LDPK* operations are determined across basic blocks. For each basic block B and a page pointer register DP, the data flow equations and the variables used by this optimization are:
    - IN(B): value of DP at the entry of B.
    - OUT(B): value of DP at the exit of B.
    - LAST(B): ordinal of the last referenced page in B.
    - While traversing B, LAST(B) is assigned the value UNKNOWN after a procedure call and prior to an *LDPK* operation.
    - LAST(B) is assigned the ordinal of the last *LDPK* operation prior to a procedure call while traversing B.
    - Finally, LAST(B) is assigned PROPAGATE if neither an *LDPK* operation nor a procedure call is encountered after the traversal of B.
    - While (OUT values have changed) {

```
for each basic block B {
    IN(B) = $\bigcap$ OUT(P)  for all P, predecessor of B;
    if (LAST(B) == PROPAGATE)
        OUT(B) = IN(B);
    else
        OUT(B) = LAST(B);
}
}
```

- After computing the data flow variables for each basic block, if the first *LDPK* operation of a basic block B is not preceded by a procedure call, then it can be removed if the ordinal of the *LDPK* operation is the same as IN(B).

An extension of this algorithm for interprocedural optimization was also provided by Sudarsanam et al. [56]. Lin provides a simple postpass algorithm that removes redundant *LDPK* instructions within basic blocks, assuming that an *LDPK* is generated for each access of a statically allocated variable [34].

16.6 Summary and Conclusions

DSPs are a significant fraction of the processor market today and, in particular, the low-cost embedded DSPs find themselves in a wide variety of products and applications of daily use. Hence, it is important not only to design the necessary tools to program these processors but also to develop tools that can help in an automatic cooperative synthesis of both the architecture and the associated software tool set for a given set of applications. A first step in this direction is the design of a highly reusable and retargetable compiler for DSPs. In this chapter, we first classified DSP architectures into two classes: FOW DSPs and VOW DSPs. Although DSPs in the former category are more regular with orthogonal instruction sets, the ones in the latter category are the low-end processors with highly optimized irregular architectures. VOW DSPs are also more cost sensitive, require special algorithms to exploit their hardware features and thus are the focus of this chapter. Sections 16.2 and 16.3 explain the consequences of a VOW-based design and point out the nature of constraints that a DSP compiler must solve. In Section 16.4, retargetable compilation is classified into three categories: automatic retargetability, user retargetability and developer retargetability depending on the following criteria, namely, the level of automation, the level of user interaction in retargeting and the level of architecture coverage provided by the compiler framework. Retargetability is a key factor in design automation and reduction of time to market. A potential approach for retargetable compiler development is also presented along with some qualitative measures.

Finally, in Section 16.5, we discuss some DSP architecture-specific issues that a compiler framework developer must solve to achieve a high level of retargetability and good efficiency. We have presented some of the common DSP optimizations that have been developed. In addition to the architecture constraints described, new features are available such as media operations that usually pack multiple identical operations into a single operation operating on smaller bit-width operands. Hence, DSP compiler frameworks also need to be extensible in that it should be easy to add new architectural features and new algorithms to exploit such features to the framework.

References

[1] A. Aho, M. Ganapathi and S. Tjiang, Code generation using tree matching and dynamic programming, *ACM Trans. Programming Languages Syst.*, 11(4), 491–516, October 1989.

[2] A.V. Aho, R. Sethi and J.D. Ullman, *Compilers: Principles, Techniques, and Tools,* Addison-Wesley, Reading, MA, 1988.

[3] G. Araujo, *Code Generation Algorithms for Digital Signal Processors*, Ph.D. thesis, Princeton University, Princeton, NJ, 1997.

[4] G. Araujo and S. Malik, Code generation for fixed-point DSPs, *ACM Trans. Design Automation Electron. Syst.*, 3(2), 136–161, April 1998.

[5] G. Araujo, A. Sudarsanam and S. Malik, Instruction Set Design and Optimizations for Address Computation in DSP Processors, in *9th International Symposium on Systems Synthesis*, IEEE, November 1996, pp. 31–37.

[6] D.I. August, D.A. Connors, S.A. Mahlke, J.W. Sias, K.M. Crozier, B.-C. Cheng, P.R. Eaton, Q.B. Olaniran and W.W. Hwu, Integrated Predicated and Speculative Execution in the IMPACT EPIC Architecture, in Proceedings of the 25th International Symposium on Computer Architecture, July 1998.

[7] D.H. Bartley, Optimizing stack frame accesses for processors with restricted addressing modes, *Software Pract. Exp.*, 22(2), February 1992.

[8] S.Bashford and R. Leupers, Phase-coupled mapping of data flow graphs to irregular data paths, *Design Automation Embedded Syst.*, 4(2/3), 1999.

[9] J.L. Bruno and R. Sethi, Code generation for one-register machine, *J. ACM*, 23(3), 502–510, July 1976.

[10] M. Cintra and G. Araujo, Array Reference Allocation Using SSA-Form and Live Range Growth, in Proceedings of the ACM SIGPLAN LCTES 2000, June 2000, pp. 26–33.

[11] C. Liem, *Retargetable Code Generation for Digital Signal Processors*, Kluwer Academic, Dordrecht, 1997.

[12] R. Cytron, J. Ferrante, B. Rosen, M. Wegman and F. Zadeck, An Efficient Method of Computing Static Single Assignment Form, in Proceedings of the ACM POPL '89, 1989, pp. 23–25.

[13] A.E. Eichenberger and E.S. Davidson, A Reduced Multipipeline Machine Description that Preserves Scheduling Constraints, in Proceedings of the Conference on Programming Language Design and Implementation, 1996.

[14] C. Fraser, D. Hanson and T. Proebsting, Engineering a simple, efficient code-generator generator, *ACM Lett. Programming Languages Syst.*, 1(3), 213–226, September 1992.

[15] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman, San Francisco, 1979.

[16] C. Gebotys, DSP Address Optimization Using a Minimum Cost Circulation Technique, in *Proceedings of the International Conference on Computer-Aided Design*, IEEE, November 1997, pp. 100–103.

[17] J.C. Gyllenhaal and W.W. Hwu, Optimization of Machine Description for Efficient Use, in Proceedings of the 29th International Symposium on Microarchitecture, 1996.

[18] S. Hanono and S. Devadas, Instruction Selection, Resource Allocation, and Scheduling in the AVIV Retargetable Code Generator, in Proceedings of the 35th Design Automation Conference, June 1998, pp. 510–515.

[19] R. Hartmann, Combined Scheduling and Data Routing for Programmable ASIC Systems, in European Conference on Design Automation (EDAC), 1992, pp. 486–490.

[20] D.J. Kolson, A. Nicolau, N. Dutt and K. Kennedy, Optimal Register Assignment to Loops for Embedded Code Generation, in Proceedings of 8th International Symposium on System Synthesis, 1995.

[21] D. Lanneer, M. Cornero, G. Goossens and H.D. Man, Data Routing: A Paradigm for Efficient Data-Path Synthesis and Code Generation, in 7th International Symposium on High Level Synthesis, 1994, pp. 17–21.

[22] D. Lanneer, J.Van Praet, A. Kifli, K. Schoofs, W. Geurts, F. Thoen and G. Goossens, CHESS: Retargetable code generation for embedded DSP processors, in Code Generation for Embedded Processors, Kluwer Academic Dordrecht, 1995, pp. 85–102.

[23] P. Lapsley, J. Bier, A. Shoham and E.A. Lee, *DSP Processor Fundamentals*, IEEE Press, Washington, 1997.

[24] R. Leupers and S. Bashford, Graph based code selection techniques for embedded processors, *ACM Trans. Design Automation Electron. Syst.*, 5(4), October 2000.

[25] R. Leupers, A. Basu and P. Marwedel, Optimized Array Index Computation in DSP Programs, in Proceedings of the ASP-DAC, 1998.

[26] R. Leupers and F. David, A Uniform Optimization Technique for Offset Assignment Problems, in International Symposium on System Synthesis, 1998.

[27] R. Leupers and P. Marwedel, Time-Constrained Code Compaction for DSPs, in Proceedings of the 8th International Symposium on System Synthesis, September 1995.

[28] R. Leupers and P. Marwedel, Algorithms for Address Assignment in DSP Code Generation, in Proceedings of International Conference on Comptuer Aided Design, 1996.

[29] R. Leupers and P. Marwedel, Instruction Selection for Embedded DSPs with Complex Instructions, in European Design and Automation Conference (EURO-DAC), September 1996.

[30] R. Leupers and P. Marwedel, Retargetable code generation based on structural processor descriptions, *Design Automation Embedded Syst.*, 3(1), 1–36, January 1998.

[31] S. Liao, S. Devadas, K. Keutzer and S. Tjiang, Instruction Selection Using Binate Covering for Code Size Optimization, in Proceedings International Conference on Computer Aided Design, 1995, pp. 393–399.

[32] S. Liao, S. Devadas, K. Keutzer, S. Tjiang and A. Wang, Storage assignment to decrease code size, *ACM Trans. Programming Languages Syst.*, 18, 235–253, May 1996.

[33] C. Liem, P. Paulin and A. Jerraya, Address Calculation for Retargetable Compilation and Exploration of Instruction-Set Architectures, in Proceedings of the 33rd Design Automation Conference, 1996.

[34] W. Lin, An Optimizing Compiler for the TMS320C25 DSP Processor, Master's thesis, University of Toronoto, Canada, 1995.

[35] P. Marwedel and G. Goossens, Eds., *Code Generation for Embedded Processors*, Kluwer Academic, Dordrecht, 1995.

[36] S. Novack and A. Nicolau, Mutation Scheduling: A Unified Approach to Compiling for Fine-Grain Parallelism, in Languages and Compilers for Parallel Computing, 1994, pp. 16–30.

[37] S. Novack, A. Nicolau and N. Dutt, A unified code generation approach using mutation scheduling, in *Code Generation for Embedded Processors*, Kluwer Academic, Dordrecht, 1995, pp. 65–84.

[38] G. Ottoni, S. Rigo, G. Araujo, S. Rajagopalan and S. Malik, Optimal Live Range Merge for Address Register Allocation in Embedded Programs, in Proceedings of the CCO1 — International Conference on Compiler Construction, April 2001.

[39] P. Paulin, C. Liem, T. May and S. Sutarwala, CodeSyn: A Re-targetable Code Synthesis System, in Proceedings of the 7th International High-Level Synthesis Workshop, 1994.

[40] P. Paulin, C. Liem, T. May and S. Sutarwala, Flexware: A flexible firmware development environment for embedded sytems, in *Code Generation for Embedded Processors*, Kluwer Academic, Dordrecht, 1995, pp. 65–84.

[41] D. Powell, E. Lee and W. Newman, Direct synthesis of optimized DSP assembly code from signal flow block diagrams, in Proceedings of the International Conference on Acoustics, Speech and Signal Processing, Vol. 5, 1992, pp. 553–556.

[42] P.P. Chang, S.A. Mahlke, W.Y. Chen, N.J. Warter and W.W. Hwu, IMPACT: An Architectural Framework for Multiple-Instruction-Issue Processors, in Proceedings of the 18th International Symposium on Computer Architecture, May 1991, pp. 266–275; *www.crhc.uiuc.edu/IMPACT.*

[43] J.V. Praet, G. Goossens, D. Lanneer and H.D. Man, Instruction Set Definition and Instruction Selection for ASIPs, in 7th International Symposium on High Level Synthesis, 1994, pp. 11–16.

[44] J.V. Praet, D. Lanneer, W. Geurts and G. Goossens, Processor modeling and code selection for retargetable compilation, *ACM Trans. Design Automation Electron. Syst.*, 6(2), 277–307, July 2001.

[45] J.V. Praet, D. Lanneer, G. Goossens and W.G.D. Man, A Graph Based Processor Model for Retargetable Code Generation, in European Design and Test Conference, 1996.

[46] A. Rajagopalan, S.P. Rajan, S. Malik, S. Rigo, G. Araujo and K. Takayama, A retargetable VLIW compiler framework for DSPs with instruction-level-parallelism, *IEEE Trans. Comput. Aided Design Integrated Circuits Syst.*, 20(11), 1319–1328, November 2001.

[47] S. Rajagopalan, M. Vachharajani and S. Malik, Handling Irregular ILP within Conventional VLIW Schedulers Using Artificial Resource Constraints, in Proceedings of International Conference on Compilers Architecture and Synthesis for Embedded Systems, November 2000.

[48] K. Rimey and P.N. Hilfinger, Lazy Data Routing and Greedy Scheduling for Application Specific Signal Processors, in 21st Annual Workshop on Microprogramming and Microarchitecture (MICRO-21), 1988, pp. 111–115.

[49] M.A.R. Saghir, P. Chow and C.G. Lee, Automatic Data Partitioning for HLL DSP Compilers, in Proceedings of the 6th International Conference on Signal Processing Applications and Technology, October 1995, pp. 866–871.

[50] M.A.R. Saghir, P. Chow and C.G. Lee, Exploiting Dual Data-Memory Banks in Digital Signal Processors, in Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems, October 1996, pp. 134–243.

[51] R. Sethi, Complete register allocation problems, *SIAM J. Comput.*, 4(3), 226–248, September 1975.

[52] M. Strik, J. van Meerbergen, A. Timmer and J. Jess, Efficient Code Generation for in-House DSP Cores, in European Design and Test Conference, 1995, pp. 244–249.

[53] A. Sudarsanam, Code Generation Libraries for Retargetable Compilation for Embedded Digital Signal Processors, Ph.D. thesis, Princeton University, Princeton, NJ, November 1998.

[54] A. Sudarsanam, S. Liao and S. Devadas, Analysis and Evaluation of Address Arithmetic Capabilities in Custom DSP Architectures, in Proceedings of ACM/IEEE Design Automation Conference, 1997.

[55] A. Sudarsanam and S. Malik, Memory Bank and Register Allocation in Software Synthesis for ASIPs, in Proceedings of International Conference on Computer-Aided Design, 1995.

[56] A. Sudarsanam, S. Malik, S. Tjiang and S. Liao, Optimization of Embedded DSP Programs Using Post-Pass Data-Flow Analysis, in Proceedings of 1997 International Conference on Acoustics, Speech, and Signal Processing, 1997.

[57] N. Sugino, H. Miyazaki, S. Iimure and A. Nishihara, Improved Code Optimization Method Utilizing Memory Addressing Operation and Its Application to DSP Compiler, in International Symposium on Circuits and Systems, 1996.

[58] Texas Instruments, TMS320C2x User's Guide, c edition, January 1993.

[59] B. Wess, Automatic Code Generation for Integrated Digital Signal Processors, in Proceedings of the International Symposium on Circuits and Systems, 1991, pp. 33–36.

[60] B. Wess, Code generation based on trellis diagrams, in *Code Generation for Embedded Processors*, Kluwer Academic, Dordrecht, 1995, pp. 188–202.

[61] B. Wess and M. Gotschlich, Constructing Memory Layouts for Address Generation Units Supporting Offset 2 Access, in Proceedings of ICASSP, 1997.

[62] T. Wilson, G. Grewal, S. Henshall and D. Banerji, An ILP based approach to code generation, in *Code Generation for Embedded Processors*, Kluwer Academic, Dordrecht, 1995, pp. 103–118.

17
Instruction Scheduling

R. Govindarajan
Indian Institute of Science

17.1 Introduction

Ever since the advent of reduced instruction set computers (RISC) [65, 113] and their pipelined execution of instructions, instruction scheduling techniques have gained importance because they rearrange instructions to "cover" the delay or latency that is required between an instruction and its dependent successor. Without such reordering, pipelines would stall, resulting in wasted processor cycles. Pipeline stalls would also occur while executing control transfer instructions, such as branch and jump instructions. In architectures that support delayed branching, where the control transfers are effected in a delayed manner [66], instruction reordering is again useful to cover the stall cycles with useful instructions. Instruction scheduling can be limited to a single basic block — a region of straight-line code with a single point of entry and a single point of exit, separated by decision points and merge points [3, 71, 101] — or to multiple basic blocks. The former is referred as *basic block scheduling* whereas the latter as *global scheduling* [3, 71, 101].

0-8493-1240-7/03/$0.00+$1.50

A significant amount of research conducted in industry and academia has resulted in processors that issue multiple instructions per cycle, and hence exploit instruction-level parallelism (ILP) [121]. Exploiting ILP has lent itself as a viable approach for providing continuously increasing performance without having to rewrite applications. ILP processors have been classified into two broad categories, namely, very long instruction word (VLIW) processors [31, 45, 126] and superscalar processors [75, 135, 136] depending on whether the parallelism is exposed at compile time or at runtime by dynamic instruction scheduling hardware. In VLIW machines, a compiler identifies independent instructions and communicates it to the hardware by packing them in a single long word instruction. At runtime, a long word instruction is fetched and decoded, and the independent instructions in it are executed in parallel in the multiple functional units available in the VLIW architecture. In a superscalar machine, on the other hand, a complex hardware identifies independent instructions and issues them in parallel at runtime.

Multiple instruction issue per cycle has become a common feature in modern processors (see, e.g., [69, 79, 106, 131, 156]). The success of ILP processors has placed even more pressure on instruction scheduling methods, because exposing ILP is the key for the performance of ILP processors. Instruction scheduling can be done by hardware at runtime [66, 136] or by software at compile time [50, 64, 85, 121, 151]. In this discussion, we concentrate on compile-time instruction scheduling methods. Such compile-time instruction scheduling is solely responsible for exposing and exploiting the parallelism available in a program in a VLIW architecture. Thus, without the instruction scheduler, the (early) VLIW processors could not have achieved an ILP of 7 to 14 operations that they are capable of issuing in a cycle [31, 126].

In superscalar processors, instruction scheduling hardware determines at runtime the independent instructions that can be issued in parallel. However, the scope of the runtime scheduler is limited to a narrow window of 16 or 32 instructions [136]. Hence, compile-time techniques may be needed to expose parallelism beyond this window size. Further, in a certain class of superscalar processors, namely, the *in-order* issue processors [66, 136], instructions are issued in program order. Hence, when an instruction is stalled due to data dependence, instructions beyond the stalled one are also stalled. Instruction scheduling can be beneficially applied for these in-order issue architectures to rearrange instructions and hence exploit higher ILP. Thus, even superscalar processors can benefit from the parallelism exposed by a compile-time scheduler.

Instruction scheduling methods for basic blocks may result in a moderate improvement (less than 5 to 10%) in performance, in terms of the execution time of a schedule, for simple pipelined RISC architectures [64]. However the performance improvement achieved for multiple instruction issue processors could be significant (more than 20 to 50%) [97]. Instruction scheduling beyond basic blocks can achieve even higher performance improvement, ranging from 50 to 300% [71, 72, 95].

Instruction scheduling is typically performed after machine-independent optimizations, such as copy propagation, common subexpression elimination, loop-invariant code motion, constant folding, dead-code elimination, strength reduction and control flow optimizations [3, 101]. Instruction scheduling is performed either on the target machine assembly code or on a low-level code that is very close to the machine assembly code. In certain implementations, instruction scheduling is performed after register allocation — another important compiler optimization that determines which variables are stored in registers and which remain in memory. When instruction scheduling follows register allocation, it is referred to as the *postpass scheduling* approach [3, 101]. In the *prepass scheduling* or *prescheduling* approach, instruction scheduling precedes register allocation. In prepass scheduling, because register allocation is performed subsequently, any code introduced due to register spills is not scheduled by the scheduler. Hence, in prepass scheduling, to handle the spill code, the instruction scheduler may be invoked again after register allocation. Instruction scheduling and register phases influence each other and hence the ordering between the two phases in a compiler is an important issue. A number of methods integrate the two phases to produce efficient register allocated instruction schedules [15, 18, 52, 100, 107, 117].

Early work on instruction scheduling related it to the problem of code compaction in microprogramming [44]. This relationship between microprogram compaction and instruction scheduling has been beneficially used in local or basic block scheduling. In fact, as pointed out in [121], this mindset remained as a serious obstacle to achieve good performance in global scheduling (instruction scheduling beyond basic blocks) until trace scheduling [45] and other approaches were proposed. These latter approaches minimize the execution time of the most likely trace or control path, instead of obtaining a compact schedule. Further, the similarities between job shop scheduling [29] and instruction scheduling was also well understood. Instruction scheduling borrows a number of concepts and algorithms from scheduling theory. Many of the theoretical results in instruction scheduling owe their origin to job scheduling.

The objective of this chapter is more to provide an overview of instruction scheduling methods than to provide a comprehensive survey of all existing techniques. The following section presents the necessary background. Simple scheduling methods for covering pipeline delays are discussed in Section 17.3. Subsequently, we describe basic block instruction scheduling methods for VLIW and superscalar processors in Section 17.4. Section 17.5 deals with global scheduling techniques. In Section 17.6, phase-ordering issues relating to instruction scheduling and register allocation are presented. Section 17.7 discusses recent research in instruction scheduling. Finally, we provide a concluding summary in Section 17.8.

17.2 Background

In this section we review the relevant background. The following subsection presents a number of definitions. In Section 17.2.2 we describe a representation for data dependences, used by instruction scheduling methods, and its construction. Finally, we discuss various performance metrics for instruction scheduling methods in Section 17.2.3.

Before we proceed further, let us clarify the use of various notations in the programming examples. We use `t1`, `t2`, etc. to represent temporaries or symbolic registers, and `r1`, `r2`, etc. to represent (logical) registers assigned to temporaries. Symbols, such as x, y or a, b, etc., represent variables stored in memory locations.

17.2.1 Definitions

Two instructions `i1` and `i2` are said to be data dependent on each other if they share a common operand (register or memory operand), and the shared operand appears as a destination operand[1] in at least one of the instructions [3, 71, 101]. Consider the following sequence of instructions:

```
i1:  r1 ← load (r2)
i2:  r3 ← r1 + 4
i3:  r1 ← r4 − r5
```

Instruction `i2` has r1 as one of its source operands, which is written by `i1`. This dependence from `i1` to `i2` is said to be a flow dependence, or true data dependence. Thus, in any legal execution of the preceding sequence, the operand read of register `r1` in instruction `i2` must take place after the

[1]If the shared operand appears as a source operand in both instructions, then there is an input dependence between the two instructions. Because an input dependence does not constrain the execution order, we do not consider this any further in our discussion.

result value of instruction i1 is written. The dependence between instructions i2 and i3 due to register r1 is an antidependence. Here, instruction i3 must write the result value in r1 only after i2 has read its operand from r1. Finally, there is an output dependence between instructions i1 and i3, where the order in which they write to the destination must be the same as the program order (i.e., i1 before i3) for correct program behavior.

As mentioned earlier, a data dependence could also arise through a memory operand. Detecting such a data dependence accurately at compile time is hard, especially if the memory operands are accessed using indirect addressing modes. The problem becomes harder in the presence of memory aliasing, where two or more variables point to the same memory location. As a consequence, a conservative analysis of data dependence must assume a true dependence from each previous store to every subsequent load instruction. For the same reason, there is an antidependence from each load to every previous store instruction. Finally, an output dependence is from each store to all subsequent stores.

Anti- and output dependences together are referred to as *false dependences*. These dependences arise due to the reuse of the same register variable or memory location. By appropriately renaming the destination register of i3 (i.e., using a different destination register, the anti- and output dependences can be eliminated). If the dependences are analyzed before register allocation, then the code sequence uses only temporaries. Because no limit exists on the number of temporaries that can be used, anti- and output dependences on the temporaries do not normally occur. However, anti- and output dependences on memory variables accessed through load and store operations can still occur.

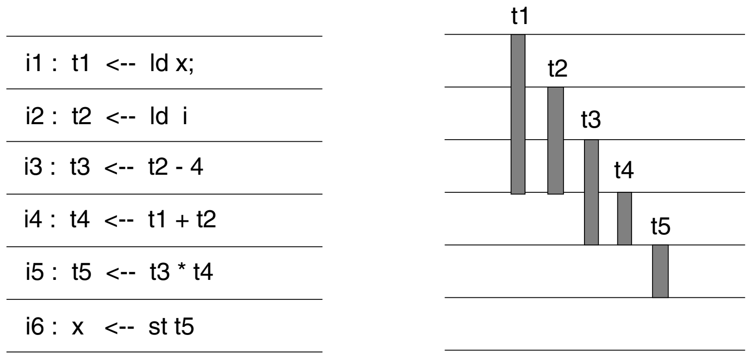
A basic block is a region of straight-line code [3, 71, 101]. The execution control, also referred to as control flow, enters a basic block at the beginning (i.e., the first instruction in the basic block), and exits at the end (i.e., the last instruction). A control flow transfer or jump occurs from one program point to another due to control transfer instructions such as branch, procedure call and return.

The control flow in a program is represented by a control flow graph whose nodes represent basic blocks. An arc exists between two blocks if a control transfer between them is possible. An instruction i is said to be control dependent on a conditional branch instruction b (or the predicate associated with it), if the outcome of the conditional branch determines whether instruction i is executed. For the sequence:

```
b1:  if (t1 > 0) goto i2
i1:  t2 ← t3 + t4
i2:  t2 ← t3 − t4
```

In the preceding sequence, instructions use temporaries, or symbolic registers. Instruction i1 is executed only if the condition associated with b1 evaluates to false. Thus, instruction i1 is control dependent on b1, whereas instruction i2, which is executed irrespective of what b1 evaluates to, is control independent.

Finally, we informally define the notion of the live range of a variable or a temporary that is used in register allocation. A variable or a temporary is said to be defined when it is assigned a value (i.e., when the variable or temporary is the destination of an instruction). A variable is said to be used when it appears as a source operand in an instruction. The last use of a variable is a program point or instruction where the variable is used for the last time in the program, or used for the last time before it is redefined at a subsequent program point. The live range of a variable starts from its definition point and ends at its last use. A variable is said to be live during its live range. For the example code shown in Figure 17.1(a), the live ranges are depicted in Figure 17.1(b). The temporary t2 is defined in instruction i2 and its last use is at i4. Thus, the live range of t2 is from i2 to i4. We follow the convention that the live range ends just before the last use so that the last use instruction can reuse the same register as destination register.



```
i1 : t1 <-- ld x;
i2 : t2 <-- ld i
i3 : t3 <-- t2 - 4
i4 : t4 <-- t1 + t2
i5 : t5 <-- t3 * t4
i6 : x  <-- st t5
```

(a) Instruction Sequence (b) Live Ranges

FIGURE 17.1 Example of code sequence and live ranges.

When the live ranges of two variables do not overlap, they can share the same register. The number of variables that are live simultaneously indicates the number of registers that would be required. Informally, the number of simultaneously live variables is referred to as the *register pressure* at a given program point. For our example, assuming that no other variable is live into this basic block, the number of simultaneously live variables is 3 at instruction `i3`. When the number of available registers is less than the number of simultaneously live variables, the register allocation phase decides which variables or temporaries do not reside in registers. Load and store instructions are introduced, respectively, to load these temporaries from memory when necessary and spill them to memory locations subsequently. This is referred to as *register spills* and the load and store instructions are referred to as *spill code*.

17.2.2 Directed Acyclic Graph

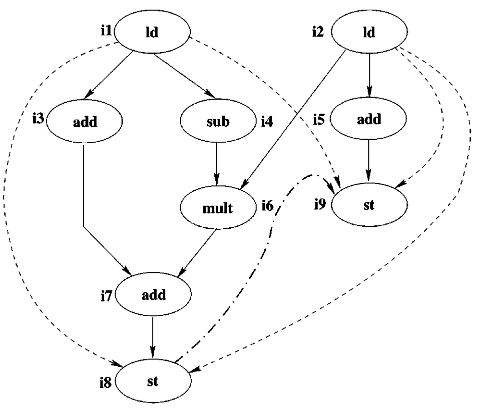
The data dependence among instructions in an instruction sequence can be represented by means of a dependence graph. The nodes of the dependence graph represent the instructions and a directed edge between a pair of nodes represents a data dependence. The dependence graph for instructions in a basic block is acyclic. Such a dependence graph is termed as *directed acyclic graph* (DAG). In a DAG, node v is said to be a successor (or immediate successor) of u, if an edge (u, v) exists. Similarly, node u is the predecessor (or immediate predecessor) of node v if an edge (u, v) exists in the DAG. We use the term *descendents* to refer to all nodes that can be reached from a node.

Next, let us discuss DAG construction for a basic block. A DAG can be constructed either in a forward pass or in a backward pass of the basic block [137]. In a forward pass method, at each step, a new node corresponding to an instruction in the sequence is added to the graph. By comparing against all previous instructions, the dependences among the instructions are determined, and appropriate dependence arcs between the corresponding nodes are added. This approach requires $O(n^2)$ steps, where n is the number of instructions. The dependences among instructions can also be determined using a table-building approach where a list of definitions and current uses are maintained [137]. The dependences checked for could be through general-purpose registers (or temporaries), memory locations and special purpose registers such as condition code registers.

Consider the example code given in Figure 17.2(a). The DAG for this example code sequence is shown in Figure 17.2(b). (Often a DAG is also drawn in a bottom-up manner [3]. A DAG drawn in this manner is shown later in Figure 17.7.) Because this code uses temporaries instead of register values, no anti- and output dependences occur through register variables or temporaries. The dependence

i1:	t1	←	load a
i2:	t2	←	load b
i3:	t3	←	t1 + 4
i4:	t4	←	t1 - 2
i5:	t5	←	t2 + 3
i6:	t6	←	t4 * t2
i7:	t7	←	t3 + t6
i8:	c	←	st t7
i9:	b	←	st t5

(a) Instruction Sequence



(b) DAG

FIGURE 17.2 Instruction sequence and its dependence graph.

arc from node `i2` to `i9` is due to antidependence on memory variable `b`. Because the dependences due to memory locations could not be analyzed accurately (i.e., the memory references could not be disambiguated) the antidependence arcs (`i1`, `i8`), (`i1`, `i9`) and (`i2`, `i8`) are also added. These antidependence edges are represented as broken lines in Figure 17.2. Finally, in the absence of memory disambiguation, also an output dependence exists from `i8` to `i9`. The output dependence arc is indicated by means of a dash-dot line.

Each arc (u, v) in the dependence graph is associated with a weight that is the execution latency of u. In a DAG, a node that has no incoming arc is referred to as a *source node*. Similarly, a node that has no outgoing arc is termed as a *sink node*. A DAG could have multiple source nodes and sink nodes. In our example DAG, nodes `i1` and `i2` are source nodes and `i8` and `i9` are sink nodes. For convenience, it is typical to add a fictitious source node, and edges from this node to every other node in the DAG are introduced. This fictitious node is henceforth referred to as the source node in the DAG. Similarly, there is a fictitious sink node, referred to as the sink node. Edges are added from each node in the DAG to the sink node. A weight 0 is associated with each of these newly introduced edges from the source node or to the sink node. To avoid clutter in Figure 17.2, we do not normally show the fictitious source and sink nodes and the associated edges.

A path in a DAG is said to be a *critical path* if the sum of the weights associated with the arcs in this path is (one of) the maximum among all paths. In our example, if the execution latency of each instruction is 1 cycle, then the path involving nodes `i1`, `i4`, `i6`, `i7` and `i8` is a critical path. Instructions in the critical path need to be given higher priority while scheduling, so that the execution time of the schedule is reduced.

17.2.3 Performance Metrics for Scheduling Methods

Typically, the objective of an instruction scheduling method is to reduce the execution time of a schedule, also referred to as the *schedule length*. Among the different scheduling methods, one that achieves the shortest schedule length is said to have the best performance or the best quality schedule. A schedule with the shortest schedule length is referred to as an *optimal schedule* (in terms of schedule length). Note that schedule length or execution time of a schedule is a static measure, because it refers to the execution of the schedule once. The overall execution time of a program is a dynamic measure that is also of interest.

Because obtaining an optimal schedule is an NP-complete problem [86, 111], the time taken to construct a schedule, referred to as the *schedule time*, is also an important performance metric.

Scheduling methods that have unacceptably long schedule time could not be used in a production compiler. Often, a scheduling method in a production compiler is expected to produce a reasonable quality schedule for a basic block within a few milliseconds. However, in certain application domains, such as digital signal processing (DSP), or in embedded applications, where the code is compiled once (at design time) and run subsequently, the schedule time is less of a constraint; in these applications, the compilation process itself may take several hours.

Apart from schedule length and schedule time, a performance metric that is of interest in an instruction scheduling method is the register pressure of the constructed schedule. Because schedules with higher register pressure are likely to incur more register spills, these spills, in turn, may increase the schedule length. Thus, beside having a lower execution time, a schedule having lower register pressure is often preferred. Many global scheduling methods, which schedule instructions beyond basic blocks, often cause an increase in the code size. Hence, code size of the scheduled code is another metric that is of interest when comparing different schedules. Code size is especially important in embedded applications, where an increase in code size increases the on-chip program memory; this, in turn, increases the system cost.

Finally, in embedded systems [114], power dissipated or energy consumed by the schedule is critical. In fact, schedules that consume lower power without incurring significant performance penalty, in terms of execution time, are often preferred in embedded applications.

The initial sections of this chapter focus on instruction scheduling methods for high performance, where schedule length is the main performance metric. Subsequently, in Section 17.6 we discuss issues relating to register pressure. Finally, in Section 17.7 we discuss instruction scheduling methods for a specific application domain, namely, DSP, and for low-power embedded applications, where code size, power and performance are important.

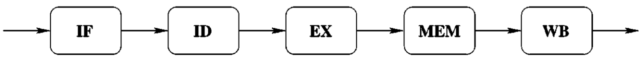
17.3 Instruction Scheduling for Reduced Instruction Set Computing Architectures

In this section we discuss early instruction scheduling methods proposed for handling pipeline stalls. First, we present a simple, generic architecture model and the need for instruction scheduling in this architecture model. In Section 17.3.2, we present the instruction scheduling method due to Gibbons and Muchnick [50] in detail. An alternative approach that combines register allocation and scheduling for pipeline stalls is discussed in Section 17.3.3. Finally, a brief review of other instruction scheduling methods is presented in Section 17.3.4.

17.3.1 Architecture Model

In a simple RISC architecture instructions are executed in a pipelined manner. Instruction execution in a simple 5-stage RISC pipeline is shown in Figure 17.3. Briefly, the instruction fetch (IF) stage is responsible for fetching instructions from memory. Instruction decode and operand fetch takes place in the decode (ID) stage. In the execute stage (EX), the specified operation is performed; for memory operations, such as load or store, address calculation takes place in this stage. The memory stage (MEM) is for load and store operations. Finally, in the write-back (WB) stage the result of an arithmetic instruction or the value loaded from memory for a load instruction is stored in the destination register.

Let instruction `(i+1)` be dependent on instruction `i` (i.e., `(i+1)` reads the result produced by instruction `i`. It can be seen that instruction `i+1` may read the operand value (in ID stage) before instruction `i` completes, that is, before `i` finishes the WB in the destination register. This dependence should cause the execution of instruction `(i+1)` to stall until instruction `i` writes the result, to ensure correct program behavior. This is known as a *data hazard* [66].



(a) A Simple Pipeline

$\longrightarrow$ time

	t	$t+1$	$t+2$	$t+3$	$t+4$	$t+5$	$t+6$	$t+7$
i	IF	ID	EX	MEM	WB			
i+1		IF	ID	EX	MEM	WB		
i+2			IF	ID	EX	MEM	WB	
i+3				IF	ID	EX	MEM	WB
i+4					IF	ID	EX	MEM

(b) Pipelined Execution

FIGURE 17.3 Instruction execution in a five-stage pipeline.

Another situation that may warrant stalls in the pipeline is due to control hazards [66]. If instruction `i` is a control transfer instruction, such as a conditional branch, unconditional branch, subroutine call or return instruction, then the subsequent instruction to be fetched may not be the instruction immediately following `i`. The location, or target address, from where the next instruction has to be fetched is not known until instruction `i` completes execution. Thus, fetch, decode and execution of subsequent instructions should be stalled until the control transfer instruction is complete.

In either situation (i.e., when we have a data or control hazard), the pipeline needs to be stalled to ensure correct program execution. The pipeline may include hardware support, referred to as *pipeline interlock*, to detect such occurrences and stall the subsequent instructions appropriately [66]. With a pipeline interlock, the dependent instruction and all subsequent instructions are stalled for a few cycles. The number of stall cycles required depends on the latency of instruction `i` and pipeline implementation, as well as whether other hardware mechanisms, such as result forwarding or bypassing, exist [66]. Pipeline forwarding reduces the number of stalls required. In a typical pipeline only certain pairs of consecutive instructions cause such a stall, and typically these stalls are for one or two cycles, except in cases where there is a dependency from either a multicycle operation such as a floating point multiply or a memory load that causes a data cache miss.

In processors that do not have pipeline interlocks, for example, in the MIPS R2000 processor [76], either the compiler or the programmer has to explicitly introduce no operation (NO-OP) instructions to ensure correct program execution. Alternatively, either the compiler or the programmer could reorder instructions, preserving data dependences, such that dependent instructions appear a few instructions apart. Such reordering would be useful for architectures with or without pipeline interlock hardware because it avoids pipeline stalls. It is easy to see how instruction reordering is useful in avoiding stalls due to data dependence. For control hazard stalls, reordering is useful only if the pipeline supports delayed branching, a common feature in most of the RISC pipelines [66]. Under delayed branching, a few (typically one or two) instructions following the branch are executed irrespective of the control transfer. The instructions following a branch or control transfer instruction are said to occupy the branch delay slots. The instructions that appear in the delay slot must preserve both control and data dependences. If such instructions cannot be found, the delay slots should be filled with NO-OP instructions.

17.3.2 Simple Instruction Scheduling Method

Optimal instruction scheduling to minimize the number of stalls under arbitrary pipeline constraints is known to be an NP-complete problem [63, 86, 111]. Several heuristic methods have been proposed in the literature [50, 63, 64, 111, 118, 151]. All these methods deal with instruction reordering within a basic block. We shall discuss two of the methods (due to Gibbons and Muchnick [50] and Proebsting and Fischer [118]) in detail and compare the rest.

The method proposed by Gibbons and Muchnick [50] assumes that (1) there must be 1 cycle stall between a load and an immediately following dependent instruction (which uses the loaded value), and (2) the architecture has hardware interlocks. Thus, the goal of the scheduling method is to reduce the pipeline stalls as far as possible; it is neither mandatory to remove all the stalls nor necessary to insert NO-OPS where stalls could not be avoided.

As mentioned earlier, the instruction reordering must preserve the data dependences present in the original instruction sequence. The dependences among instructions in a basic block are represented by means of a DAG.

Consider the evaluation of a simple statement

$$d = (a + b) \cdot (a + b - c)$$

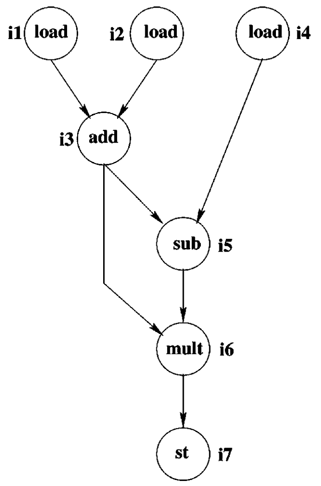
The code sequence to compute the expression on a RISC architecture is shown in Figure 17.4(a). The DAG for the sequence of instructions is shown in Figure 17.4(b). The given instruction order incurs two stall cycles: one at instruction `i3` because `i3` immediately follows a dependent load `i2`, and another at instruction `i5` that immediately follows the dependent load instruction `i4`. We shall now discuss how the Gibbons and Muchnick method obtains an instruction schedule with reduced number of stalls while preserving program dependences.

If instructions in the basic block are scheduled in the topological order of the DAG, then the dependences are preserved. An instruction is said to be ready if all its immediate predecessors in the DAG have been scheduled. Ready instructions are kept in a ReadyList. Among the instructions from the ReadyList, the best instruction is selected based on the following two guidelines:

1. An instruction that will not interlock with the one just scheduled
2. An instruction that is most likely to cause interlock with the instructions after it

Whereas the first guideline tries to reduce the stalls, the second guideline attempts to schedule early those instructions that may cause stalls, so that there is possibly a wider choice of instructions to

i1:	r1	←	load a
i2:	r2	←	load b
i3:	r3	←	r1 + r2
i4:	r4	←	load c
i5:	r5	←	r3 - r4
i6:	r6	←	r3 * r5
i7:	d	←	st r6



(a) Sample Code Sequence (b) DAG

FIGURE 17.4 Instruction sequence for $d = (a + b) \cdot (a + b - c)$ and its DAG.

```
Input: The DAG for the basic block
Output: The reordered instruction sequence.

Form the ReadyList of instructions by including all source nodes
while (there are instructions to be scheduled in the DAG) do
{
  choose a ready instruction based on guidelines (1) and (2)
    and also based on the static heuristics (1) -- (3);
  schedule the instruction, and remove it from ReadyList;
  add newly enabled instructions to ReadyList;
}
```

FIGURE 17.5 Gibbons–Muchnick scheduling method.

i1:	r1	←	load a
i2:	r2	←	load b
i4:	r4	←	load c
i3:	r3	←	r1 + r2
i5:	r5	←	r3 − r4
i6:	r6	←	r3 * r5
i7:	d	←	st r6

FIGURE 17.6 Schedule for the instruction sequence in Figure 17.4.

follow them. In addition to these guidelines, the method uses the following three heuristics, applied in the specified order, to select the next instruction to be scheduled:

1. Choose an instruction that causes a stall with any of its immediate successors in the DAG; this is somewhat similar to preceding guideline 2.
2. Choose an instruction that has the largest number of immediate successors in the DAG because this can potentially make many successor instructions ready.
3. Schedule an instruction that is farthest from the sink node in the DAG. This enables the schedule process to be balanced among various paths toward the sink node.

After scheduling each instruction, the list of ready instructions is updated including any successor instruction that has now become ready. The scheduling process proceeds in this way until all instructions in the basic block are scheduled. The scheduling algorithm is shown in Figure 17.5. The schedule generated by the preceding method for the example is depicted in Figure 17.6. This schedule incurs no stalls, because none of the load instructions is immediately followed by a dependent (arithmetic) instruction.

The worst-case complexity of the instruction scheduling method is $O(n^2)$. This happens in the degenerated case when all remaining instructions are in the ready list at each time step.

17.3.3 Combined Code Generation and Register Allocation Method

Proebsting and Fischer propose a linear time code scheduling algorithm, which integrates code scheduling and register allocation for a simple RISC architecture [118]. The scheduling method, known as the delayed load scheduling (DLS) method, assumes that the leaf nodes are memory loads. It produces optimal code, in terms of both the schedule length and the number of registers used, for a restricted architecture when the delay stalls due to load instructions are one cycle and when the

DAG is a tree. The method produces an efficient, near-optimal, schedule for the general case (i.e., when the delay is greater than one or when the dependence graph is a DAG).

The DLS method is an adaptation of the Sethi–Ullman (SU) method [3, 130] for generating code (instruction sequence) for basic blocks from its DAG representation. To understand the DLS method, first we shall explain the SU method with the help of an example. Unlike the RISC instruction scheduling method discussed in Section 17.3.2, the SU and DLS methods are applied at the time of code generation. These methods are applied to low-level intermediate form, typically the three-address code, and can produce code sequence with register allocation. Hence, these methods can be considered as integrated methods for code generation and register allocation.

17.3.3.1 Sethi–Ullman Method

The objective of the SU method is to generate a code sequence that either has minimum (sequence) length or uses the minimum number of temporaries. The SU method does not deal with pipeline stalls and hence may schedule a dependent instruction immediately after the load instruction on which it is dependent. Further, the SU method considers basic blocks with no live-in registers (i.e., all values are available in memory). Hence, the DAG representation of the basic block consists of leaf nodes that correspond to memory values must be loaded in registers (through load instructions) to perform any operation on them. The interior nodes of a DAG are all arithmetic operations.

Let us once again consider the basic block for the statement:

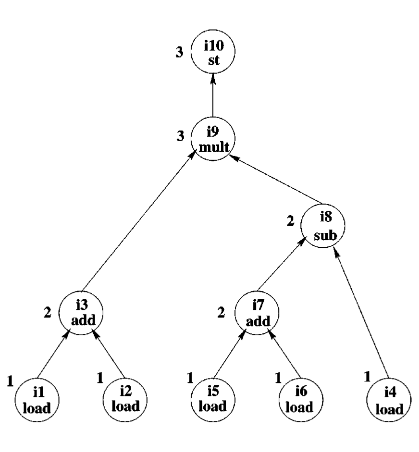
$$d = (a + b) \cdot (a + b - c)$$

The 3-address code for the basic block is shown in Figure 17.7(a). This code sequence uses temporaries and is unoptimized. In the three-address code, the evaluation of $(a + b)$ takes place twice. This would be eliminated by common subexpression elimination optimization. Hence, we call the three-address code given here unoptimized code. Without common subexpression elimination, the DAG for the basic block is a tree, as shown in Figure 17.7(b).

The code sequence shown in Figure 17.8(a) has a higher register pressure. A register allocated code for this sequence requires 4 registers. If the architecture has fewer than four registers, then a few values need to be spilled and reloaded at subsequent points in the computation. The spill loads and stores can increase the length of the code sequence. Let us now describe how the SU method generates a code sequence from the DAG that minimizes the length of the code sequence or the number of registers used. The method finds the optimal solution when the DAG is a tree. The

i1:	t1 ← load a
i2:	t2 ← load b
i3:	t3 ← t1 + t2
i4:	t4 ← load c
i5:	t5 ← load a
i6:	t6 ← load b
i7:	t7 ← t5 + t6
i8:	t8 ← t7 - t4
i9:	t9 ← t3 * t8
i10:	d ← st t9

(a) 3-Address Code



(b) Expression Tree

FIGURE 17.7 3-address code and expression tree for $d = (a + b) * (a + b - c)$.

i1:	r1	←	load a
i2:	r2	←	load b
i3:	r1	←	r1 + r2
i4:	r2	←	load c
i5:	r3	←	load a
i6:	r4	←	load b
i7:	r3	←	r3 + r4
i8:	r2	←	r3 − r2
i9:	r1	←	r1 ∗ r2
i10:	d	←	st r1

(a) Code Sequence using 4 Registers

i5:	r1	←	load a
i6:	r2	←	load b
i7:	r1	←	r1 + r2
i4:	r2	←	load c
i8:	r1	←	r1 − r2
i1:	r2	←	load a
i2:	r3	←	load b
i3:	r2	←	r2 + r3
i9:	r1	←	r1 ∗ r2
i10:	d	←	st r1

(b) Optimal Code Sequence with 3 Registers

FIGURE 17.8 Register allocated code sequences.

optimal solution for the DAG is shown in Figure 17.8(b). Let us describe the method for the special case when the DAG is a tree.

The SU method has two phases: the first phase assigns a label for each node of the tree, and the second phase is a tree traversal that generates code as the nodes and subtrees of the DAG are visited. Intuitively, the label of a node represents the number of registers or temporaries that are needed to compute the subtree rooted under the node. In the tree traversal phase, a subtree is completely traversed before proceeding to other sibling subtrees. The traversal order among the siblings is based on the labels, and the node with a larger label is visited first. Thus, the method generates code first for the subtree that requires more registers.

The labeling phase traverses the tree in postorder, visiting all children before visiting a node. The label assigned for a node n corresponding to a binary operator is given by:

$$label(n) = \begin{cases} \max(l_1, l_2) & \text{if } l_1 \neq l_2 \\ l_1 + 1 & \text{if } l_1 = l_2 \end{cases}$$

where l_1 and l_2 are the labels assigned to the left and right children of n. The label assignments for the nodes are shown in Figure 17.7(b).

The label assigned to a nonbinary operator n, having k children, with $l_1, l_2, \ldots, l_k$ as the labels of the children nodes arranged in nonincreasing order, is:

$$label(n) = \max_{1 \le i \le k} (l_i + i - 1)$$

Intuitively, when there are r children with the same label l_i (i.e., requiring the same number l_i registers for computing the subtrees rooted under them), then $l_i + r - 1$ registers are required for computing the subtree rooted under the parent node.

Each leaf node is assigned the label value 1. For architectures supporting register–memory operands, only the leftmost child of each node that are leaf nodes need be assigned the label value 1. Other leaf nodes are assigned the label 0. The intuitive reasoning behind this is that the operand corresponding to the right child (in the case of a binary operator), which is a leaf node, can be used directly as a memory operand (without requiring it to be loaded in a register) in the instruction corresponding to the parent node. Whereas for RISC architectures where all arithmetic operators have only register operands, all leaf nodes that are memory locations must first be loaded in a register. Hence, they are assigned a label one.

Next we describe the second phase of the SU method, the generate code sequence phase. This phase traverses the labeled tree recursively, first generating the code for the child having the higher label. When the children have the same label value, they can be traversed in any order. The method maintains a register stack of the available registers. If the node visited is a leaf node (or a node having a label one), then the code, a load instruction, is generated for the node. The register on the top of the stack is used as the destination register. For machines that support register–memory operands, a load instruction is generated only for the leftmost child that has a label value one. The right child (assuming the parent node to be a binary operator) can be used directly as a memory operand when the code for the parent node is generated. Because we concentrate on RISC architectures with register–register operands in this discussion, we omit further details for complex instruction set computer (CISC) architectures. The interested reader is referred to [3, 130].

The code for an interior node is generated by emitting the corresponding instruction, with the source operands the same as the destination operand of the children nodes, and the destination register the same as that of the left child. In the special case that the DAG is a tree, while generating the code for the parent node, it is always possible to free the destination registers of the children nodes. This is because the value produced by a node is used only by its parent, and in a tree, each node has at most one parent. The destination registers of the right children can be freed and are pushed into the register stack so that they can be reused subsequently. The original algorithm also uses a swap operation on the register stack to ensure that a left child and its parents are evaluated on the same register. We leave out these details as well in our discussion. Finally, because the code is generated by traversing the tree recursively, generating code for the children nodes before generating the code for a parent node, the dependences are easily satisfied. For the tree used in our motivating example, the instruction sequence generated by the SU method is shown in Figure 17.8(b). This sequence uses three registers.

In the general case where the dependence graph is a DAG (and not a tree), the optimal code generation problem is known to be NP complete [2]. A near-optimal solution is obtained by partitioning the DAG into trees. A node having more than one parent is called a shared node. The partitioning is done in such a way that each root or shared node forms the root of the tree, with the maximal subtree that includes no shared nodes except as leaves. Shared nodes with more than one parent can be turned into as many leaves as necessary. More details can be found in [3, 130].

In the expression used in our example, $(a + b)$ is common to the left and right subtrees of the expression tree. More specifically, the subtrees rooted on `i3` and `i7` compute the same expression. On performing common subexpression elimination, one of these subtrees is eliminated, resulting in a DAG. By splitting the DAG into a set of subtrees, it is possible to obtain a code sequence using three registers.

17.3.3.2 Delayed Load Scheduling Method

Next we shall discuss the DLS method that integrates code generation and register allocation for pipelined architectures that incur delays [118]. As before, we shall first discuss the method when the DAG is a tree. The main idea behind the DLS method is to produce a canonical form of the instruction sequence. Suppose the sequence consists of L memory load instructions (referred to as *loads*) and $(L - 1)$ arithmetic operations[2] (referred to as *operations*) and uses R registers. Then the canonical form consists of R load instructions followed by an alternating sequence of $(L - R)$ $\langle$*operation, load*$\rangle$ pairs, followed by $(R - 1)$ operations. The canonical form is generated from the sequence generated by the SU method.

The DLS method is a three-phase method, starting with a labeling phase where the nodes in the expression tree are assigned the *minReg* value. The *minReg* value is the label given by the labeling phase of the SU method. The second phase, *order*, generates the relative order of loads and operations in two separate data structures. The ordering is accomplished by recursively generating the order for

[2]There are $(L - 1)$ (binary) operations in a binary tree with L leaf nodes.

`i5:`	`r1 ← load a`	
`i6:`	`r2 ← load b`	
`i7:`	`r1 ← r1 + r2`	`% 1 stall`
`i4:`	`r2 ← load c`	
`i8:`	`r1 ← r1 − r2`	`% 1 stall`
`i1:`	`r2 ← load a`	
`i2:`	`r3 ← load b`	
`i3:`	`r2 ← r2 + r3`	`% 1 stall`
`i9:`	`r1 ← r1 * r2`	
`i10:`	`d ← st r1`	

(a) Stalls in Sethi-Ullman Sequence

`i5:`	`r1 ← load a`
`i6:`	`r2 ← load b`
`i4:`	`r3 ← load c`
`i1:`	`r4 ← load a`
`i7:`	`r1 ← r1 + r2`
`i2:`	`r2 ← load b`
`i8:`	`r1 ← r1 − r3`
`i3:`	`r4 ← r4 + r2`
`i9:`	`r1 ← r1 * r4`
`i10:`	`d ← st r1`

(b) DLS Sequence with No Stalls

FIGURE 17.9 Instruction sequences with and without stalls.

the left and right subtrees, starting with the one that has the higher register requirement. The ordering phase is also similar to that in the SU method discussed in the previous subsection. The schedule phase also assigns registers for all the instructions as discussed in the SU method. The number of registers R used is one more than the *minReg* value of the root of the expression tree.

Finally, the schedule phase of the DLS method essentially generates the canonical order, listing R loads followed by an alternating sequence of ⟨*operation,load*⟩ pairs. The DLS method generates optimal instruction sequence without any pipeline stall except in two special cases. The two cases are (1) where the expression tree consists of a single load, or (2) where the expression tree consists of a single operation and two leaf nodes (load instructions).

Let us consider the example expression introduced in Section 17.3.3.1. As discussed earlier, an optimal schedule for this code considering no pipeline delays is shown in Figure 17.8(b). Then let us consider an architecture with delay $D = 1$ between a load and a dependent instruction. In this example, we have $L = 5$ loads, $(L - 1) = 4$ (binary) operations, and finally an unary operation (store). The sequence obtained from the SU method, causes 3 stall cycles, at instructions `i7`, `i8` and `i3`, as shown in Figure 17.9(a). The SU schedule uses three registers. To avoid the stalls completely, the DLS sequence uses one more register than in SU schedules; hence, $R = 3 + 1 = 4$. The sequence shown in Figure 17.9(b) is obtained from the DLS method that incurs no stall cycle. In this sequence, initially we have $R = 4$ loads, followed by an alternating sequence of $(L - R) = (5 - 4) = 1$ ⟨*operation, load*⟩ pair, followed by 4 arithmetic operations.[3] This sequence is optimal in terms of the number of execution cycles. It uses one more register than that used by the SU method, but completely avoids all stall cycles. It should be noted here that among the sequences that incur the lowest execution cycles, this sequence uses the minimum number of registers (i.e., no other instruction sequence incurs the minimum number of execution cycles and requires fewer registers).

The complexity of the DLS method is $O(n)$, as the labeling and ordering phases can be performed by traversing through the nodes once (bottom up), and the schedule phase visits each node exactly once. Not only is this superior to the $O(n^2)$ complexity of the instruction scheduling method due to Gibbons and Muchnick [50] (discussed in Section 17.3.2) but also it performs scheduling and register allocation together in a single framework. The DLS method also serves as an excellent heuristic when the dependence graph is an arbitrary DAG or when the delay is greater than one.

[3]In this sequence, due to the additional unary store operation, we have R, instead of $(R - 1)$, arithmetic operations at the end of the sequence.

Finally, recall that the DLS method requires that the leaf nodes be memory load operations. This precludes register variables, or live-in registers, in the expression tree. An extension that relaxes this constraint is presented in [83, 146].

17.3.4 Other Pipeline-Scheduling Methods

A heuristic pipeline-scheduling method that is performed during the code generation was implemented in the PL-8 compiler [10]. A major advantage of this method is that it performs code scheduling before register allocation and hence avoids false dependences. Hennessy and Gross describe a heuristic method that is applied after code generation and register allocation [63, 64]. This method uses a dependence graph representation that eliminates false dependences. However, to accomplish this, their method needs to check for deadlocks in scheduling. It uses a look-ahead window to avoid deadlock. The worst-case running time of this method is $O(n^4)$.

Finally, we briefly discuss filling the delay slot of a branch instruction. As mentioned earlier, if a processor supports delayed branching [66], then moving independent instructions in the branch delay slots helps to reduce the number of control hazard stalls. Some of the instruction scheduling methods discussed in this section (e.g., the Gibbons–Muchnick method [50] or the Hennessy–Gross method [63]) can be used to fill the branch delay slot with an useful instruction. It is most desirable to fill the branch delay slot with an instruction from the basic block that the branch terminates. Otherwise, an instruction from either the target block (of the branch) or the fall-through block, whichever is most likely to be executed, is selected to be placed in the delay slot. The selected instruction either occurs as a source node in both (target and fall-through) basic blocks, or has a destination register that is not live-in in the other block or has a destination register that can be renamed.

The instruction scheduling methods discussed in this section do not consider (functional unit) resource constraints. They merely try to reorder instructions to reduce the NO-OP instructions or the pipeline stalls needed to ensure correct program behavior. In contrast, the instruction scheduling methods to be discussed in the following section for VLIW and superscalar architectures take into consideration the resource constraints.

17.4 Basic Block Scheduling

Instruction scheduling can be broadly classified based on whether the scheduling is for a single basic block, multiple basic blocks, or control flow loops involving single or multiple basic blocks [121]. Algorithms that schedule single basic blocks are termed as *local-scheduling* algorithms, which is the topic of discussion in this section. Algorithms that deal with multiple basic blocks or basic blocks with cyclic control flow are termed as *global-scheduling* algorithms and are dealt with in Section 17.5. The term *cyclic scheduling* is used to refer to methods that schedule single or multiple basic blocks with cyclic control flow. Cyclic scheduling overlaps the execution of multiple instances of a static basic block corresponding to different iterations.

In this section we discuss local or basic block scheduling methods for VLIW and superscalar processors. First, we present the necessary preliminaries in the following section. In Section 17.4.2 we present the basic list-scheduling algorithm. Operation based instruction-scheduling methods are discussed in Section 17.4.3. An exact approach to obtain an optimal schedule using an integer linear programming formulation is presented in Section 17.4.4. Section 17.4.5 deals with resource usage models that are used in instruction-scheduling methods. We present a few case studies in Section 17.4.6.

17.4.1 Preliminaries

With the advent of multiple instruction issue processors, namely, superscalar processors [66, 75, 136] and VLIW architectures [46, 126], it has become important to expose instruction level parallelism at compile time. Both VLIW and superscalar architectures have a number of functional units and are capable of executing multiple independent operations in a single cycle. Hence, instruction scheduling for these architectures must identify the instructions that can be executed in parallel in the same cycle. In a VLIW architecture, the identification of independent instructions and their reordering to expose instruction level parallelism must be done at compile time. Multiple independent instructions and operations that can be issued in the same cycle should be packed in a single long word instruction for a VLIW architecture.

Superscalar processors provide hardware mechanisms to detect dependences between instructions at runtime and to schedule multiple independent instructions in a single cycle. *In-order* issue superscalar processors are capable of issuing multiple independent instructions in each cycle; however, once they encounter an instruction for which the source operands are not yet ready (i.e., the instruction producing the source operand has not completed execution), the dependent instruction as well as all future instructions are stalled until the dependent instruction becomes data ready. *Out-of-order* issue superscalar processors, on the other hand, are capable of issuing independent instructions, even beyond a dependent stalled instruction. In other words, they can issue instructions out of the order in which they appear in the program. Both in-order and out-of-order issue processors benefit by a runtime register renaming mechanism [136] that helps to eliminate false dependences (anti- and output dependences). Due to the hardware support available in superscalar processors, instruction reordering to expose ILP is not mandatory, although such a reordering would certainly benefit both in-order and out-of-order superscalar processors. This is especially the case for in-order issue superscalar processors.

The instruction schedule constructed for a VLIW or a superscalar processor must satisfy both dependence constraints and resource constraints. Dependence constraints ensure that an instruction is not executed until all the instructions on which it is dependent on are scheduled, and their execution are complete. Once again, dependences among instructions are represented by means of a DAG — because local instruction scheduling deals only with basic blocks, the dependence graph is acyclic. In our discussion we shall assume that register allocation is performed after instruction scheduling and that the DAG is constructed from an instruction sequence that uses temporaries (instead of registers) and hence avoids all anti- and output-dependence arcs for all nonmemory operations.

Resource constraints enforce that the constructed schedule does not require more resources (functional units) than available in the architecture. The resource usage model in a realistic instruction scheduling method needs to take into consideration the finite resources available in the architecture, the actual (nonunit) execution latencies incurred by some of the instructions and simple or complex resource usage patterns. In a simple resource usage pattern, each instruction uses one resource for a single cycle and thus multiple instructions scheduled on different cycles do not cause any structural hazard. With such a simple resource usage pattern, it is possible to schedule a new instruction in the functional unit in each cycle. Whereas when the resource usage pattern is complex, a single resource could be used for multiple cycles.

The resource usage model specifies the usage pattern of resources for different classes of instructions (such as integer, load/store, multiply, floating point (FP) add, FP multiply, and FP divide instruction classes) as well as the available functional units. A simple representation for resource usage is a reservation table, which is an $r \times l$ matrix, where l is the latency of the instruction and r is the number of stages in the functional unit [81]. An entry $R[r, t]$ is 1 if resource r is used t time steps after the initiation of an instruction in the functional unit and 0 otherwise. The resource usage pattern is simple when the functional unit is fully pipelined. A pipelined functional unit can initiate a new operation in every cycle. We defer discussion on complex resource usage and more sophisticated resource usage models to Section 17.4.5.

The resource requirements of a schedule can be modeled using a global reservation table (GRT), an $M \times T$ matrix, where M is the number of resources (including all stages of all functional units as well as other resources such as memory ports) for which contention must be explicitly modeled in the schedule, and T is an upper bound on the length of the schedule (i.e., the number of time steps taken by the schedule). An entry $GRT[r, t]$ is 1 if resource r is used at time step t in the current schedule, and 0 otherwise. As a schedule is constructed, the GRT represents the resource requirements of the partial schedule of instructions that are already scheduled. Any new instruction scheduled should not cause a resource contention, i.e., is two or more instructions requiring the same resource at the same time step, with the partial schedule. Resource contention is checked by a contention query model that checks whether scheduling an instruction at time step t causes any conflicting resource requirements with instructions that are already scheduled. We shall return to the contention query model in greater detail in Section 17.4.5.

An instruction-scheduling method assumes a fixed execution latency for each instruction. However, this does not cover variations in latency caused by events such as cache misses. A schedule constructed with an underestimated or optimistic value of the latency may cause unnecessary stalls when a cache miss occurs. This would be the case even though enough parallelism may be in the basic block to hide the latency. On the other hand, when a pessimistic latency value is used, the schedules are unnecessarily stretched, even for cache hit cases. Balanced scheduling [78] and improved balanced-scheduling methods [94] generate schedules that can adapt more readily to uncertainties in memory latencies. These methods use a load latency estimate that is based on the number of independent instructions that are available in the basic block to mask the load latency. All other instruction-scheduling methods assume an optimistic estimate for the execution latency, which is followed in our discussion in the rest of this chapter.

17.4.2 List-Scheduling Method

The list-scheduling method schedules instructions from time step zero, starting with the source instructions in the basic block. At each time step t, it maintains a list of ready instructions (ReadyList) that are data ready, i.e., instructions whose predecessors have already been scheduled and would produce the result value in the destination register by time t. List scheduling is a greedy heuristic method that always schedules a ready instruction in the current time step whenever no resource conflict occurs.

The list-scheduling algorithm is similar to the Gibbons and Muchnick instruction-scheduling method discussed in Section 17.3.2, except that multiple instructions can be scheduled in the same step. The resource requirements of scheduled instructions are maintained in the GRT. At each time step, among the set of ready instructions from ReadyList, instructions are scheduled one at a time based on certain priority ordering of instructions. The priorities assigned to different instructions are decided by heuristics. Different list scheduling methods differ in the way they assign priorities to the instructions. We shall discuss some of the important heuristics that have been used in various instruction scheduling methods in Section 17.4.2.1. It should be noted here that the priorities assigned to instructions could be either static (i.e., assigned once and remains constant throughout the instruction scheduling), or dynamic (i.e., change during the instruction scheduling), and hence require that the priorities of unscheduled instructions be recomputed after scheduling each instruction. Although the basic list-scheduling algorithm discussed later assumes a static priority, it can easily be adapted for a heuristic that assigns dynamic priority values.

After scheduling all instructions that are in the ReadyList that do not cause a resource conflict, the time step is incremented by one. Any instruction that has now become data ready is included in the ReadyList. The ReadyList is sorted on decreasing order of priority. The scheduling process

```
Input: DAG
Output: Instruction Schedule
AssignPriority (DAG); /* assigns priority to each instruction
                  in the DAG based on the priority policy */
ReadyList = source nodes in the DAG;
timestep = 0;
while (there exists an unscheduled instruction in the DAG) do
{
  Sort ReadyList in non-decreasing priority order;
  while (not all instructions in ReadyList are tried)
  {
    pick next instruction i from ReadyList;
    check for resource conflict;
    if (instruction can be scheduled)
    {
      update GRT (i,timestep);
      remove instruction i from ReadyList;
    }
  }
  increment timestep by 1;
  add instructions that have now become data ready in ReadyList;
}
```

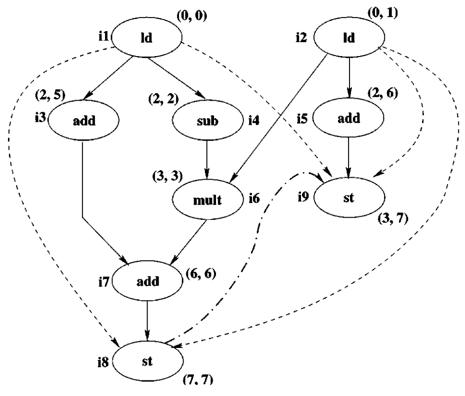
FIGURE 17.10 Generic list scheduling algorithm.

```
c = (a+4)+(a-2)*b;
b = b+3;
```

(a) High-Level Code

i1:	t1	←	load a
i2:	t2	←	load b
i3:	t3	←	t1 + 4
i4:	t4	←	t1 - 2
i5:	t5	←	t2 + 3
i6:	t6	←	t4 * t2
i7:	t7	←	t3 + t6
i8:	c	←	st t7
i9:	b	←	st t5

(b) 3-Address Code



(c) DAG with *(Estart, Lstart)* Values

FIGURE 17.11 Example code, three-address representation and DAG.

continues in this way until all the instructions are scheduled. The complete algorithm is shown in Figure 17.10.

We illustrate the list-scheduling method with the help of a simple example. Consider the code sequence and its 3-address representation shown in Figures 17.11(a) and 17.11(b), respectively. Its DAG is depicted in Figure 17.11(c). Assume that the target architecture consists of two integer functional units, which can execute integer instructions as well as load and store instructions, and one multiply and divide unit. All functional units are fully pipelined. Further, assume that the execution latencies of `add`, `mult`, `load` and `store` instructions are one, three, two and one cycles, respectively. These latency values also imply that there should be one stall cycle between a `load` and a dependent instruction and two stall cycles between a `mult` and a dependent instruction. No stall cycles are required between an `add` and a dependent instruction. Further, the path `i1` → `i4` → `i6` → `i7` → `i8` is the critical path in the DAG.

Time	Int. Unit 1	Int. Unit 2	Mult. Unit
0	t1 ← load a	t2 ← load b	
1			
2	t4 ← t1 − 2	t5← t2 + 3	
3	t3 ← t1 + 4	b ← st t5	t6 ← t4 ∗ t2
4			
5			
6	t7 ← t6 + t3		
7	c ← st t7		

FIGURE 17.12 Schedule for the example code in Figure 17.11.

A list schedule for the example code is shown in Figure 17.12. Instructions that are on the critical path are scheduled at their earliest possible start time to achieve this schedule whose length is eight. Note that if the 2 `add` instructions (`i3` and `i5`) are scheduled ahead of `sub` instruction in time step 2, it would have delayed the execution of instructions on the critical path, namely, `i4`, `i6`, `i7` and `i8` instructions and hence would have increased the schedule length. To achieve schedules that require fewer execution cycles, the scheduling method should use an efficient heuristic that gives priorities to instructions on the critical path.

The schedule is presented as a parallel schedule, as shown in Figure 17.12, to a VLIW architecture, where multiple instructions that can be executed in the same cycle are packed into a single long word instruction. For a superscalar architecture, the parallel instructions in each cycle are linearized in a simple way, say from left to right. It is shown in [133] that the linearizing order could have a performance impact on out-of-order issue superscalar processors [66, 136]. We defer a discussion on the linearization order to Section 17.6.3.3.

It has been shown that the worst-case performance of a list-scheduling method is within twice the optimal schedule [86, 111]. That is, if T_{list} is the execution time of a schedule constructed by a list scheduler, and T_{opt} is the minimal execution time that would be required by any schedule for the given resource constraint, then T_{list}/T_{opt} is less than two. The quality of list scheduling can degrade and approach the above bound as the number of resources increases or the maximum of the latencies of all instructions increases.

In this example, the list scheduling method uses a greedy approach, trying to schedule instructions as soon as possible. If enough resources exist, then in the list scheduling method each instruction would get scheduled at the earliest start time (*Estart*) possible. Further, the list scheduling method described earlier makes a forward pass of the DAG, starting from a source node. It is possible to have a backward pass list scheduling method that schedules instructions starting from the sink node to the source node. Whereas forward pass schedulers attempt to schedule an instruction at the earliest time possible, backward pass schedulers typically attempt to schedule each instruction as late as possible. The instruction scheduler of the GNU C compiler (version 2) [141] and the local instruction scheduler of the Cydra 5 compiler [34] are backward pass schedulers whereas the scheduler in the IBM XL compiler family makes a forward pass [151].

17.4.2.1 Heuristics Used

The list-scheduling method uses heuristics to assign priorities to instructions. These priorities are used in selecting the ready instructions for scheduling in each time step. This section briefly discusses some of the heuristics used. An extensive survey and a classification of the various heuristics used in instruction-scheduling methods are presented in [137].

A commonly used heuristic is based on the maximum distance of a node to the fictitious sink node. The maximum distance (MaxDistance) of the sink node to itself is zero. The MaxDistance of a node u is defined as:

$$\text{MaxDistance}\ (u) = \max_{i=1,\dots,k} (\text{MaxDistance}\ (v_i) + \text{weight}\ (u, v_i))$$

where $v_1, v_2, \dots, v_k$ are the successors of node u in the DAG. MaxDistance is calculated using a backward pass on the DAG, and is a static priority. Priority is given to nodes having larger MaxDistance. A variation of this heuristic is to consider the maximum distance to the sink node, where distance is measured in terms of the path length (number of edges in the path), and not as the sum of execution latencies of the instructions in the path.

Another heuristic used in list scheduling method is to give priority to instructions that have larger execution latency. The maximum number of children heuristic gives priority to instructions that have more successors. A refinement of this is to consider only successors for which this instruction is the only parent. An alternative is to consider not only the immediate successors but all the descendents of the instruction. In all these cases, giving higher priority to instructions having larger number of descendents may enable a larger number of instructions to be added to the ReadyList. All these heuristics are static in nature and are computationally inexpensive compared with dynamic heuristics.

Many list scheduling methods give higher priority to instructions that have the smallest Estart. The Estart value of the fictitious source node is zero. The Estart value of an instruction v is defined as:

$$\text{Estart}\ (v) = \max_{i=1,\dots,k} (\text{Estart}\ (u_i) + \text{weight}\ (u_i, v))$$

where $u_1, u_2, \dots, u_k$ are the predecessors of v. Similarly, priorities can be given to instructions having the smallest latest start time (*Lstart*), which is defined as:

$$\text{Lstart}\ (u) = \min_{i=1,\dots,\ k} (\text{Lstart}\ (v_i) - \text{weight}\ (u, v_i))$$

where $v_1, v_2, \dots, v_k$ are the successors of u. The a start value or of the sink node is set the same as its Estart value. Estart and Lstart are computed using a forward or backward pass of the DAG. The Estart and Lstart values of the instructions in our example DAG are also shown in Figure 17.11(c).

The difference between Lstart(u) and Estart(u), referred to as *slack*, or *mobility*, can also be used to assign priorities to the nodes. Instructions having lower slack are given higher priority. Instructions that are on the critical path may have a slack zero, and hence get priority over instructions that are on the off-critical path. Instructions on the critical path of the DAG shown in Figure 17.11, namely, `i1`, `i4`, `i6` and `i8`, have a slack 0, indicating that there is no slack or freedom in scheduling them. Many list scheduling methods use Estart and Lstart as static measures, although their values can be calculated after scheduling instructions in each step. Instructions that are scheduled in the current time step may affect the Estart (or Lstart) values of successor (or predecessor) nodes, and hence these values need to be recomputed in each time step. Slack can also be treated as a static or a dynamic measure. The list-scheduling method described in [111] uses a combination of weighted path length and Lstart values.

A heuristic based on computing a force metric is used in scheduling data path operations in behavioral synthesis [115]. The self force of each instruction u at time step t reflects the effect of an attempted time step assignment t to instruction u on the overall instruction concurrency. The force is positive if the time step causes an increase in the concurrency and negative otherwise. The predecessor and successor forces refer to the effect of scheduling an instruction at a time step on its predecessors and successors, respectively. The force metric is a product, $K \cdot x$, where K represents the extent of concurrency of each type of instruction at a given time step t, and x is a function of the slack of the instruction u. Instructions having the lowest force are given the highest priority.

For further details on calculating the force metric, the reader is referred to [115]. Although the objective of the force-directed scheduling method in behavioral synthesis is to minimize the resources while minimizing the execution time, it still obtains an efficient schedule for the instructions. In this sense, the force-directed heuristic may serve as a useful heuristic in an instruction scheduling method as well.

Finally, priorities can be given to instructions that define fewer registers (or temporaries) defined (i.e., instructions that start fewer new live ranges). Intuitively, it is advantageous to defer the scheduling of an instruction that defines new temporaries to a later time step, because it would defer the increase in register pressure. Such a heuristic is typically used in prepass scheduling methods. Likewise, it is advantageous to give higher priority to instructions that end the live range of a variable or temporary. Version 2 of the GNU C compiler uses this heuristic [141].

17.4.3 Operation Scheduling Method

Whereas a list scheduling method schedules instructions on a cycle-by-cycle basis, an operation-scheduling method attempts to schedule instructions one after another, trying to find the first time step at which each instruction can be scheduled. Operation-driven schedulers sort the instructions in the DAG in a topological order, giving priority to instructions that are on the critical path [128]. An operation scheduling method could be nonbacktracking or backtracking. We discuss a backtracking method next.

In a backtracking operation scheduling method, at each iteration, an instruction i is selected, based on certain priority function. An attempt is made to schedule instruction i at time t that is between Estart (i) and Lstart (i) and that does not cause a resource conflict. The scheduling of an instruction at time step t may affect the Estart and Lstart values of other unscheduled instructions. If dynamic priority is used to select the instruction, then the priorities of unscheduled instructions are recomputed.

If no time step t is between the Estart (i) and Lstart (i) at which the instruction can be scheduled, an already scheduled instruction j that has conflicting resource requirement with this instruction is descheduled, making room for this instruction. The descheduled instruction j is put back in the list of unscheduled instructions and is scheduled subsequently. For the method not to get into a loop where a set of instructions deschedule each other, a threshold on the number of descheduled instructions is kept. When this threshold exceeds, the partial schedule is discarded, and new Lstart values for instructions are computed by increasing the Lstart value of the sink node.

17.4.4 Optimal Instruction Scheduling Method

The resource-constrained, instruction scheduling problem is known to be NP complete [86, 111]. The instruction scheduling problem has been formulated as an integer linear programming problem [9, 23, 28]. Such an approach is attractive for the evaluation of (performance) bounds that can be achieved by any heuristic method. Also, more recently Wilken, Liu and Heffernan [155] have shown that optimal schedules can be obtained in a reasonable time even for large basic blocks; and hence, such an optimal scheduling method can potentially be applied to even production compilers.

In this section we illustrate an integer linear programming formulation for resource-constrained basic block instruction scheduling. We assume a simple resource model in which all functional units are fully pipelined. Altman, Govindarajan and Gao [7] present methods for modeling functional units with complex resource usage pattern in integer linear programming formulation, in the context of software pipelining — an instruction scheduling method for iterative computation [70, 84, 121, 122, 125]. Our discussion considers an architecture consisting of functional units of different types (e.g., Integer ALU, Load/Store Unit, FP Add Unit and FP Mult/Divide Units), and the execution latency of instructions in these functional units can be different. Further, we assume that there are R_r instances in functional unit type r.

Let σ_i represent the time step at which instruction i is scheduled, and $d_{(i,j)}$ represent the weight of edge (i, j). To satisfy dependence constraints, for each arc (i, j) in the DAG:

$$\sigma_j \geq \sigma_i + d_{(i,j)} \tag{17.1}$$

To represent the schedule in a form that can be used in the integer linear programming formulation, a matrix K of size $n \times T$ is used, where n is the number of instructions or nodes in the DAG, and T is an estimate of the (worst-case) execution time of the schedule. Typically, T is the sum of the execution times of all the instructions in the basic block. Note that T is a constant and can be obtained from the DAG. An element of K, say $K[i, t]$, is one if instruction i is scheduled at time step t and zero otherwise. The schedule time σ_i of instruction i can be expressed using K as:

$$\sigma_i = k_{i,0} \cdot 0 + k_{i,1} \cdot 1 + \cdots + k_{i,T-1} \cdot (T-1)$$

This can be written in the matrix form for all σ_i's as:

$$\begin{bmatrix} \sigma_0 \\ \sigma_1 \\ \vdots \\ \sigma_{n-1} \end{bmatrix} = \begin{bmatrix} k_{0,0} & k_{0,1}, & \cdots & k_{0,T-1} \\ k_{1,0} & k_{1,1} & \cdots & k_{1,T-1} \\ \vdots & \vdots & \vdots & \vdots \\ k_{n-1,0} & k_{n-1,1} & \cdots & k_{n-1,T-1} \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 1 \\ \vdots \\ T-1 \end{bmatrix} \tag{17.2}$$

To express that each instruction is scheduled exactly once within the schedule, the constraint:

$$\sum_t k_{i,t} = 1, \quad \forall i \tag{17.3}$$

is included in the formulation.

Finally, the resource constraint that no more than R_r instructions are scheduled in any time step, where R_r is the number of functional units of type r, can be enforced through the equation:

$$\sum_{i \,\in\, F(r)} k_{i,t} \leq R_r, \quad \forall t \text{ and } \forall\, r \tag{17.4}$$

where $F(r)$ represents the set of instructions that can be executed in functional unit type r.

The objective function is to minimize the execution time or schedule length. That is:

$$\text{minimize } (\max_i(\sigma_i + d_{(i,j)}))$$

To express this in the linear form, we introduce

$$z \geq \sigma_i + d_{(i,j)}. \tag{17.5}$$

Now, the objective is to minimize z subject to Equations (17.1) to (17.5).

17.4.5 Resource Usage Models

This subsection deals with different resource usage models used in instruction scheduling. First, we motivate the need for sophisticated resource usage models. In the subsequent subsection, we review some of the existing resource usage models.

17.4.5.1 Motivation

Modern processors implement very aggressive arithmetic and instruction pipelines. With aggressive multiple instruction issue, structural hazard resolution in modern processors is expected to be more complex. Furthermore, in certain emerging application areas, such as mobile computing or space vehicle on-board computing, the size, weight and power consumption may put tough requirements on the processor architecture design; this, in turn, may result in more resource sharing. All these lead to pipelines with more structural hazards. With such complex resource usage, the scheduling method must check and avoid any structural hazard (e.g., contention for hardware resources by instructions). Such a check for resource contention is done by a contention query module [38].

The contention query module uses a resource usage model that specifies the resource usage patterns of various instructions in the target architecture. The contention query module answers the query: "Given a target machine and a partial schedule, can a new instruction be placed in time slot t without causing any resource conflicts?" Because a resource contention check needs to be performed before scheduling every instruction and for each instruction several timesteps of the schedule need to be examined, a significant part of the schedule time is spent in the contention query module. This is especially the case when the resource usage pattern is complex due to many structural hazards. Thus, an efficient resource usage model is critical for reducing the contention check time in instruction scheduling.

Portability and preciseness are two important aspects in choosing a resource model. Compilers designed to support a wide range of processors usually define the machine details to the scheduler in a form that can be easily modified when porting the compiler across different processors [60]. A model portable can only approximately model the complex execution constraints that are typical in modern-day superscalar and VLIW processors. Precise modeling of machine resources is important to avoid some of the stalls in the pipeline. Precise modeling of resource usages often involves a very low level representation of the machine description that is generally coded directly into the compiler. As a result, it is tedious and time consuming to modify the code every time the compiler is targeted for a new processor.

17.4.5.2 Reservation Table Model

Traditionally, the resource usage pattern of an instruction i is represented using a reservation table. Instructions having identical resource usage patterns are said to belong to the same instruction class. The resource usage of any instruction in instruction class I has a single reservation table (RT) RT_I, which is an $m_I \times l_I$ bit matrix, where m_I is the number of resources needed by the instruction for its execution in the pipeline and l_I is the execution latency of the instruction [121]. An entry $RT_M[r, t] = 1$ indicates that the resource r is needed by this instruction t cycles after it is launched. Typically, each row of the reservation table is stored as a bit vector. The reservation tables for two instruction classes I_1 and I_2 are shown in Figure 17.13.

Apart from storing the reservation tables for each instruction class, the contention query module also maintains a GRT that is used to keep track of the machine state at every point in the schedule.

Resources	Time Steps			
	0	1	2	3
r_0	1	0	0	0
r_1	0	1	1	0
r_2	0	0	0	1

(a) Reservation Table for I_1

Resources	Time Steps			
	0	1	2	3
r_0	1	0	0	0
r_3	0	1	0	0
r_4	0	0	1	1

(b) Reservation Table for I_2

FIGURE 17.13 Reservation table (RT) for the example machine.

The GRT is an $M \times T$ bit matrix, where M is the total number of resources in the target machine and T is an upper bound on the length of the schedule.

To answer the query: "Can an instruction of class I be scheduled in the current cycle?," the scheduler performs bitwise AND operations of the nonzero bit vectors of RT_I with the corresponding bit vectors in the GRT, starting from the current cycle. If the results of the AND operations are all zeros, then the instruction can be scheduled in the current cycle; otherwise it cannot be scheduled. On scheduling the instruction, similar bitwise OR operations are performed on the GRT to reflect the resource usages of the scheduled instruction.

17.4.5.3 Reduced Reservation Table Model

The reduced reservation table (RRT) approach, due to Eichenberger and Davidson [38], for answering contention query is similar to the RT approach, except that the RRT uses a simplified reservation table. This simplified table is derived by eliminating much of the redundant information in the original reservation table. However, the scheduling constraints present in the original reservation table are preserved in the RRT. The resource usages are modeled using logical resources unlike the RT model, in which the actual resources of the target architecture are used. This offers a compact form of representing the reservation table, thus reducing the space required to store the tables and also the time spent in contention queries.

First, we define *forbidden latency*. For an ordered pair of instruction classes, A and B, a latency value f is said to be forbidden if two instructions, one belonging to instruction class A and another to B, when initiated on the respective functional unit types with a latency f between them cause a structural hazard. Such a structural hazard occurs if at least one resource r exists and a time step t such that $RT_A[r]$ and $RT_B[r, t + f]$ are both 1. The forbidden latency set $F_{A,A}$ consists of the forbidden latencies between two initiations of the same instruction class. Similarly, the forbidden latency set $F_{A,B}$ consists of the forbidden latencies between two initiations of two different instruction classes. For example, for the reservation tables shown in Figure 17.13, latency 0 is in the forbidden set F_{I_1,I_2}, as two instructions, one each from these two instruction classes, initiated with a latency 0 (i.e., initiated in the same time step t) requires the resource r_0 at time step t. Similarly latency 1 is in F_{I_1,I_1} and latency 1 is in F_{I_2,I_2}, as resource r_1 for instruction class I_1 and r_4 for instruction class I_2 are required for two consecutive time steps.

Construction of the RRT is explained in detail in [38]. The RRT approach uses a set of logical resources to model all the forbidden latencies of the original resource usage. We shall illustrate the RRT approach using the example machine considered in Section 17.4.5.2 (refer to Figure 17.13). The RRTs for this machine are shown in Figure 17.14. Logical resources r'_0, r'_1 and r'_2 are used to model the resource usages. Note that the resource r'_0 models the forbidden latencies $0 \in F_{I_1,I_2}$ and $0 \in F_{I_2,I_1}$. Further, the resource r'_1 models the forbidden latency $1 \in F_{I_1,I_1}$ and the resource r'_2 models the forbidden latency $1 \in F_{I_2,I_2}$. Finally, forbidden latency $0 \in F_{I_1,I_1}$ and $0 \in F_{I_2,I_2}$ are modeled by every resource. It can be verified that these RRTs model all and only those forbidden latencies that are present in the original RT. Compared with the reservation table in Figure 17.13, the RRT in Figure 17.14 is compact. The advantage of the RRT model is that it is likely to have fewer logical resources than physical resources. For the example machine, the number of logical resources are three whereas the number of physical resources in the RT model are five. As a consequence, both the space requirements of the resource model and the contention check computation become efficient.

The contention query module of the instruction scheduler uses the RRT in the same manner as in the case of the original RT. The differences, however, are in the size of the GRT and the individual reservation tables. The GRT in this case consists only of M' rows, where M' is the total number of logical resources in the machine. For the considered example, M' is three, which is significantly less compared with the number of physical resources. As a consequence, the size of the GRT also reduces significantly.

Logical Resources	Time Steps 0	1
r_0'	1	0
r_0'	1	1

(a) RRT for I_1

Logical Resources	Time Steps 0	1
r_0'	1	0
r_2'	1	1

(b) RRT for I_2

FIGURE 17.14 Reduced reservation table (RRT) for the example machine.

$$CM_{I_1} = \begin{array}{c} I_1 \\ I_2 \end{array} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}; \quad CM_{I_2} = \begin{array}{c} I_1 \\ I_2 \end{array} \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$$

FIGURE 17.15 Collision matrices for instruction classes I_1 and I_2.

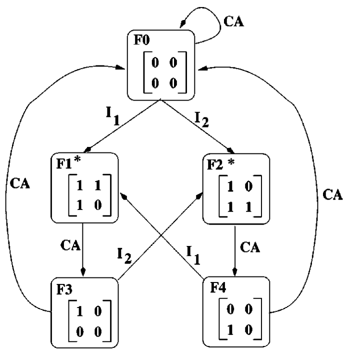
17.4.5.4 Automaton-Based Approaches

The automaton approach models the resource usage using a finite state automaton. This approach processes the RTs off-line to generate all possible legal initiation sequences in the architecture. The states of the automaton correspond to machine states at different points in the scheduling process. The automaton is constructed just once for the target architecture and thereafter the compiler uses this during the instruction scheduling phase.

The Muller method constructs the automaton directly using the RTs [102]. Each state in this automaton is essentially a snapshot of the GRT (refer to Section 17.4.5.2) in the partial schedule. Proebsting and Fraser [119] improved on the Muller technique by using collision matrices (to be defined later) for constructing the automaton. Bala and Rubin [11] extended the Proebsting technique to complex machines and introduced the notion of factoring. Although the Proebsting method directly produces the minimal automaton, it is still large for complex machines. Instead of building one large automaton, Bala and Rubin [11] used a factoring scheme to create multiple smaller automatons, the sum total of whose states is less than the number of states in the original automaton. The factoring scheme is based on the observation that modern-day processors typically divide the instruction set into different classes and each class is executed by a different functional unit. For instance, the integer and floating point units have different pipelines and use separate resources. As such, these resources can be divided into separate factors and the automaton can be constructed separately for each of the factors.

Before we proceed with the construction of the automaton, we define collision matrix [81]. A *collision matrix* CM_I for the instruction class I is a bit matrix of size $n \times l'$, where n is the number of instruction classes and l' is the longest repeat latency of an instruction class. The repeat latency of an instruction class is the minimum value such that any latency of greater than or equal to the repeat latency is permissible for the instruction class [81]. The collision matrix CM_I specifies whether a resource conflict can occur in initiating instructions of various classes, including itself, at different time steps. The rows of the collision matrix represent various instruction classes and the columns represent different time steps. More specifically, the entry $CM_I[J, t] = 1$, if t is a forbidden latency between the instructions classes I and J (i.e., $t \in F_{I,J}$). The collision matrices for the instruction classes considered in Section 17.4.5.2 are shown in Figure 17.15.

The construction of the finite state automaton proceeds as follows: each state F in the automaton is associated with a state matrix SM_F, which is an $n \times l'$ bit matrix. Given a state F and an instruction of class I, it is legal to issue I in the current cycle from state F, if $SM_F[I, 0]$ is 0. A legal issue causes a state transition $F \xrightarrow{I} F'$. The state matrix $SM_{F'}$ is computed by ORing the respective rows of SM_F with the collision matrix CM_I. The automaton for the motivating example is shown in Figure 17.16(a). When the automaton reaches a state where all the entries in the first column are



(a) Automaton

Current State	State Transition Upon		
	I_1	I_2	CA
F0	F1	F2	F0
F1*	–	–	F3
F2*	–	–	F4
F3	–	F2	F0
F4	F1	–	F0

(* indicates Cycle Advancing States)

(b) Transition Table

FIGURE 17.16 Automaton for the example machine.

1, it means that no more instructions of any instruction class can be issued from the current cycle. State F_2 is an example of such a state. Hence, the state is cycle advanced, or the automaton moves to the next time step, which results in shifting the state matrix left by one column [11]. This pseudo instruction class is marked as CA in Figure 17.16(a). When the automaton-based resource model is used in conjunction with a simple list scheduler (such as the one discussed in Section 17.4.2), which schedules instructions on a cycle by cycle basis, it suffices to examine only transition latency zero, provided cycle-advancing transitions are considered.

The automaton is represented in the form of a transition table that is used by the scheduler. The transition table is a two-dimensional matrix of size $N \times r$, where N is the number of states in the automaton and r is the number of instruction classes including the pseudo instruction class CA. The entries in the transition table are either state numbers or null (denoting illegal transitions). The transition table corresponding to the automaton for our motivating example is shown in Figure 17.16(b). Thus, in the automaton-based approach, answering a contention query corresponds to a transition table lookup; updating the machine state on scheduling an instruction is changing to a new state. Both of these are constant time operations.

Two major concerns in using the automaton-based approach are the construction time of the automaton and the space requirements of the transition table. The construction of the automaton, though a one time cost incurred at the time of compiler construction for this target architecture, could be significant because the number of states in the automaton can grow very large, with an increase in either the number of instruction classes or the latencies of instructions. For example, for the DEC Alpha architecture, Bala and Rubin report 13,254 states when the automaton is constructed for integer and floating point instruction classes together. When separate automatons are constructed, the number of states decrease to 237 and 232. For the Cydra 5 architecture [126], the number of states in the factored automaton is 1127. Thus, the number of states is still large, which directly contributes to the increase in space requirements. This is because the transition table is an $N \times n$ matrix, where N is the number of states in the automaton and n is the number of instruction classes.

In [73, 74], the automaton model is further extended to a group automaton model that reduces the space requirements by observing and eliminating certain symmetry in the constructed automaton. It identifies instruction groups, a set of instruction classes, which exhibit this symmetrical behavior. The work also proposes a resource model based on collision matrices, referred to as the *dynamic collision matrix* model. This resource model combines collision matrices with an RT-based approach to strike a good balance between space and time requirements of the resource usage model. Finally, a classification of resource usage models is also presented in [74].

An automaton-based resource model for software pipelining has been proposed independently by Govindarajan, Altman and Gao for instruction classes that do not share any resource [55]. It is subsequently extended to instruction classes that share resources in [56, 158]. The state diagram of the automaton, referred to as *modulo-scheduled (MS) state diagram*, though similar to the Bala and Rubin automaton, has two important differences. First, the MS-state diagram is specialized for software pipelining, or modulo scheduling (MS), which takes into account the repetitive scheduling of different instances of the same instruction, corresponding to different iterations. It is argued in [55, 56] that the Bala and Rubin automaton could not be directly used in a software pipelining method. Second, in the MS-state diagram, for each state, a state transition corresponds to each permissible latency, including latency of 0; whereas in the Bala and Rubin automaton, the state transitions in a state correspond only to latency value of 0.

17.4.5.5 AND/OR-Tree Model

Gyllenhaal, Hwu and Rau have proposed a two-tier model for resource usage representation [60]. It allows the user to specify the resource usage or machine description in a high-level language. The high-level language is designed to specify the machine description in an easy to understand, maintainable and retargetable manner. The high-level description is translated into a low-level representation for efficient use by the instruction scheduler. The two-tier model helps to easily retarget the scheduler for different architectures.

They have also proposed a new representation for machine description based on the AND/OR tree concept used in search algorithms, which is especially useful when a single instruction class has multiple options or resource usage patterns to choose from. This happens, for example, when there are two decode units, and an instruction can use either one of them in the decode stage. The new representation is an AND tree of OR trees: whereas the OR trees encode the multiple options available within a stage, the AND tree represents the usage of different stages. The AND/OR tree model uses the short-circuit property of the AND/OR trees to detect the resource conflicts quickly. This reduces the space complexity of the RT-based approaches and also the computation time required to answer contention queries.

17.4.6 Case Studies

In this section we review the instruction scheduling methods used in (1) the compiler for Cydra 5 VLIW architecture [13, 34], (2) the GNU C compiler [141] and (3) the IBM XL compiler family [151] as case studies.

17.4.6.1 Instruction Scheduling in Cydra 5 Compiler

The scheduler implemented in the Cydra 5 compiler is a backward pass list scheduler. It works bottom up, scheduling from the sink node of the DAG. For each instruction `i` from the priority list, the scheduler attempts to schedule the instruction starting from the largest possible start time, based on its (already) scheduled successors. The priority algorithm ensures that all successor instructions are placed in the list ahead of an instruction. In addition, the priority algorithm can give greater priority either to instructions with the least slack or to instructions that reduce register lifetimes. The former heuristic results in schedules with low execution time, but may increase the register pressure. This may cause register spills. The second heuristic is used to counter this effect. In the Cydra 5 compiler instruction scheduling is performed prior to register allocation.

17.4.6.2 Instruction Scheduling in GNU C Compiler

The instruction scheduling method used in the GNU C compiler (version 2) also follows backward pass list scheduling [141]. The method orders instructions based on a priority algorithm that gives relatively higher priority to all successor instructions compared with the parent instruction. By placing instructions with higher priority later in the schedule than ones with lower priority, the scheduler preserves dependence constraints. Further, instructions with larger execution time are also given higher priority, exposing instructions on the critical path.

The algorithm then starts scheduling by issuing the instruction with the highest priority, scheduling from the last instruction in the basic block to the first. Each time an instruction is scheduled, a check is performed on each predecessor instruction `p` to see if it has no more unscheduled successors. Such instructions are marked "ready," and added to the ready list in priority order. When all instructions are scheduled, the algorithm terminates. The preceding scheduling method works well to produce good schedules, but generally increases the register pressure, and results in poor performance when the number of available registers is less than the required number. For this purpose, the list scheduling method also gives higher priority to instructions that end live ranges of variables.

17.4.6.3 Instruction Scheduling in the IBM XL Compiler Family

The XL family compiler of IBM uses a forward pass list scheduling method [151]. The primary priority heuristic used is based on the maximum distance to the sink node. In addition, it also uses a combination of smallest Estart value, minimum liveness and greatest uncovering — corresponds to how many (ready) instructions can be added to the ReadyList — heuristics. The list scheduling algorithm starts scheduling from the source node, attempting to schedule instructions at time steps closer to its Estart time. The scheduling method is applied both as a Prepass method (before register allocation) and as a Postpass method (after register allocation). It takes care of a number of types of delay stalls, and schedules fixed and floating point instructions in an alternating sequence for the RS/6000 processor.

17.5 Global Scheduling

Instruction scheduling within a basic block has limited scope because the average size of a basic block is quite small, typically in the range of 5 to 20 instructions. Thus, even if the basic block scheduling method produces optimal schedules, the performance, in terms of the exploited ILP, is low. This is especially a serious concern in architectures that support greater ILP (e.g., VLIW architectures with several functional units or superscalar architectures that can issue multiple instructions every cycle). The reason for the low ILP, especially near the beginning and end of basic blocks, is that basic block boundaries act like barriers, not allowing the movement of instructions past them.

Global instruction scheduling techniques, in contrast to local scheduling, schedule instructions beyond basic blocks (i.e, overlapping the execution of instructions from successive basic blocks). These global scheduling methods are either for a set of basic blocks with acyclic control flow among them [34, 45, 72, 97], or for single or multiple basic blocks of a loop [24, 30, 44, 84, 122]. The former case is referred to as *global acyclic scheduling* and the latter as *cyclic scheduling*. First, we discuss a few global acyclic scheduling methods. Section 17.5.2 deals with cyclic scheduling.

17.5.1 Global Acyclic Scheduling

Early global scheduling methods performed local scheduling within each basic block and then tried to move instructions from one block to an empty slot in a neighboring block [24, 145]. However, these methods followed an ad hoc approach in moving instructions. Further, the local compaction or scheduling that took place in each of the blocks resulted in several instruction movements (reorderings) that were done without understanding the opportunities available in neighboring blocks. Hence, some of these reorderings may have to be undone to get better performance. In contrast, global acyclic scheduling methods, such as trace scheduling [45], percolation scheduling [105], superblock scheduling [72], hyperblock scheduling [97], and region scheduling [61], take a global view in scheduling instructions from different basic blocks. In the following subsections we describe these approaches.

17.5.1.1 Trace Scheduling

Trace scheduling attempts to minimize the overall execution time of a program by identifying frequently executed traces — acyclic sequences of basic blocks in the control flow graph — and scheduling the instructions in each trace as if they were in a single basic block. The trace scheduling method first identifies the most frequently executed trace, a single path in the control flow graph, by first identifying the unscheduled basic block that has the highest execution frequency; the trace is then extended forward and backward along the most frequent edges. The frequency of edges and basic blocks are obtained by a linear combination of branch probabilities and loop trip counts obtained either through heuristics or through profiling [12]. Various profiling methods are discussed in greater detail in [58].

The instructions in the selected trace (including branch instructions) are then scheduled using a list scheduling method. The objective of the scheduling is to reduce the schedule length, and hence the execution time of the instructions in the trace. During the scheduling, instructions could move above or below a branch instruction. Such movement of instructions may warrant compensation code to be inserted at the beginning or end of the trace. After scheduling the most frequently executed trace, the next trace (involving unscheduled basic blocks) is selected and scheduled. This process continues until all the basic blocks are considered.

Let us illustrate the trace scheduling method with the help of the example code shown in Figure 17.17(a), adapted from [71]. The instruction sequence and the control flow graph for the code are shown in Figure 17.17(b) and 17.17(c). Consider a simple two-way issue architecture with two integer units. Let us assume that the latency of an integer instruction, such as `add`, `sub` or `mov` instruction, is one cycle, and that of a `load` instruction is two cycles. Thus, a stall of one cycle is required between a `load` and a dependent instruction. For simplicity, we assume here that branch instructions do not require any stall cycle.

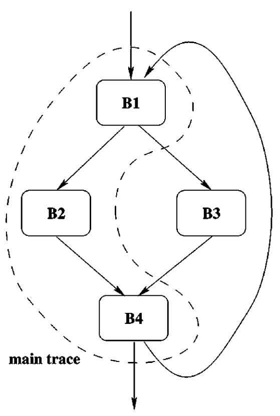
A basic block scheduling method achieves the schedule shown in Figure 17.18. Because instructions cannot be moved beyond basic block boundaries, this is the best schedule that can be achieved for the given machine. The first column in Figure 17.18 represents the time steps at which the instructions can be issued. The time steps shown in parentheses for instructions `i9` to `i11`

```
for (i=0; i < 100; i++)
{
   if (A[i] == 0)
     B[i] = B[i] + s;
   else
     B[i] = A[i];
   sum = sum + B[i];
}
```

(a) High-Level Code

		%% r1 ← 0 %% r5 ← 0 %% r6 ← 400 %% r7 ← s
B1:	i1:	r2 ← load a(r1)
	i2:	if (r2 != 0) goto i7
B2:	i3:	r3 ← load b(r1)
	i4:	r4 ← r3 + r7
	i5:	b(r1) ← r4
	i6:	goto i9
B3:	i7:	r4 ← r2
	i8:	b(r1) ← r2
B4:	i9:	r5 ← r5 + r4
	i10:	r1 ← r1 + 4
	i11:	if (r1 < r6) goto i1

(b) Assembly Code



(c) Control Flow Graph

FIGURE 17.17 Multiple basic blocks example.

Time	Int. Unit 1		Int. Unit 2	
0	`i1:`	`r2 ← load a(r1)`		
1				
2	`i2:`	`if (r2 != 0) goto i7`		
3	`i3:`	`r3 ← load b(r1)`		
4				
5	`i4:`	`r4 ← r3 + r7`		
6	`i5:`	`b(r1) ← r4`	`i6:`	`goto i9`
3	`i7:`	`r4 ← r2`	`i8:`	`b(r1) ← r2`
4				
7 (5)	`i9:`	`r5 ← r5 + r4`	`i10:`	`r1 ← r1 + 4`
8 (6)	`i11:`	`if (r1 < r6) goto i1`		

FIGURE 17.18 Basic block schedule for the instruction sequence of Figure 17.17.

correspond to the schedule time when the control flow is B1 → B3 → B4. Note that the extra cycle (time step 4) in the schedule (after instruction i7 and i8) in basic block B3 is due to the stall for load instruction (i8) at the basic block boundary. It takes 9 cycles for the path B1 → B2 → B4 and 7 cycles for B1 → B3 → B4.

Assume that basic blocks $B1$, $B2$ and $B4$ are frequently executed, and they form the main trace. By allowing the instructions in basic block $B2$ to be moved above the split point in the control flow graph, a compact schedule for the most frequently executed trace can be obtained. For example, instruction i3 can be moved to block $B1$. Such movement of instructions above a conditional branch instruction is referred to as *speculative code motion*. Moving an instruction that could potentially raise an exception, such as a memory load or a divide instruction, speculatively above a control split point, requires additional hardware support as discussed in [22]. This is to avoid raising unwanted exceptions.

The original program semantics must be ensured under speculative code motion. For this, the destination register of an instruction i should not be live on entry on alternative paths on which i is control dependent. The reason is, when execution proceeds on an alternative path, instruction i that was speculatively executed would have modified the destination register that is live on entry in this path. Suppose register r3 is live on entry for basic block B3 in our example. (That is, some instruction j is in B3 for which r3 is a source operand and no instructions are in B3, before j, that defines r3). Then speculative motion of i3 from basic block B2 to B1 destroys the live-in value of r3 for instruction j. Thus, to perform speculative code motion of an instruction whose destination register is live-in on an alternative path, the destination register must be renamed appropriately at compile time.

A schedule for the main trace is shown in Figure 17.19. In this schedule, the main trace consisting of basic blocks B1, B2 and B4 can be executed in 6 cycles whereas the off-trace path involving B1, B3 and B4 can be executed in 7 cycles. By scheduling instructions across basic blocks, the execution time of the main trace is reduced from nine to six cycles.

When execution goes through the less frequently executed path, off-trace path, to preserve correct program execution, some of the instructions may be duplicated. The code inserted to ensure correct program behavior and thus compensate for the code movement is known as *compensation code*. For example, if an instruction from basic block B1 is moved down below the control split point to B2, then a compensation code has to be inserted in B3. Several other examples of code movement and the required compensation code are illustrated in [95].

The trace-scheduling algorithm should maintain the need for introducing such compensation code at various program points. This is known as *bookkeeping*. The compensation code may increase the

Time	Int. Unit 1		Int. Unit 2	
0	i1:	r2 ← load a(r1)	i3:	r3 ← load b(r1)
1				
2	i2:	if (r2 != 0) goto i7	i4:	r4 ← r3 + r7
3	i5:	b(r1) ← r4		
4 (5)	i9:	r5 ← r5 + r4	i10:	r1 ← r1 + 4
5 (6)	i11:	if (r1 < r6) goto i1		

3	i7:	r4 ← r2	i8:	b(r1) ← r2
4	i12:	goto i9		

FIGURE 17.19 Trace schedule for the instruction sequence of Figure 17.17.

schedule length of other traces. Because the objective is to reduce the overall execution time and the trace that is scheduled first is the most frequently executed one, compacting the schedule of the instructions in this trace is desirable, even if this increases the schedule length of other traces. A key property of trace scheduling, as pointed out in [121], is that the decisions as to whether to move an instruction from one basic block to another, where to schedule it, etc. are all made jointly in the same compiler phase. The trace-scheduling method was implemented in the Bulldog compiler [41]. The work was later enhanced into a production quality Multiflow compiler [95].

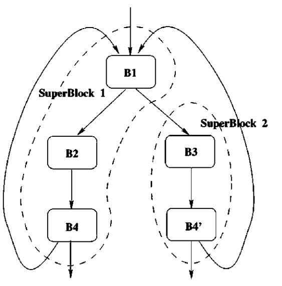
17.5.1.2 Superblock Scheduling

Hwu et al. propose a variant of trace scheduling called *superblock scheduling* [72] in the IMPACT project [22]. The motivation and the basic idea behind superblocks comes from the observation that the complexities involved in maintaining bookkeeping information in trace scheduling arise due to several incoming control flow edges at various points in a trace. The bookkeeping associated with these entrances, known as side entrances, can be avoided if the side entrances themselves are eliminated in the trace. For example, in the trace shown in Figure 17.17(c), there is a side entrance into basic block B4. Thus, by eliminating the side entrances, the (control flow) join points, as well as the associated bookkeeping, can be eliminated in a superblock. To summarize, a superblock trace consists of a sequence of basic blocks with a single entry (at the beginning of the superblock) and multiple exits. Superblocks are formed in two steps: in the first step traces are identified using profile information; the second step performs tail duplication to eliminate side entrances.

We explain the construction of superblocks with the help of the example discussed in Section 17.5.1.1. Once again, let us assume that basic blocks B1, B2 and B4 constitute the main trace as shown in Figure 17.17(c). As explained earlier, the second trace in the control flow graph makes a control flow entry to $B4$, and hence a side entrance to the main trace. To form superblocks for this trace, the tail block $B4$ is replicated to eliminate the side entrance. The superblocks for the control flow graph are shown in Figure 17.20(a).

Three optimizations to enlarge the size of a superblock have been proposed in [72], which, in turn, enhance the scope for exploiting higher ILP. Additional dependence-removing optimizations are subsequently performed to expose greater ILP. After these optimizations, the instructions in enlarged superblocks are scheduled using a list scheduling method. A schedule for the superblock is shown in Figure 17.20(b). Although no bookkeeping code is needed in this example, avoiding the side entrance enables further compaction of the schedule for the main trace or superblock 1. It can be seen that superblock 1 can be executed in 5 cycles. When the control flows to superblock 2, execution takes 6 cycles.

Both trace and superblock scheduling consider a linear sequence of basic blocks from a single control flow path. Both methods can move instructions from one basic block to another. However,



(a) Control Flow Graph

Time	Int. Unit 1		Int. Unit 2	
0	i1:	r2 ← load a(r1)	i3:	r3 ← load b(r1)
1				
2	i2:	if (r2!=0) goto i7	i4:	r4 ← r3 + r7
3	i5:	b(r1) ← r4	i10:	r1 ← r1 + 4
4	i9:	r5 ← r5 + r4	i11:	if (r1<r6) goto i1

3	i7:	r4 ← r2	i8:	b(r1) ← r2
4	i9':	r5 ← r5 + r4	i10':	r1 ← r1 + 4
5	i11':	if (r1<r6) goto i1		

(b) Superblock Schedule

FIGURE 17.20 Superblock formation and scheduling.

store instructions that write into memory locations are not speculatively moved above branches, because this would modify a memory location whose old contents may be needed in an alternative path when execution proceeds on an off-trace path. Likewise, instructions that could potentially cause an exception, such as load, store, integer divide and floating point instructions, are typically not speculatively moved; otherwise, additional hardware support, in the form of nontrapping instructions, would be required [22].

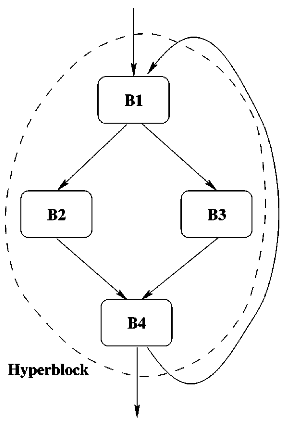
17.5.1.3 Hyperblock Scheduling

Trace scheduling and superblock scheduling rely on the existence of a main trace, the most frequently executed path in the control flow graph. Although this is likely in scientific computations, it may not be the case in control-intensive symbolic computing that dominates integer benchmark programs. To handle multiple control flow paths simultaneously, Mahlke et al. propose hyperblock scheduling [97]. In this approach, the control flow graph is IF converted [6] so as to eliminate conditional branches. IF conversion replaces conditional branches with corresponding comparison instructions each of which sets a predicate. Instructions that are control dependent on a branch are replaced by predicated instructions that are dependent on the corresponding predicate. For example, an instruction `t1` ← `t2 + t3` that is control dependent on the condition `(t4 = 0)` is converted to:

```
i :  p1 ← (t4 == 0)
i':  t1 ← t2 + t3, if p1
```

Instruction `i'` is predicated on `p1` and `t1` ← `t2 + t3` is performed only if `p1` is true. Thus by using IF conversion, a control dependence can be changed to a data dependence. In architectures supporting predicated execution [22, 77, 126], a predicated instruction is executed as a normal instruction if the predicate is true; it is treated as a NO-OP otherwise.

A hyperblock is a set of predicated basic blocks, and as with superblocks, has a single entry and multiple exits. However, unlike a superblock that consists of instructions from only one path of control, a hyperblock may consist of instructions from multiple paths of control. The presence of multiple control flow paths in a hyperblock enables better scheduling for programs with heavily biased branches. The region of blocks chosen to form a hyperblock is from an innermost loop, although a hyperblock is not necessarily restricted only to loops. Whereas conventional IF conversion can predicate all basic blocks in an innermost loop, hyperblocks selectively predicate only those that would improve program performance. A heuristic based on the frequency of execution, size and characteristics (such as whether they contain function calls) of basic blocks is used in selecting the blocks for predication. The reason for selectivity in predication is that combining unnecessary basic blocks (from different control flow paths) results in wasting the available resources, leading to poor performance.



(a) Control Flow Graph

Time	Int. Unit 1		Int. Unit 2	
0	`i1:`	`r2 ← load a(r1)`	`i3:`	`r3 ← load b(r1)`
1				
2	`i2':`	`p1 ← (r2 == 0)`	`i4:`	`r4 ← r3 + r7`
3	`i5:`	`b(r1) ← r4, if p1`	`i8:`	`b(r1) ← r2, if !p1`
4	`i10:`	`r1 ← r1 + 4`	`i7:`	`r4 ← r2, if !p1`
5	`i9:`	`r5 ← r5 + r4`	`i11:`	`if (r1<r6) goto i1`

(b) Hyperblock Schedule

FIGURE 17.21 Hyperblock formation and scheduling.

The selected set of basic blocks should (1) not have a side entrance and (2) not contain an inner loop. Tail duplication is done to eliminate side entrances in a hyperblock. Loop peeling is performed on a nested inner loop that iterates only a few times to enable including both inner and outer loops in the hyperblock. Finally, when basic blocks from different control paths are included in a hyperblock, and when the execution times of the control paths are vastly different, node splitting is performed on nodes subsequent to the merge point (corresponding to the multiple control path). Node splitting duplicates the merge and its successor nodes.

Once the blocks are selected for a hyperblock, they are IF converted. Then certain hyperblock specific optimizations are performed [97]. Finally the instructions in a hyperblock are scheduled using a list scheduling method. In hyperblock scheduling, two instructions that are in mutually exclusive control flow paths may be scheduled on the same resource. If the architecture does not support predicated execution, reverse IF conversion [153] is performed to regenerate the control flow paths.

Let us once again consider the control flow graph shown in Figure 17.17(c). If basic blocks `B2` and `B3` are both equally likely to be executed, then the superblock scheduling method can choose only one of the two basic blocks, whereas both can appear in a hyperblock as shown in Figure 17.21(a). A new instruction `i2'` that sets a predicate is introduced in the code. Instructions `i3`, `i4` and `i5` are predicated on `p1` whereas `i7` and `i8` are predicated on the complement of `p1` (i.e., `!p1`). Because instructions `i3` and `i4` are now data dependent on `i2'`, they can be scheduled only after time step 2. This results in a lengthier schedule. However, by identifying that these two instructions can be speculatively executed, predicate promotion can be performed on these instructions [97], and they can be scheduled earlier. The resulting schedule is shown in Figure 17.21(b). Note, however, that the resulting hyperblock schedule takes 6 cycles, and hence actually results in a performance degradation if the control flow path `B1` $\rightarrow$ `B2` $\rightarrow$ `B4` is taken.

Tail duplication and node splitting performed to form hyperblocks result in duplication of code. This may increase the code size significantly. Another concern in hyperblock scheduling is that an aggressive selection for alternate control flow paths may unnecessarily increase the resource usage and hence may result in degenerated schedules. For hyperblock scheduling to be effective, both code duplication and inclusion of alternate control flow paths must be kept under check.

17.5.1.4 Other Global Acyclic Scheduling Methods

In [98], global acyclic scheduling methods have been classified as either a profile-driven or a structure-driven approach. The global acyclic methods discussed in the earlier subsections fall under the profile-driven approach. They identify the most frequently executed paths using profile information and coalesce them into an extended basic blocks. In contrast, the structure-driven approach identifies and attempts to increase parallelism along all execution paths by moving operations between basic blocks, considering program structure and without using profile information. Examples of structure-driven approaches are region scheduling [59], percolation scheduling [105] and global scheduling [14]. In this section we briefly review some of the structure-driven global scheduling methods, as well as a few other profile-driven scheduling methods.

Trace scheduling is generalized to deal with general control flow in percolation scheduling [105]. Percolation scheduling uses four transformations, namely, delete, move, move conditional and unify. For each node in the control flow graph, it tries to apply each of the four transformations and repeats the transformation until none can be applied. Three out of four of these transformations move operations upward in the control flow graph. Delete removes a node when it is empty. Percolation scheduling originally assumed unbounded resources. It is then extended to nonunit execution latencies, but still with unbounded resources, in [104]. Trailblazing [103] is another extension to percolation scheduling that exploits the structure of the hierarchical task graph representation to move instructions across large blocks of code in a nonincremental way (i.e., without having to move instructions in a step-by-step manner through every block in the control flow path). This facilitates both efficient code motion and elimination of code explosion in certain cases.

Meld scheduling is a simple but effective instruction-scheduling method across basic blocks that was used in the Cydra 5 compiler [34]. It follows a simple basic block scheduling approach, except that during the scheduling of a basic block B, if either the predecessor (or the successor block), say B', has already been scheduled, then the resource usage information at the end (or, respectively, the beginning) of schedule for B' is used as the resource usage at the start (or end) of the the basic block B. The resource usage entering into the basic block should take into account the multiple basic blocks from where (to which) control flow could enter (leave) block B. Taking into account the resource usage at the boundary of neighboring basic blocks and scheduling instructions from the current basic block allows the overlap of instructions across basic blocks. The work of Abraham, Kathail and Deitrich generalizes this idea and quantitatively evaluates the benefits of meld scheduling [1].

Another global code scheduling, called *region scheduling*, has been discussed in [59]. This approach is based on extended program dependence graph representation allowing code motion between regions consisting of control equivalent statements [43]. Regions are classified according to their parallelism content that is used to drive a set of powerful code transformations. Golumbic and Rainish propose several simple schemes for scheduling instructions beyond basic blocks [51].

A global instruction scheduling method, also based on program dependence graph, has been implemented in the IBM XL family of compilers for the IBM RS/6000 systems [14]. The scheduling method proceeds by processing one basic block at a time. However, when scheduling instructions in a basic block B, consideration is given to instructions from control equivalent blocks, instructions from successors of B and successors of control equivalent blocks. Whereas the latter two categories of instructions (from successor blocks) are considered speculative instructions, instructions from control equivalent blocks are considered as useful instructions. During a scheduling step, speculative instructions can be scheduled, provided they are data ready, resources for them are available and they are schedulable across basic blocks. However, preference is given to useful instructions as opposed to speculative ones. This is especially important in machines having a small number of functional units, such as the RS/6000 system.

Next we turn our attention to a few other profile-driven scheduling methods. Hank, Hwu and Rau have proposed region-based compilation, an approach that allows an arbitrary collection of basic blocks, possibly extending over multiple function bodies, to be considered as a compilation unit [61]. The region formation approach is a generalization of profile-based trace selection. A region can expand across more than one control path. Region formation considers aggressive function inlining to extend regions across function bodies. The region formation approach is proposed as a generalized technique that is applicable to the entire compilation process, including ILP optimizations, instruction scheduling and register allocation. A global scheduling technique that operates over a restricted region, a single entry subgraph, has been proposed in [96]. A region-based register allocation approach has been discussed in [80].

As mentioned earlier, trace scheduling and superblock scheduling operate on linear sequences of basic blocks from a single control flow path, and favor the current trace path at the expense of instructions in the off-trace trace. Hsu and Davidson [68] and, more recently, Havanaki, Banerjia and Conte [62] have proposed global scheduling methods that operate on a tree of basic blocks, possibly involving multiple control flow paths. A treeregion, as the name suggests, is a tree in the control flow graph, where, except for the root node, all other nodes (basic blocks) have a single incoming edge. Scheduling of instructions in the tree of basic blocks can benefit from profile information. Further, a tree of regions, referred to as *treegion* [62], can be enlarged by performing tail duplication of merge nodes (and its successors). Compile-time register renaming is used to allow speculative code motion of instructions above their control-dependent branches.

The integrated global local scheduling (IGLS) approach [98] is a hybrid of profile-driven and structure-driven scheduling approaches. This method avoids the tail duplication and bookkeeping overheads of profile-driven approaches. IGLS orders the selection of blocks using profile information. However, in applying the code-reordering transformation, it follows a structure-driven approach and does not necessarily restrict code reordering to any trace or extended block. Also, the selection of the appropriate code motion and the target block selection are made flexible and depends on the current properties of the block such as the available parallelism within the block. The method has been implemented in the SGI MIPSpro compiler.

An important consequence of aggressive speculative scheduling of instructions in a global instruction scheduling method is that it may unduly delay some of the paths in the global region (such as a superblock) considered for schedule. This happens especially when the resources (functional units) are limited. The reason for this is that the profile information is used only during the formation of the global region and not while scheduling the instructions. Fisher proposed the use of speculative yield — probability that a speculatively scheduled instruction produces useful work — along with dependence height (similar to MaxDistance defined in Section 17.4.2.1) in scheduling instructions in the global region [47]. Successive retirement is another profile-independent scheduling heuristic that attempts to retire each path (or exit) in order, as early as possible [26]. This heuristic, applied to superblock regions, minimizes speculation, so that it only speculates when no nonspeculative instructions are available. The speculative hedge heuristic attempts to ensure that no path gets delayed unnecessarily by accounting for different processor resources while scheduling, and not just using a common scheduling priority function based on dependence height [35]. Finally, the treegion-scheduling method [62], by virtue of scheduling multiple paths in parallel, also avoids unduly penalizing the off-trace paths.

17.5.2 Cyclic Scheduling

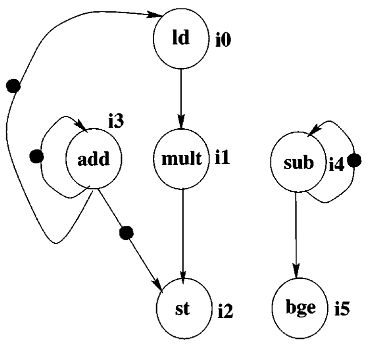
To exploit higher ILP in loops, several cyclic-scheduling methods have been proposed to overlap the execution of instructions from multiple basic blocks, where the multiple blocks could be multiple instances of the same static basic block corresponding to different iterations. Early cyclic-scheduling methods unrolled the loop several times and performed some form of global scheduling on the

```
for (i = 0; i < n; i++) {
    a[i] = s * a[i];
}
```

(a) High-Level Code

	`% t0 ← 0 %`
	`% t1 ← (n-1) %`
	`% t2 ← s %`
i0:	`t3 ← load a(t0)`
i1:	`t4 ← t2 * t3`
i2:	`a(t0) ← t4`
i3:	`t0 ← t0 + 4`
i4:	`t1 ← t1 - 1`
i5:	`if (t1 ≥ 0) goto i0`

(b) Instruction Sequence



(c) Dependence graph

FIGURE 17.22 Software pipelining example.

unrolled loop [44]. Although this approach exploits greater ILP within the unrolled iteration, very little or no overlap occurs across the iterations of the unrolled loop.

Software pipelining [5, 84, 121, 122] overlaps the execution of instructions from multiple iterations of a loop. The objective here is to sustain a high initiation rate, where the initiation of a subsequent iteration may start even before the previous iteration is complete. We discuss software pipelining only briefly here, because this is the topic of discussion of Chapter 18 [4] in this book.

Let us first explain software pipelining with the help of the example (refer to Figure 17.22) adapted from [54, 124]. The dependences among the instructions in the loop are represented by means of a data dependence graph (DDG). The DDG, unlike the DAG used thus far for acyclic scheduling, may be cyclic. In particular, a dependence from an instruction `i` to `i'` could be across iterations. That is, the value produced by `i` in the jth iteration could be used by `i'` in iteration $(j + d)$. Such a dependence is known as a *loop-carried dependence* with a *dependence distance d*. A loop-carried dependence is marked in the DDG by means of tokens on the dependence arc. The number of tokens present in an arc indicates the dependence distance.

For the instruction sequence shown in Figure 17.22(b), the dependence graph is depicted in Figure 17.22(c). In this graph, we assume that the possible dependence from `store` to `load` can be disambiguated, and hence omitted. We shall assume an architecture with two integer functional units and two floating point units. Let the latencies of instructions in these functional units be one and two cycles, respectively. Further, we assume that load and store instructions are executed by a load and store unit with execution times of 2 and 1 time units, respectively. An acyclic-scheduling approach is able to achieve a schedule in which the execution time of each iteration is five cycles. This corresponds to an initiation rate of $\frac{1}{5}$ iterations per cycle.

Software pipelining overlaps successive iterations of the loop and hence can exploit higher ILP. Successive iterations of a loop are initiated with an initiation interval (II) and initiation rate (1/II). The minimum initiation interval (MII) achievable for a given loop is governed by resource constraints and recurrences or cyclic data dependences. The MII of a loop is the maximum of Resource MII (ResMII) and Recurrence MII (RecMII).

RecMII is determined from the dependence cycles in the DDG [127]. Specifically:

$$\text{RecMII} = \max_{\forall\ cycles\ C} \left\lceil \frac{\text{sum of execution latencies of instructions in } C}{\text{sum of dependence distances in } C} \right\rceil$$

ResMII, for simple resource usage patterns (fully pipelined functional units), is given by:

$$\text{ResMII } = \max_r \left\lceil \frac{N_r}{F_r} \right\rceil$$

where N_r is the number of instructions that can be executed in functional unit of type r and F_r is the number of instances of type r functional unit. For our target architecture:

$$\text{ResMII } = \max(\text{ResMII}_{Int}, \text{ResMII}_{FP}, \text{ResMII}_{Ld/St})$$

$$\text{ResMII } = \max\left(\frac{3}{2}, \frac{1}{2}, \frac{2}{1}\right) = 2$$

For the DDG in Figure 17.22(b) there are two self-cycles on instructions `i3` and `i4`. Hence, RecMII for the loop is:

$$RecMII = \max \left(\frac{1}{1}, \frac{1}{1}\right) = 1$$

Thus:

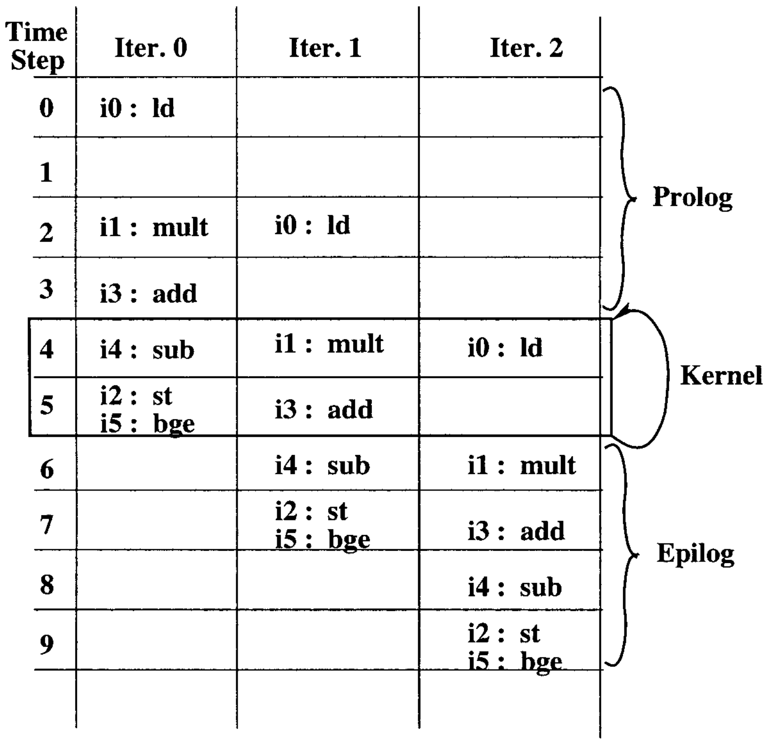
$$MII = \max(\text{RecMII}, \text{ResMII}) = \max(1, 2) = 2$$

In our discussion we consider periodic linear schedules, under which various instructions begin their execution at time steps given by a simple linear relationship. The jth iteration of an instruction `i` begins execution at time $II \cdot j + t_i$, where $t_i \geq 0$ is an integer offset and II is the initiation interval of the given schedule. It can be seen that t_i is also the schedule time of instruction `i` in iteration 0.

Figure 17.23 gives a resource constrained schedule with $II = 2$ for our example loop. This schedule is obtained with $II = 2$, $t_{\texttt{i0}} = 0$, $t_{\texttt{i1}} = 2$, $t_{\texttt{i2}} = 5$, $t_{\texttt{i3}} = 3$, $t_{\texttt{i4}} = 4$ and $t_{\texttt{i5}} = 5$. The schedule has a prologue (from time step 0 to time step 3), a repetitive pattern (at time steps 4 and 5) and an epilogue code starting from time step 6 to 9 as shown in Figure 17.23. Further, at the first time step in the repetitive pattern (time step 4), instructions `i0`, `i1` and `i4` are scheduled. Instructions `i2`, `i3` and `i5` are scheduled at the second time step (time step 5). The repetitive kernel is executed $(n-2)$ times in this case and hence `t1` must be appropriately set. It can be seen that the resource requirement in both cycles in the repetitive kernel is within the available resources. Further, the schedule shown earlier is one of those resource-constrained schedules that achieves the lowest initiation interval (MII = 2). Hence, the schedule is a rate-optimal schedule.

Obtaining a rate-optimal resource constrained software pipelined schedule is known to be NP complete [84, 121]. Hence, many of the proposed methods for software pipelining attempt to obtain a near-optimal resource constrained schedule. A number of heuristic methods for software pipelining have been proposed [34, 36, 49, 84, 99, 120, 125, 138, 150], starting with the work of Rau and Glaeser [122] and its application in the FPS compiler and Cydra 5 compiler [33, 34]. Some of these algorithms backtrack some of the scheduling decisions to obtain efficient schedules.

A resource-constrained software pipelining method using list scheduling and hierarchical reduction of cyclic components has been proposed by Lam [84]. Her approach identifies strongly connected components — the maximal connected subgraph of the underlying undirected graph, where a path exists between every pair of nodes — and schedules the instructions in them. The strongly connected component is then treated as a single pseudo operation with a complex resource usage pattern. Like this, the remaining DDG is reduced in a hierarchical way. Other heuristic-based scheduling methods have been proposed by Gasperoni and Schwiegelshohn [49], Wang and Eisenbeis [150] and Rau [125]. The problem of obtaining a rate-optimal resource constrained software pipelined schedule is formulated as an integer linear programming problem in [42, 53, 55]. Altman, Govindarajan and



Time Step	Iter. 0	Iter. 1	Iter. 2
0	i0 : ld		
1			
2	i1 : mult	i0 : ld	
3	i3 : add		
4	i4 : sub	i1 : mult	i0 : ld
5	i2 : st i5 : bge	i3 : add	
6		i4 : sub	i1 : mult
7		i2 : st i5 : bge	i3 : add
8			i4 : sub
9			i2 : st i5 : bge

Prolog
Kernel
Epilog

FIGURE 17.23 Software pipelined schedule with II = 2.

Gao have extended their integer linear program formulation to handle complex resource usage patterns by unifying the scheduling and mapping problem in a single framework [7]. Efficient integer linear program formulation has been proposed in [39] which makes use of structured 1-0 formulation [25].

In [54, 56], a novel scheduling method, called co-scheduling, has been proposed, which is a heuristic method that uses MS-state diagram, an automaton-based resource usage model. The MS-state diagram model, proposed independently, is similar to the finite state automaton approach proposed by Bala and Rubin [11]; the main difference is that the former incorporates information about the initiation interval (II).

In addition to obtaining efficient schedules, in terms of low II, many software pipelining methods also attempt to reduce the register requirements of the constructed schedule. The Huff slack scheduling method [70] is an iterative solution that gives priority to scheduling instructions with minimum slack (as defined in Section 17.4.2.1) and tries to schedule an instruction at a time step that minimizes register pressure. Stage scheduling constructs a schedule with lower register requirements from an already constructed resource constrained software pipeline schedule either using a number of heuristics [37] or solving a linear programming problem [40]. The newly constructed schedule and the original schedule have the same repetitive kernel and $t_{\mathtt{i}}$ values (the schedule time of instruction `i` in iteration 0). The hypernode reduction modulo scheduling (HRMS) method [93], register-sensitive software pipelining [32] and swing modulo scheduling [92] are some of the other software pipelining methods that reduce the register requirements of the software pipelined schedule.

Register allocation of software pipelined schedules has been studied in [124]. A number of register allocation strategies were discussed and evaluated for architectures with and without specific hardware support. Zalamea et al. have studied register spills in software pipelining [157]. An important issue in software pipelining is that of handling the live ranges of the same variable corresponding to different iterations that overlap with themselves. For example, the value produced by instruction `i1` at time step 2 in the schedule shown in Figure 17.23 is used by instruction `i2` only at time step 5. However, another instance of `i1` corresponding to the next iteration is executed (at time step 4), which could overwrite the destination register. Modulo variable expansion is a technique that unrolls the schedule a required number of times, and renames the destination register appropriately to handle multiple simultaneously live values [84, 122]. Hardware support in the form of rotating registers was proposed in Cydra 5 [126] as a solution to this problem. With rotating registers, unrolling of loop schedules as in modulo variable expansion is not necessary. A software pipelining method that is sensitive to modulo variable expansion has been proposed in [147]. This method first unrolls the loop an estimated number of times and schedules it in such a way as to avoid overlapping live ranges of the same variable.

Loops consisting of multiple basic blocks with arbitrary acyclic control flow in the loop body pose another important challenge for software pipelining. The hierarchical reduction approach that schedules strongly connected components and reduces them as a single pseudo operation can handle conditionals as well [84]. In this approach, the two branches of a conditional are first scheduled independently. The entire conditional is then represented as a single node whose resource usage at each time step is the union of resource usages of the two branches, with the length of the schedule equal to the maximum of the lengths of the branches. After the entire loop is scheduled, the explicit control structure is regenerated by inserting conditionals. Another approach to handle conditionals in a loop body is by performing IF conversion [6]. The IF-converted (or predicated) code can be scheduled [33] for architectures that support predicated execution [22, 77, 126] as if it were a single basic block. However, the resource usage for predicated code is the sum of the resource usages instead of their union.

The enhanced modulo-scheduling method [152] follows an approach similar to software pipelining predicated code. However, it regenerates the explicit control structure as in the hierarchical reduction method [84]. This not only eliminates the disadvantage on resource requirements of predicated methods, but also does not require hardware support for predicated execution. In [154], a software pipelining method that uses multiple II values has been proposed. The scheduling procedure is reminiscent of trace scheduling; the most likely trace of execution is chosen and scheduled separately with the smallest possible II. The next trace is scheduled on top of this trace, filling in holes with an II that is a multiple of the smallest II and so on.

A comprehensive survey of various software-pipelining methods can be found in [5, 121].

17.6 Scheduling and Register Allocation

In this section we discuss the interaction between instruction scheduling and register allocation, another important phase in an optimizing compiler. Register allocation determines which frequently used variables are kept in registers to reduce memory references. Instruction scheduling and register allocations phases influence each other and hence the ordering of these two phases in a compiler is an important issue.

17.6.1 Phase-Ordering Issues

In many early compilers, instruction scheduling and register allocation phases were performed separately, with each phase ignorant of the requirements of the other, leading to degradation of performance. The performance degradation can be explained as follows: in postpass scheduling where register allocation precedes instruction scheduling [50, 64], the register allocator, in an attempt to reduce the register requirements, may reuse the same register for different variables. This reuse of registers could result in anti- and output dependences, which in turn limit the reordering opportunities. On aggressive multiple instruction issue processors, especially those that are statically scheduled, the parallelism lost may far outweigh any penalties incurred due to spill code.

On the other hand, in a prepass method [10, 52, 151], instruction scheduling is performed before register allocation. This typically increases the lifetimes of registers, possibly leading to more spills and hence degrading performance. Further, any spill code generated after the register allocation pass may go unscheduled because scheduling was done before register allocation. This may even lead to illegal schedules in statically scheduled processors, if the resources required for the spill code are not available. Therefore, it is customary that prepass scheduling is followed by register allocation and postpass scheduling.

17.6.2 Integrated Methods

A number of integrated techniques have been proposed in the literature to introduce some communication between the two phases [16, 18, 52, 117]. These integrated techniques increase the ILP exposed to the processor without drastically increasing the number of spills and hence improve performance considerably. We discuss two of these integrated methods, namely, integrated prepass scheduling (IPS) [52] and parallel interference graph method [117] in detail. A number of other integrated methods have also been proposed in the literature [15, 18, 100, 107], which are reviewed briefly.

17.6.2.1 Integrated Prepass Scheduling

In IPS [52], instruction scheduling precedes register allocation; however, the scheduler is given a bound on the number of registers that guides it to increase parallelism when the register pressure is low and to limit the parallelism otherwise.

The basic idea is to keep track of the number of available registers during the scheduling phase. Each issued instruction may create a new live register and terminate the lifetime of some registers. Hence, the method keeps track of the number of available registers at each schedule step. The main algorithm switches between two schedulers. When there are enough registers, the scheduler uses code scheduler to avoid pipeline (CSP) delays, which schedules instructions to avoid delays in pipelined machines. When the number of registers falls below a threshold, the scheduler switches to code scheduling to minimize registers usage (CSR), which essentially controls the use of registers.

Switching between CSP and CSR is driven by the number of available registers, AVLREG. AVLREG is increased when a live range ends, and decreased when an instruction creates live registers. CSP is responsible for code scheduling most of the time. When AVLREG falls below a threshold (say one), CSR is invoked. The goal of CSR at this point is to find the next instruction that will not increase the number of live registers, or if possible, decrease that number. That is, CSR tries to schedule an instruction that frees more registers than the number of live registers it creates. After AVLREG is restored to an acceptable value, CSP resumes scheduling. Thus, IPS performs prepass scheduling without excessively increasing the register requirements of the schedule.

The scheduling phase is subsequently followed by the global register allocation phase. The IPS scheduler is similar to the list scheduler described in Section 17.4.2. The IPS scheduler uses the data dependence graph of a basic block to perform the scheduling in each basic block.

17.6.2.2 Parallel Interference Graph Method

The integrated technique developed by Pinter is based on the coloring of a graph called the *parallel interference graph* [117]. The graph provides a single framework within which the considerations of both register allocation and instruction scheduling can be applied simultaneously. In this technique, the parallel interference graph — an interference graph that also takes into account scheduling constraints — is first constructed. By using this graph, register allocation is carried out, which is then followed by instruction scheduling. Hence, this is a postpass method.

The parallel interference graph combines properties of the traditional interference graph and the scheduling graph. However, a simple combination of the two graphs is not possible because the vertices in the two graphs represent different aspects: the vertices in the interference graph stand for symbolic or virtual registers in the program whereas the vertices in the scheduling graph correspond to instructions in the program. Likewise, an edge in the interference graph indicates an interference of live ranges of two symbolic registers, whereas an edge in the scheduling graph represents a precedence constraint between two instructions.

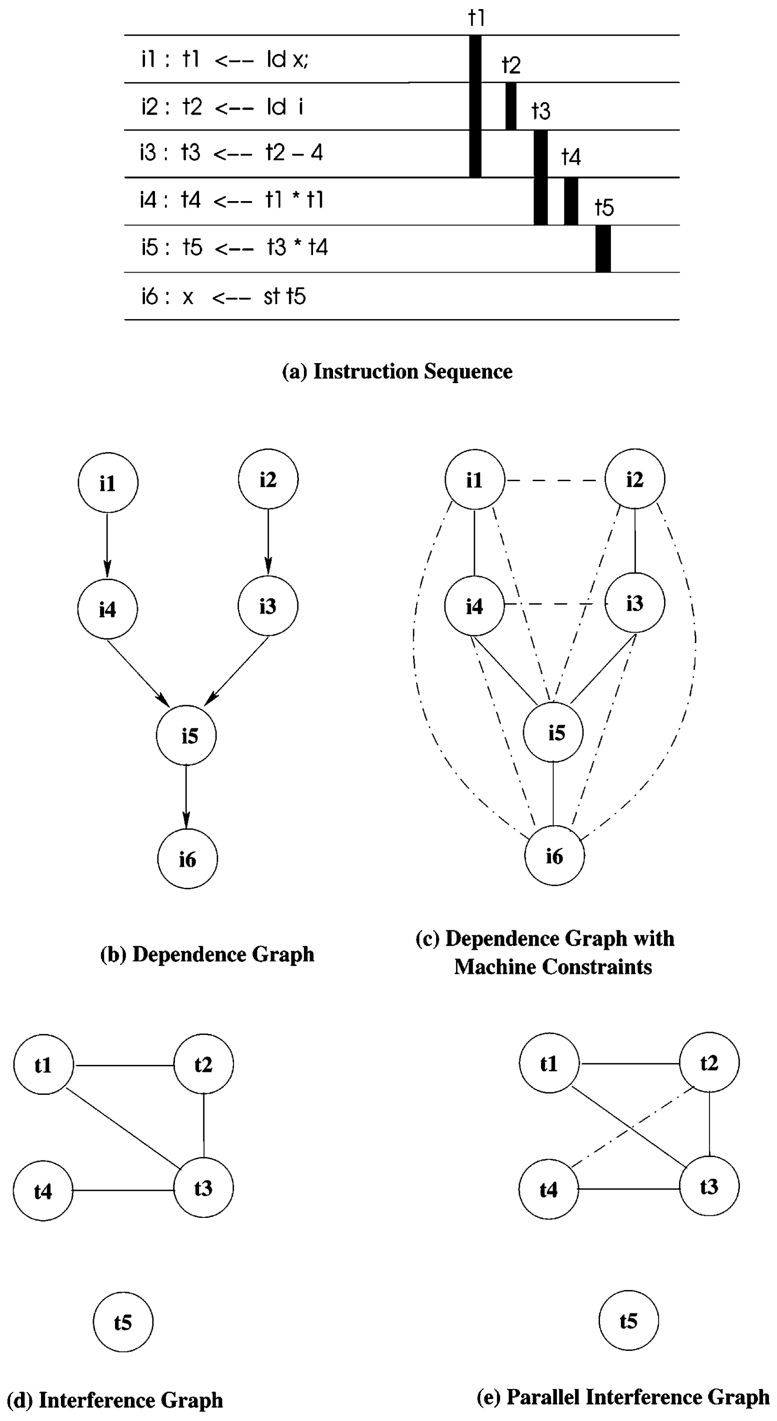
To see how the two graphs are combined, consider the example code sequence shown in Figure 17.24(a). The live ranges of variables are also shown in the same figure. Figure 17.24(b) shows the DAG for the code. The DAG gives the precedence constraints of the program. The transitive closure of this graph is generated and the edge directions are removed — refer to Figure 17.24(c). The transitive closure edges are shown as dash–dot lines in the graph in Figure 17.24(c). To this new graph all the machine-related dependencies that are not of precedence type are added. For example, consider a target machine with only one integer unit and one load and store unit. Then, instructions `i1`, `i2` and `i6` that execute on the load and store unit form a group. Similarly, instructions `i3`, `i4` and `i5` that execute on the integer unit form another group. Any pair of instructions in the same group cannot be executed in parallel. This constraint is represented by adding an edge between each pair of instructions in a group. A machine constraint edge, shown as a dashed line, is added only if neither a dependency edge nor a transitive closure edge already exists between that pair of instructions. Figure 17.24(c) shows the graph after transitive closure and machine-related edges are added. For example, edges `(i1,i2)` and `(i3,i4)` are machine-related edges. The edges in the complement of this graph represent the actual parallelism available in the given program. The complement graph consists of only two edges, namely, `(i1,i3)` and `(i2,i4)`. If we can ensure that the two definitions corresponding to each edge in the complement graph are given different registers, then no false dependence can be introduced by the register allocator. For example, the live ranges for `t1` and `t3`, corresponding to the complement edge (`i1,i3`), should be given different registers to ensure that no false dependences are introduced between `i1` and `i3`.

In the interference graph, nodes represent symbolic or virtual registers. An edge is added between a pair of nodes in the interference graph if their live ranges overlap. The interference graph for the code sequence is shown in Figure 17.24(d). Now, the parallel interference graph is built by adding edges from the complement graph to the interference graph, if they are not already present. In our example, the complement edge (`i2,i4`) should be added to the parallel interference graph. The resulting parallel interference graph is shown in Figure 17.24(e). An optimal coloring of this graph ensures that no false dependence can be introduced. While coloring the graph, if it is found that spill code has to be added, the scheduling edges in the interference graph are removed one at a time to avoid spilling, thus giving up some possible parallelism.

17.6.2.3 Other Integrated Methods

The unified resource allocator (URSA) method deals with function unit and register allocation simultaneously [15]. The method uses a three-phase measure–reduce–assign approach, where resource requirements are measured and program regions of excess requirements are identified in the first phase. The second phase reduces the requirements to what is available in the architecture, and the final phase carries out resource assignment. Norris and Pollock [107] proposed a cooperative scheduler-sensitive global-register allocator, which is followed by local instruction scheduling. The scheduler-sensitive global register allocator is a graph-coloring allocator that takes into consideration the scheduler's objectives throughout each phase of its allocation. The potential for code reordering is reflected in the construction of the interference graph. Scheduling constraints and possibilities are also taken into consideration when the allocator cannot find a coloring and decides to spill.

Bradlee, Eggers and Henry [18] developed an integrated approach called RASE in which a prescheduling phase is run to calculate cost estimates for guiding the register allocator. A global



```
i1 : t1 <-- ld x;
i2 : t2 <-- ld i
i3 : t3 <-- t2 - 4
i4 : t4 <-- t1 * t1
i5 : t5 <-- t3 * t4
i6 : x  <-- st t5
```

FIGURE 17.24 Construction of parallel interference graph.

register allocator then uses the cost estimates and spill costs to obtain an allocation and to determine a limit on the number of local registers for each block. A final scheduler is run using the register limit from allocation and inserting spill code as it schedules.

Combined register allocation and instructions scheduling problem (CRISP) has been studied by Motwani et al. in [100]. They formulate the problem as a single optimization problem and proposed an efficient heuristic algorithm, called (α, β)-*combined algorithm*. The parameters α and β provide relative weightage for register pressure and instruction-level parallelism.

17.6.2.4 Evaluation of Integrated Methods

Several studies have compared the prepass and postpass scheduling methods with integrated techniques [16, 18, 21, 108]. In [18], Bradlee, Eggers and Henry compared three code generation strategies, namely, postpass, integrated prepass scheduling and their own integrated technique called RASE. Their study, conducted for a statically scheduled in-order issue processor, demonstrated that while some level of integration is necessary to produce efficient schedules, the implementation and compilation expense of strategies that very closely couple the two phases is unnecessary. Chang et al. studied the importance of prepass scheduling using the IMPACT compiler in [22]. Their method applies both prepass and postpass scheduling to control-intensive nonscientific applications. Their study considers single issue, superscalar and superpipelined processors. Further their evaluation also included superblock scheduling [72]. Their study reveals that prepass scheduling does not improve the performance in control-intensive applications, when a restricted percolation model was used. With a more general code motion, scheduling before register allocation is important to achieve good speedup, especially for machines with 48 or more registers.

In [108], Norris and Pollock describe a strategy for providing cooperation between register allocation and instruction scheduling. They considered both global and local instruction scheduling techniques. They experimentally compared their strategy with other cooperative and noncooperative techniques. Their results suggest either cooperative or noncooperative global instruction scheduling phase, followed by register allocation that is sensitive to the subsequent local instruction scheduling, and local instruction scheduling yields good performance over noncooperative methods. Berson, Gupta and Soffa [16] compared two previous integrated strategies [52, 107] with their strategy [15], which is based on register reuse dags for measuring the register pressure. They have evaluated register spilling and register-splitting methods for reducing register requirements. They studied the performance of the preceding methods on a six-issue VLIW architecture. Their results reveal that (1) the importance of integrated methods is more significant for programs with higher register pressure, (2) methods that use precomputed information (prior to instruction scheduling) on register demands perform better than the ones that compute register demands on-the-fly (e.g., using register pressure as an index for register demands) and (3) live range splitting is more effective than live-range spilling.

17.6.3 Phase Ordering in Out-of-Order Issue Processors

Many modern processors (e.g., MIPS R10000 [156], DEC Alpha 21264 [79] and the AMD K5 [134]), support out-of-order (o-o-o) issue. In an o-o-o issue processor, instructions are scheduled dynamically with the help of complex hardware support mechanisms such as register renaming and instruction window. Register renaming is a technique by which logical registers are mapped to hardware physical registers or locations in the reorder buffer [136]. Such mapping removes anti- and output dependences and hence exposes greater ILP in the program. Further, the number of available physical registers is typically larger (roughly twice) than the number of logical registers visible to the register allocator. The instruction window holds the fetched and decoded instructions; the dynamic issue hardware selects data-ready instructions from the window and issues them. Instructions may be issued in an order different from the original program order. The register renaming mechanism and the reorder buffer together remove anti- and output dependences. This, in spirit, is similar to what the integrated register allocation and instruction scheduling techniques do at compile time. This makes the issues in phase ordering for o-o-o issue processors to be different from that for statically scheduled processors, namely, in-order issue and VLIW processors.

17.6.3.1 Evaluation of Phase Ordering in Out-of-Order Processors

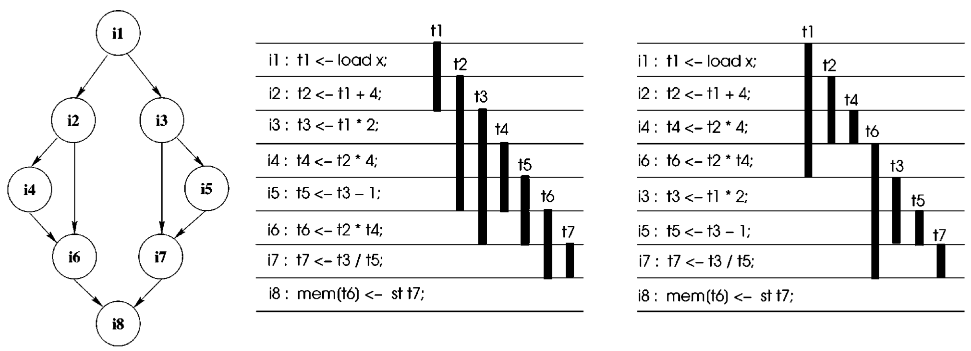
The phase-ordering problem in the context of o-o-o issue has been studied in [148, 149]. The study investigates (1) whether complex compile-time techniques do improve the overall performance and (2) whether a prepass-like or a postpass-like approach should be followed for o-o-o issue processors. The study observes an insignificant improvement in performance due to integrated methods, when scheduling is limited to basic blocks. Further, it advocates postpass-like methods, because it is important to minimize register spills in o-o-o issue processors, even at the expense of obscuring some instruction-level parallelism [148, 149].

17.6.3.2 Minimum Register Instruction Sequencing

Recall the optimal code generation problem and a solution to it, the SU method for integrated code generation and register allocation, discussed in Section 17.3.3.1. This problem is revisited in the context of o-o-o issue superscalar processors in [57]. The problem addressed in this work is that of obtaining an instruction sequence for a DAG that uses the minimum number of registers. This problem, termed as *minimum register instruction sequencing* (MRIS), is motivated by the fact that in o-o-o issue processors it is important to reduce the number of register spills, even at the expense of not exposing instruction level parallelism. The MRIS problem and its solution take into account neither the resource constraints in the architecture nor the execution latencies of instructions. The emphasis of this method is to generate an instruction sequence instead of a schedule.

Let us motivate the MRIS problem with the help of an example. Consider the computation represented by the DAG shown in Figure 17.25. Two possible instruction sequences for this DAG are also shown in the figure along with the live ranges of the variables `t1` to `t7`. For the instruction sequence A shown in Figure 17.25(b) four variables are simultaneously live during instruction `i5`; therefore, 4 registers are required for sequence A. If the number of available registers is fewer than 4, sequence A results in spill loads and stores. However, for sequence B shown in Figure 17.25(c), only three variables are simultaneously live and therefore this sequence requires only 3 registers. In this particular example, the minimum register requirement is three. Hence, the sequence shown in Figure 17.25(c) is one of the minimum register sequences.

A solution to the MRIS problem proposed in [57] proceeds by identifying which instructions can share the same register in any legal instruction sequence. A complete answer to this question is



```
i1 : t1 <- load x;
i2 : t2 <- t1 + 4;
i3 : t3 <- t1 * 2;
i4 : t4 <- t2 * 4;
i5 : t5 <- t3 - 1;
i6 : t6 <- t2 * t4;
i7 : t7 <- t3 / t5;
i8 : mem(t6) <- st t7;
```

```
i1 : t1 <- load x;
i2 : t2 <- t1 + 4;
i4 : t4 <- t2 * 4;
i6 : t6 <- t2 * t4;
i3 : t3 <- t1 * 2;
i5 : t5 <- t3 - 1;
i7 : t7 <- t3 / t5;
i8 : mem(t6) <- st t7;
```

(a) Dependence Graph **(b) Instruction Sequence A** **(c) Instruction Sequence B**

FIGURE 17.25 Instruction sequences with different register requirements.

known to be NP-hard [48]. The approach proposed in [57] uses the notion of an instruction lineage, which corresponds to a sequence of instructions that forms a path in the DAG. That is, a sequence of instructions {`i1`, `i2`, `i3`, ... , `in`} in the DAG where `i2` is the successor of `i1`, `i3` is the successor of `i2` and so on. When {`i1`, `i2`, ... , `in`} forms a lineage, the instructions in a lineage share the same register. That is, the register assigned to `i1` is passed on to `i2` (`i1`'s heir), which is passed on to `i3`, and so on. Due to data dependence between pairs of instructions in the lineage, any legal sequence orders the instructions as `i1`, `i2`, ... , `in`. Hence, the instructions in a lineage can certainly share the same destination register.

When an instruction `i1` has more than one successor, one of the successors, say `i2`, is chosen as the legal heir. To make `i2` as the last use instruction of `i1`, and hence reuse the destination register, sequencing arcs are added from each successor of `i1` to the chosen heir `i2`. For example, for the DAG shown in Figure 17.25(a), L1 = `[i1,i3,i7,i8)` forms a lineage. Typically the last node in a lineage is either a store node (in this case, `i8`) or is already in some other lineage. Thus, all instructions in a lineage except the last one share the same destination register. To emphasize this fact that the last instruction in a lineage does not use the same destination register, a semiopen interval notation is used for a lineage, as in L1 = `[i1,i3,i7,i8)`. Because instruction `i3` is chosen as the heir of `i1`, a sequencing edge is added from `i2` to `i3`. A simple but efficient heuristic based on the maximum distance (MaxDistance), measured in terms of the path length to the sink node, is used to select heirs. If the MaxDistance heuristic used is dynamic (i.e., it is calculated after the introduction of each set of sequencing edges), then the introduction of sequencing edges does not introduce cycles in the DAG [57]. The remaining lineages for the DAG are: L2 = `[i2,i6,i8)`, L3 = `[i5,i7)` and L4 = `[i4,i6)`.

To address the question whether the live ranges of two lineages definitely overlap in any legal schedule, a sufficient condition is established in [57]. The sufficient condition tests whether there exists a path from the start node of lineage L1 to the end node of L2 and vice versa. If such paths exist, then the live ranges of two lineages overlap in all legal sequences and the lineages cannot share the same register. In our example, lineages L1 and L2 overlap, as do the pairs of lineages (L1, L3), (L1, L4), (L2, L3) and (L2, L4). However, lineages L3 and L4 do not necessarily overlap in all sequences. Hence, they can be made to share the same register. Doing so would result in sequencing the execution of some of the instructions due to false dependences. However, in an o-o-o issue processor, these false dependences would be removed at runtime, and hence the parallelism exposed.

Based on the overlap relation, a lineage interference graph is constructed and colored using a traditional graph-coloring algorithm. The number of colors required to color the graph is a heuristic lower bound on the minimum registers required. By using this lower bound as a guideline, a modified list-scheduling method is used to generate a sequence that results in a near-optimal solution to the MRIS problem. This approach to the MRIS problem was found to be very effective in reducing the register pressure and in reducing the number of spill loads and stores in a number of SPEC benchmarks. Although the sequencing method does not take into consideration resource constraints and execution latencies, the execution time of the generated sequence was found to be comparable with that generated by a production quality compiler.

17.6.3.3 Linearization of Instruction Schedule

A superscalar processor expects a linear sequence of instructions. Hence, a parallel instruction schedule, such as the one shown in Figure 17.12, is presented to a superscalar processor by linearizing it in a simple way. The linearization method sequences the instructions in each cycle of the schedule in a left to right order. However, simple linearization methods are not aware of the register renaming capabilities of o-o-o issue superscalar processors and may generate a sequence that would have higher register pressure. As a consequence it may result in spill code. Further, the linearization does not take into account the size of the instruction window [136], the register renaming capabilities of superscalar architecture and the in-order graduation mechanism. These may result in certain inefficiencies in the form of stall cycles. An efficient linearization method that is sensitive to register pressure and is aware of the architectural features of o-o-o issue superscalar processors has been proposed in [133].

The linearization proposed in [133] uses a set of matching conditions that ensure the ILP available in the given parallel schedule is not lost in the linearization process. The linearization method is an extension of the list scheduling method and adds instructions to the linear sequence in such a way that it reduces the register pressure without losing any parallelism compared with the given parallel schedule. The method was applied to basic blocks.

17.7 Recent Research in Instruction Scheduling

In this section we report some of the recent research work on instruction scheduling.

17.7.1 Instruction Scheduling for Application-Specific Processors

Instruction scheduling methods have been proposed for application-specific processors, such as DSP. Originally most DSP applications, or their important kernels, were hand coded in assembly language. The application programmer is required to perform the necessary instruction reordering to take full advantage of the parallelism that is available in these processors. With the increasing complexity of the processors and their programmability, programming in higher level languages and compilation techniques to produce efficient code automatically are becoming increasingly important. A major challenge in applying existing instruction scheduling methods arises due to the irregularities of DSP processors [88]. These irregularities include having special-purpose registers in the data path; heterogeneous registers; dedicated memory address generation units; and chained instructions such as multiply-and-accumulate, saturation arithmetic, multistage functional units and parallel memory banks. Further, in most cases, code running on a DSP processor also has to meet real-time constraints. Thus, instruction scheduling for DSP processors, which needs to take into account the irregularities of the architectures and the real-time constraints in resource usage, poses a major challenge.

Several instruction scheduling methods for DSP processors have been proposed. A DSP-specific code compaction technique has been developed in [142], which considers both resource and timing constraints. Instruction scheduling for the TriMedia VLIW processor [116] has been reported in [67]. The instruction scheduling problem is transformed into an integer linear program problem in [89]. Another integer linear program formulation for integrated instruction scheduling and register allocation has been proposed in [19]. Methods for simultaneous register allocation and instruction scheduling for DSP processors (involving heterogeneous registers) have been proposed in [27, 91]. An extensive survey of code generation for signal-processing systems has been presented in [17].

Certain DSP processors (e.g., the Texas Instruments TMS320C series [140]) support multiple operating modes, such as the sign–extension mode and product–shift mode, which provide slightly different execution semantics for instructions [8]. Multiple operating modes raise another interesting instruction scheduling problem. Here the objective is to schedule instructions, making use of the multiple modes, while reducing the number of mode-setting instructions required, and hence the associated overhead cost.

Code size is another key concern in application-specific processors. Because code size relates to on-chip program memory in these processors, code size can directly influence the cost of the system. Hence, compilation methods in general, and instruction scheduling methods in particular, optimize the code not only for performance, but also for code size. In the presence of code size constraints, the scope of global code-scheduling methods is limited, because these methods are known for their code-bloating problem.

The TMS C6x DSP processor, well known for its compiler-friendly architecture, has a cluster of functional units [140]. Each cluster has a register file of its own. Each functional unit in a cluster can access two read ports and one write port to its local register file. In addition, at most one functional unit in a cluster can access a register file in a cross cluster in any given time step. If more than one functional unit need to access data across clusters, or multiple accesses to a cross register file are needed, it is desirable to explicitly copy these data to a local register.

In this type of architecture, known as *clustered architecture*, associated with the instruction scheduling is the problem of mapping, which assigns an instruction and its destination operand to a cluster. Methods that perform instruction scheduling before mapping [20] or mapping before scheduling [123] could result in poor schedules, because the first phase makes certain decisions without knowing their consequences on the subsequent phase. To take full advantage of the ILP that can be exploited in this architecture, a compiler needs to perform instruction scheduling along with instruction mapping. This requires that the two problems, assignment and scheduling, be solved in a unified framework as in [90, 110]. A modulo-scheduling method for clustered architecture is discussed in [109, 129].

17.7.2 Instruction Scheduling for Low Power

Another area that has been receiving increasing attention is instruction scheduling for low power in embedded processors [114]. Embedded processors are used in many handheld devices, such as cell phones, pagers, digital cameras and toys, in which battery longevity and its size are key factors that determine system cost. Because the overall power dissipated or energy consumed directly relates to battery life, embedded processors generally have low-power requirements. In these systems, it is quite common that an application is compiled more for power efficiency than for performance. By reordering instructions the power dissipated can be decreased.

The transition or switching activities — toggling of signals from zero to one or vice versa — that take place on the system bus, more specifically on the instruction bus, can be reduced by instruction reordering methods, which in turn help to reduce power. Su, Tsui and Despain [139] propose a technique that reorders instructions in such a way that the toggles between the encodings or machine codes of adjacent instructions are reduced. This is accomplished by a simple list scheduling method that uses a priority function that gives a higher priority to an instruction in the ReadyList that has the lowest power cost. Power cost of an instruction is estimated based on the last scheduled instruction in the partial schedule and a power cost table. The essential idea of this method is to reduce the amount of switching activities between adjacent instructions to reduce the power consumed.

Another scheduling method for reducing power consumption is presented by Tiwari, Malik and Wolfe [143]. The goal in this work is to judiciously select instructions as opposed to reordering them to reduce power consumption. This approach uses a power table that contains power consumed by individual instructions as well as certain commonly paired instructions. By using this power table, code is rescheduled to use instructions that result in less power consumption. In [144], a method to reduce the peak power dissipation has been proposed. This method uses a predefined per cycle energy dissipation threshold, and limits the number of instructions that can be scheduled in a given cycle based on this threshold. A method to reduce the power consumption on the instruction bus of a VLIW architecture has been proposed in [87]. Each VLIW instruction, referred to as a long word instruction, consists of a number of instructions or operations. This method uses a greedy approach to reschedule operations within a long instruction word and across long instruction words, but within a limited instruction window size, to reduce the switching activities among the instruction. When the operations are rescheduled across long word instructions, dependence constraints are preserved. The method attempts to obtain a schedule that consumes low power, without sacrificing performance. Several scheduling strategies that attempt to reduce the energy consumption with and without sacrificing performance have been evaluated in [112].

More recently, a number of scheduling methods [82, 132] have been proposed that deal with voltage and frequency scaling, an approach in which the operating voltage or the operating frequency is scaled down to reduce the power consumption.

17.8 Summary

Instruction-scheduling methods rearrange instructions in a code sequence to expose ILP for multiple instruction issue processors and to reduce the number of stall cycles incurred in a single-issue, pipelined processor. Simple scheduling methods that cover pipeline stalls use information on the number of stall cycles required between dependent instructions. Basic block instruction scheduling methods are limited to rearranging instructions in a straight-line code sequence with a single control flow entry and exit. In this chapter, we have reviewed several approaches to basic block instruction scheduling, including list scheduling, operation scheduling, and integer linear programming based methods. The heuristics used in list-scheduling methods and resource models used for modeling complex resource usage patterns have also been discussed.

Global scheduling refers to instruction scheduling that extends beyond instructions in a basic block. In the case of global acyclic scheduling, the control flow graph on which the scheduling method is applied is acyclic. Trace scheduling and superblock scheduling consider acyclic control flow subgraphs that consist of a single control flow path. In hyperblock scheduling and treegion scheduling, multiple control flow paths can be explored in a single trace. Software-pipelining methods schedule multiple instances of either single or multiple static basic blocks, corresponding to multiple iterations, of a cyclic control flow graph.

Register allocation is a closely related ILP compilation issue. Instruction scheduling and register allocation phases in an optimizing compiler mutually influence each other. Issues related to the phase ordering of instruction scheduling and register allocation in both statically scheduled and dynamically scheduled multiple instruction issue processors have been discussed in this chapter. The need to reduce register spills, even at the expense of obscuring some ILP, has resulted in new approaches for instruction sequencing. Finally, recent research on the application of instruction scheduling methods for application-specific processors and low-power embedded systems has been briefly discussed.

Acknowledgments

The author is grateful to Jose Nelson Amaral, Guang R. Gao, Matthew Jacob and Hongbo Yang for their numerous suggestions on the various drafts of this chapter; and to Bhama Sridharan and T.S. Subashree for proofreading an initial draft. The author acknowledges Jose Nelson Amaral, Erik R. Altman, A. Dani, N.S.S.Narasimha Rao, S.M. Sandhya, V. Janaki Ramanan, Raul Silvera, Madhavi G. Valluri, Jian Wang, Hongbo Yang and Chihong Zhang, with whom he has co-authored several papers on instruction scheduling and software pipelining that have both directly and indirectly contributed to this chapter. Finally, the research funding received from the Department of Science and Technology, India, which partly supported this work, is acknowledged.

References

[1] S.G. Abraham, V. Kathail and B.L. Deitrich, Meld Scheduling: Relaxing Scheduling Constraints Across Region Boundaries, in Proceedings of the 29th Annual International Symposium on Microarchitecture, Paris, December 1996, pp. 308–321.

[2] A.V. Aho, S.C. Johnson and J.D. Ullman, Code generation for expressions with common subexpressions, *J. ACM*, 24(1), 146–160, January 1977.

[3] A.V. Aho, R. Sethi and J.D. Ullman, *Compilers — Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA, corrected edition, 1988.

[4] V.H. Allan and S.J. Allan, Software pipelining, in *The Compiler Design Handbook: Optimization and Machine Code Generation*, Y.N. Srikant and P. Shankar, Eds., CRC Press, Boca Raton, FL, 2002.

[5] V.H. Allan, R. Jones, R. Lee and S.J. Allan, Software pipelining, *ACM Comput. Surv.*, 27(3), 367–432, June 1974.

[6] J.R. Allen, K. Kennedy, C. Porterfield and J. Warren, Conversion of Control Dependence to Data Dependence, in Conference Record of the 10th Annual ACM Symposium on Principles of Programming Languages, Austin, TX, January 1983, pp. 177–189.

[7] E.R. Altman, R. Govindarajan and G.R. Gao, Scheduling and Mapping: Software Pipelining in the Presence of Structural Hazards, in Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation, La Jolla, CA, June 1995, pp. 139–150.

[8] G. Araujo, S. Devadas, K. Keutzer, S. Liao, S. Malik, A. Sudarsanam, S. Tjiang and A. Wang, Challenges in code generation for embedded processors, in *Code Generation for Embedded Processors*, P. Marwedel and G. Goossens, Eds., Kluwer Academic, Boston, MA, 1995.

[9] S. Arya, An optimal instruction-scheduling model for a class of vector processors, *IEEE Trans. Comput.*, 34(11), 981–995, November 1985.

[10] M. Auslander and M. Hopkins, An Overview of the PL.8 Compiler, in Proceedings of the SIGPLAN '82 Symposium on Compiler Construction, Boston, MA, June 1982, pp. 22–31.

[11] V. Bala and N. Rubin, Efficient Instruction Scheduling Using Finite State Automata, in Proceedings of the 28th Annual International Symposium on Microarchitecture, Ann Arbor, MI, November 1995, pp. 46–56.

[12] T. Ball and J.R. Larus, Optimally profiling and tracing programs, *ACM Trans. Programming Languages Syst.*, 16(4), 1319–1360, July 1994.

[13] G.R. Beck, D.W.L. Yen and T.L. Anderson, The Cdra 5 minisupercomputer: architecture and implementation, *J. Supercomput.*, 7, 143–180, May 1993.

[14] D. Bernstein and M. Rodeh, Global Instruction Scheduling for Superscalar Machines, in Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, Toronto, Ontario, June 1991, pp. 241–255.

[15] D.A. Berson, R. Gupta and M.L. Soffa, URSA: A unified resource allocator for registers and functional units in VLIW architectures, in *Proceedings of the IFIP WG 10.3 Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism* (number A-23 in IFIP Transactions), M. Cosnard, K. Ebcioğlu and J-L. Gaudiot, Eds., 1993, North-Holland, Amsterdam, pp. 243–254.

[16] D. Berson, R. Gupta and M.L. Soffa, An evaluation of integrated scheduling and register allocation techniques, in *Proceedings of the 11th International Workshop on Languages and Compilers for Parallel Computing, Lecture Notes in Computer Science*, Springer-Verlag, 1998.

[17] S.S. Bhattacharyya, R. Leupers and P. Marwedel, Software synthesis and code generation for signal processing systems, *IEEE Trans. Circuits Syst.* II, 47(9), September 2000.

[18] D.G. Bradlee, S.J. Eggers and R.R. Henry, Integrating Register Allocation and Instruction Scheduling for RISCs, in Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems, Santa Clara, CA, April 1991, pp. 122–131.

[19] T. Bruggen and A. Ropers, Optimized Code Generation for Digital Signal Processors, Technical report, Institute for Integrated Signal Processing Systems, Aachen, Germany, August 1999.

[20] A. Capitanio, N. Dutt and A. Nicolau, Partitioned Register Files for VLIWs: A Preliminary Analysis of Tradeoffs, in Proceedings of the 25th Annual International Symposium on Microarchitecture, Portland, OR, December 1992, pp. 292–300.

[21] P.P. Chang, D.M. Lavery, S.A. Mahlke, W.Y. Chen and W.W. Hwu, The importance of prepass code scheduling for superscalar and superpipelined processors, *IEEE Trans. Comput.*, 44(3), 353–370, March 1995.

[22] P.P. Chang, S.A. Mahlke, W.Y. Chen, N.J. Warter and W.W. Hwu, IMPACT: An Architectural Framework for Multiple-Instruction-Issue Processors, in Proceedings of the 18th Annual International Symposium on Computer Architecture, Toronto, Ontario, May 1991, pp. 266–275.

[23] L.-F. Chao and E. H.-M. Sha, Rate-Optimal Static Scheduling for DSP Dataflow Programs. in Proceedings of 1993 Great Lakes Symposium on VLSI, March 1993, pp. 80–84.

[24] A.E. Charlesworth, An approach to scientific array processing: the architectural design of the AP-120B/FPS-164 family, *Computer*, 14(9), 18–27, September 1981.

[25] S. Chaudhuri, R.A. Walker and J.E. Mitchell, Analyzing and exploiting the structure of the constraints in the ILP approach to the scheduling problem, *IEEE Trans. Very Large Scale Integration (VLSI) Syst.*, 2(4), 456–471, December 1994.

[26] C. Chekuri, R. Motwani, R. Johnson, B. Natarajan, B. R. Rau and M. Schlansker, Profile-Driven Instruction Level Parallel Scheduling with Applications to Super Blocks, in Proceedings of the 29th Annual International Symposium on Microarchitecture, Paris, December 1996, pp. 58–67.

[27] W.-K. Cheng and Y.-L. Lin, Code generation of nested loops for DSP processors with heterogeneous registers and structural pipelining, *ACM Trans. Design Automation Electron. Syst. (TODAES)*, 4(3), 231–256, July 1999.

[28] H.-C. Chou and C.-P. Chung, An optimal instruction scheduler for superscalar processor, *IEEE Trans. Parallel Distributed Syst.*, 6(3), 303–313, March 1995.

[29] E.G. Coffman, *Computer and Job-Shop Scheduling Theory*, John Wiley & Sons, New York, 1976.

[30] D. Cohen, A Methodology for Programming a Pipeline Array Processor, in Proceedings of the 10th Annual Microprogramming Workshop, Niagara Falls, NY, October 1977, pp. 82–89.

[31] R.P. Colwell, R.P. Nix, J.J. O'Donnell, D.B. Papworth and P.K. Rodman, A VLIW architecture for a trace scheduling compiler, *IEEE Trans. Comput.*, 37(8), August 1988.

[32] A. Dani, V. Janaki Ramanan and R. Govindarajan, Register-Sensitive Software Pipelining, in Proceedings of the 12th International Parallel Processing Symposium and the 9th Symposium on Parallel and Distributed Processing, Orlando, FL, April 1998.

[33] J.C. Dehnert, P.Y.-T. Hsu and J.P. Bratt, Overlapped loop support in the Cydra 5, in Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems, Boston, MA, April 1989, pp. 26–38.

[34] J.C. Dehnert and R.A. Towle, Compiling for Cydra 5, *J. Supercomput.*, 7, 181–227, May 1993.

[35] B.L. Deitrich and W.W. Hwu. Speculative Hedge: Regulating Compile-Time Speculation against Profile Variations, in Proceedings of the 29th Annual International Symposium on Microarchitecture, Paris, December 1996, pp. 70–79.

[36] K. Ebcioğlu and A. Nicolau, A Global Resource-Constrained Parallelization Technique, in Conference Proceedings, 1989 International Conference on Supercomputing, Crete, Greece, June 1989, pp. 154–163.

[37] A.E. Eichenberger and E.S. Davidson, Stage scheduling: A Technique to Reduce the Register Requirements of a Modulo Schedule, in Proceedings of the 28th Annual International Symposium on Microarchitecture, Ann Arbor, MI, November 1995, pp. 338–349.

[38] A.E. Eichenberger and E.S. Davidson, A Reduced Multipipeline Machine Description that Preserves Scheduling Constraints, in Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation, Philadelphia, May 1996, pp. 12–22.

[39] A.E. Eichenberger and E.S. Davidson, Efficient formulation for optimal modulo schedulers, in Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation, Las Vegas, NV, June 1997, pp. 194–205.

[40] A.E. Eichenberger, E.S. Davidson and S.G. Abraham, Minimum Register Requirements for a Modulo Schedule, in Proceedings of the 27th Annual International Symposium on Microarchitecture, San Jose, CA, November 1994, pp. 75–84.

[41] J.R. Ellis, Bulldog: A Compiler for VLIW Architectures, Research report YALEU/DCS/RR-364, Department of Computer Science, Yale University, New Haven, CT, 1984.

[42] P. Feautrier, Fine-grain scheduling under resource constraints, in *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*, K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau and D. Padua, Eds., *Lecture Notes in Computer Science*, Vol. 892, Springer-Verlag, 1995, pp. 1–15.

[43] J. Ferrante, K.J. Ottenstein and J.D. Warren, The program dependence graph and its use in optimization, *ACM Trans. Programming Languages Syst.*, 9(3), July 1987, 319–349.

[44] J.A. Fisher, D. Landskov and B.D. Shriver, Microcode Compaction: Looking Backward and Looking Forward, in Proceedings of the 1981 National Computer Conference, New Orleans, LA, 1981, pp. 95–102.

[45] J.A. Fisher, Trace scheduling: a technique for global microcode compaction, *IEEE Trans. Comput.*, 30(7), 478–490, July 1981.

[46] J.A. Fisher, Very Long Instruction Word Architectures and the ELI-512, in Proceedings of the 10th Annual International Symposium on Computer Architecture, Stockholm, Sweden, June 1983, pp. 140–150.

[47] J.A. Fisher, Global Code Generation for Instruction-Level Parallelism: Trace Scheduling-2, Technical Report, HPL-93-43, Hewlett-Packard Laboratory, Palo Alto, CA, June 1993.

[48] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman, New York, 1979.

[49] F. Gasperoni and U. Schwiegelshohn, Efficient Algorithms for Cyclic Scheduling, Research Report RC 17068, IBM T. J. Watson Research Center, Yorktown Heights, NY, 1991.

[50] P.B. Gibbons and S.S. Muchnick, Efficient Instruction Scheduling for a Pipelined Architecture, in Proceedings of the SIGPLAN ’86 Symposium on Compiler Construction, Palo Alto, CA, June 1986, pp. 11–16.

[51] M.C. Golumbic and V. Rainish, Instruction scheduling beyond basic blocks, *IBM J. Res. Dev.*, 34(1), 93–97, January 1990.

[52] J.R. Goodman and W.-C. Hsu, Code Scheduling and Register Allocation in Large Basic Blocks, in Conference Proceedings, 1988 International Conference on Supercomputing, St. Malo, France, July 1988, pp. 442–452.

[53] R. Govindarajan, E.R. Altman and G.R. Gao, Minimizing Register Requirements under Resource-Constrained Rate-Optimal Software Pipelining, in Proceedings of the 27th Annual International Symposium on Microarchitecture, San Jose, CA, November 1994, pp. 85–94.

[54] R. Govindarajan, E.R. Altman and G.R. Gao, Co-scheduling Hardware and Software Pipelines, in Proceedings of the Second International Symposium on High-Performance Computer Architecture, San Jose, CA, February 1996, pp. 52–61.

[55] R. Govindarajan, E.R. Altman and G.R. Gao, A framework for resource-constrained rate-optimal software pipelining, *IEEE Trans. Parallel Distributed Syst.*, 7(11), 1133–1149, November 1996.

[56] R. Govindarajan, N.S.S. Narasimha Rao, E.R. Altman and G.R. Gao, Enhanced Co-Scheduling: a software pipelining method using modulo-scheduled pipeline theory, *Int. J. Parallel Programming*, 28(1), 1–46, February 2000.

[57] R. Govindarajan, H. Yang, C. Zhang, J. Nelson Amaral and G.R. Gao, Minimum Register Instruction Sequence Problem: Revisiting Optimal Code Generation for Dags, in Proceedings of the International Parallel and Distributed Processing Symposium, San Francisco, CA, April 2001.

[58] R. Gupta, E. Mehofer and Y. Zhang, Profile guided compiler optimizations, in Y.N. Srikant and P. Shankar, Eds., *The Compiler Design Handbook: Optimization and Machine Code Generation*, CRC Press, Boca Raton, FL, 2002.

[59] R. Gupta and M.L. Soffa, Region Scheduling, in L.P. Kartashev and S.I. Kartashev, Eds., Proceedings of the Second International Conference on Supercomputing ’87, Vol. 3, 1987, pp. 141–148.

[60] J.C. Gyllenhaal, W.W. Hwu and B.R. Rau, Optimization of Machine Descriptions for Efficient Use, in Proceedings of the 29th Annual International Symposium on Microarchitecture, Paris, December 1996, pp. 349–358.

[61] R.E. Hank, W.W. Hwu and B.R. Rau, Region-Based Compilation: An Introduction and Motivation, in Proceedings of the 28th Annual International Symposium on Microarchitecture, Ann Arbor, MI, November 1995, pp. 158–168.

[62] W.A. Havanki, S. Banerjia and T.M. Conte, Treegion Scheduling for Wide Issue Processors, in Proceedings of the 4th International Symposium on High-Performance Computer Architecture, Las Vegas, NV, February 1998, pp. 266–276.

[63] J.L. Hennessy and T.R. Gross, Code Generation and Reorganization in the Presence of Pipeline Constraints, in Conference Record of the 9th Annual ACM Symposium on Principles of Programming Languages, Albuquerque, NM, January 1982, pp. 120–127.

[64] J.L. Hennessy and T. Gross, Postpass code optimization of pipeline constraints, *ACM Trans. Programming Languages Syst.*, 5(3), 422–448, July 1983.

[65] J.L. Hennessy, N. Jouppi, F. Baskett, T. Gross and J. Gill, Hardware/Software Tradeoffs for Increased Performance, in Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems, Palo Alto, CA, March 1982, pp. 2–11.

[66] J.L. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach*, 2nd ed., Morgan Kaufmann, San Francisco, 1996.

[67] J. Hoogerbrugge and L. Augusteijn, Instruction scheduling for TriMedia, *J. Instruction-Level Parallelism*, 1(1), 1999, *www.jilp.org*.

[68] P.Y.-T. Hsu and E.S. Davidson, Highly Concurrent Scalar Processing, in Proceedings of the 13th Annual International Symposium on Computer Architecture, Tokyo, June 1986, pp. 386–395.

[69] P.Y.-T. Hsu, Design of the R-8000 Microprocessor, Technical report, MIPS Technologies Inc., Mountainview, CA, June 1994.

[70] R.A. Huff, Lifetime-Sensitive Modulo Scheduling, in Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation, Albuquerque, NM, June 1993, pp. 258–267.

[71] W.W. Hwu, R.E. Hank, D.E. Gallagher, S.A. Mahlke, D.M. Lavery, G.E. Haab, J.C. Gyllenhaal and D.I. August, Compiler technology for future microprocessors, *Proc. IEEE*, 83(12), 1625–1640, December 1995.

[72] W.W. Hwu, S.A. Mahlke, W.Y. Chen, P.P. Chang, N.H. Warter, R.G. Ouellette, R.E. Hank, T. Kyohara, G.E. Haab, J.G. Holm and D.M. Lavery, The superblock: an effective technique for VLIW and superscalar compilation, *J. Supercomput.*, 7, 229–248, May 1993.

[73] V. Janaki Ramanan, Efficient Resource Usage Modelling, M.Sc.(Eng.) thesis, Supercomputer Education and Research Centre, Indian Institute of Science, Bangalore, India, April 1999.

[74] V. Janaki Ramanan and R. Govindarajan, Resource Usage Models for Instruction Scheduling: Two New Models and a Classification, in Conference Proceedings of the 1999 International Conference on Supercomputing, Rhodes, Greece, June 1999.

[75] M. Johnson, *Superscalar Microprocessor Design*, Prentice Hall, Englewood Cliffs, NJ, 1991.

[76] G. Kane, *MIPS R2000 RISC Architecture*, Prentice Hall, Englewood Cliffs, NJ, 1987.

[77] V. Kathail, M.S. Schlansker and B.R. Rau, HPL PlayDoh Architecture Specification: Version 1.0, Technical report, HP Labs., Palo Alto, CA, March 1994.

[78] D. Kerns and S. Eggers, Balanced Scheduling: Instruction Scheduling when Memory Latency is Uncertain, in Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation, Albuquerque, NM, June 1993, pp. 278–289.

[79] R.E. Kessler, The Alpha 21264 microprocessor, *IEEE Micro*, 19(2), 24–36, March 1999.

[80] H. Kim, K. Gopinath and V. Kathail, Region based register allocation for EPIC processors with predication, *Parallel Computing*, 1999.

[81] P.M. Kogge, *The Architecture of Pipelined Computers*, McGraw-Hill, New York, 1981.

[82] C.M. Krishna and Y.-H. Lee, Voltage-Switching Scheduling Algorithms for Low Power in Hard Real-Time Systems, in Proceedings of the IEEE Real-Time Technology and Application Symposium, May 2000.

[83] S.M. Kurlander, T.A. Proebsting and C.N. Fischer, Efficient instruction scheduling for delayed-load architectures, *ACM Trans. Programming Languages Syst.*, 17(5), 740–776, September 1995.

[84] M. Lam, Software Pipelining: An Effective Scheduling Technique for VLIW Machines, in Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation, Atlanta, GA, June 1988, pp. 318–328.

[85] M. Lam, Instruction scheduling for superscalar architectures, *Annu. Rev. Comput. Sci.*, 4, 173–201, 1990.

[86] E. Lawler, J.K. Lenstra, C. Martel, B. Simons and L. Stockmeyer, Pipeline Scheduling: A Survey, Technical report, IBM Research Division, San Jose, CA, 1987.

[87] C. Lee, J. Lee, T. Hwang and S. Tsai, Compiler Optimization on Instruction Scheduling for Low Power, in Proceedings of the 13th International Symposium on System Synthesis (ISSS'00), Madrid, Spain, September 2000.

[88] E.A. Lee, Programmable DSP architectures: Part I, *IEEE ASSP*, 4–19, October 1988.

[89] R. Leupers and P. Marwedel, Time-constrained code compaction for DSPs, *IEEE Trans. Very Large Scale Integration Syst.*, 5(1), January 1997.

[90] R. Leupers, Instruction Scheduling for Clustered VLIW DSPs, in Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques, Philadelphia, October 2000, pp. 291–300.

[91] S. Liao, S. Devadas, K. Keutzer, S. Tjiang and A. Wang, Storage Assignment to Decrease Code Size, in Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation, La Jolla, CA, June 1995, pp. 186–195.

[92] J. Llosa, A. González, E. Ayguadé and M. Valero, Swing Modulo Scheduling: A Lifetime-Sensitive Approach, in Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT '96), Boston, MA, October 1996, pp. 80–86.

[93] J. Llosa, M. Valero, E. Ayguadé and A. González, Hypernode Reduction Modulo Scheduling, in Proceedings of the 28th Annual International Symposium on Microarchitecture, Ann Arbor, MI, November 1995, pp. 350–360.

[94] J.L. Lo and S.J. Eggers, Improving Balanced Scheduling with Compiler Optimizations that Increase Instruction-Level Parallelism, in Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation, La Jolla, CA, July 1995, pp. 151–162.

[95] P.G. Lowney, S.M. Freudeberger, T.J. Karzes, W.D. Lichtensein, R.P. Nix, J.S. O'Donnell and J.C. Ruttenberg, The Multiflow trace scheduling compiler, *J. Supercomput.*, 7, 51–142, May 1993.

[96] U. Mahadevan and S. Ramakrishnan, Instruction scheduling over regions: A framework for scheduling across basic blocks, in *Proceedings of the 5th International Conference on Compiler Construction, CC '94*, P.A. Fritzson, Ed., *Lecture Notes in Computer Science*, Vol. 786, Springer-Verlag, pp. 419–434.

[97] S.A. Mahlke, D.C. Lin, W.Y. Chen, R.E. Hank and R.A. Bringmann, Effective Compiler Support for Predicated Execution Using the Hyperblock, in Proceedings of the 25th Annual International Symposium on Microarchitecture, Portland, OR, December 1992, pp. 45–54.

[98] S. Mantripragada, S. Jain and J. Dehnert, A New Framework for Integrated Global Local Scheduling, in Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT '98), Paris, October 1998, pp. 167–174.

[99] S.-M. Moon and K. Ebcioğlu, An Efficient Resource-Constrained Global Scheduling Technique for Superscalar and VLIW Processors, in Proceedings of the 25th Annual International Symposium on Microarchitecture, Portland, OR, December 1992, pp. 55–71.

[100] R. Motwani, K. Palem, V. Sarkar and S. Reyen, Combined Instruction Scheduling and Register Allocation, Technical report TR 1995-698, Department of Computer Science, New York University, New York, NY, 1995.

[101] S.S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann, San Francisco, 1997.

[102] T. Müller, Employing Finite Automata for Resource Scheduling, in Proceedings of the 26th Annual International Symposium on Microarchitecture, Austin, TX, December 1993, pp. 12–20.

[103] A. Nicolau and S. Novack, Trailblazing: A Hierarchical Approach to Percolation Scheduling, in Proceedings of the 1993 International Conference on Parallel Processing, St. Charles, IL, August 1993.

[104] A. Nicolau and R. Potasman, Realistic Scheduling: Compaction for Pipelined Architectures, in Proceedings of the 23rd Annual Workshop on Microprogramming and Microarchitecture, Orlando, FL, November 1990, pp. 69–79.

[105] A. Nicolau, Percolation Scheduling: A Parallel Compilation Technique, Technical report TR 85-678, Department of Computer Science, Cornell University, Ithaca, NY, 1985.

[106] K.B. Normoyle, M.A. Csoppenszky, A. Tzeng, T.P. Johnson, C.D. Furman and J. Mostoufi, UltraSPARC-II: expanding the boundaries of a system on a chip, *IEEE Micro*, 18(2), 14–24, March 1998.

[107] C. Norris and L.L. Pollock, A Scheduler-Sensitive Global Register Allocator, in Proceedings of the 1993 Supercomputing Conference, November 1993.

[108] C. Norris and L.L. Pollock An Experimental Study of Several Cooperative Register Allocation and Instruction Scheduling Strategies, in Proceedings of the 28th Annual International Symposium on Microarchitecture, Ann Arbor, MI, November 1995, pp. 169–179.

[109] E. Nystrom and A.E. Eichenberger, Effective Cluster Assignment for Modulo Scheduling, in Proceedings of the 31st Annual International Symposium on Microarchitecture, Dallas, TX, November 1998, pp. 103–114.

[110] E. Özer, S. Banerjia and T. M. Conte, Unified Assign and Schedule: A New Approach to Scheduling for Clustered Register File Microarchitectures, in Proceedings of the 31st Annual International Symposium on Microarchitecture, Dallas, TX, November 1998, pp. 308–315.

[111] K.V. Palem and B.B. Simons, Scheduling time-critical instructions on RISC machines, in Conference Record of the 17th Annual ACM Symposium on Principles of Programming Languages, San Francisco, CA, January 1990, pp. 270–280.

[112] A. Parikh, M. Kandemir, N. Vijaykrishnan and M.J. Irwin, Energy-Aware Instruction Scheduling, in Proceedings of the 7th International Conference on High Performance Computing, Bangalore, India, 2000, pp. 335–344.

[113] D.A. Patterson and C.H. Sequin, RISC I: A Reduced Instruction Set VLSI Computer, in Proceedings of the 8th Annual Symposium on Computer Architecture, Minneapolis, MN, May 1981, pp. 443–457.

[114] P.G. Paulin, M. Cornero and C. Liem, Trends in embedded system technology, an industrial perspective, in *Hardware/Software Co-Design*, M. Giovanni and M. Sami, Eds., Kluwer Academic, Boston, MA, 1996.

[115] P.G. Paulin and J.P. Knight, Force-directed scheduling for the behavioral synthesis of ASIC's, *IEEE Trans. Comput.-Aided Design*, 8(6), 661–679, June 1989.

[116] Philips Semiconductors, 1995, *www.trimedia.philips.com.*

[117] S.S. Pinter, Register Allocation with Instruction Scheduling: A New Approach, in Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation, Albuquerque, NM, June 1993, pp. 248–257.

[118] T.A. Proebsting and C.N. Fischer, Linear-Time, Optimal Code Scheduling for Delayed-Load Architectures, in Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, Toronto, Ontario, June 1991, pp. 256–267.

[119] T.A. Proebsting and C.W. Fraser, Detecting Pipeline Structural Hazards Quickly, in Conference Record of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, January 1994, pp. 280–286.

[120] M. Rajagopalan and V.H. Allan, Efficient Scheduling of Fine Grain Parallelism in Loops, in Proceedings of the 26th Annual International Symposium on Microarchitecture, Austin, TX, December 1993, pp. 2–11.

[121] B.R. Rau and J.A. Fisher, Instruction-level parallel processing: History, overview and perspective, *J. Supercomput.*, 7, 9–50, May 1993.

[122] B.R. Rau and C.D. Glaeser, Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing, in *Proceedings of the 14th Annual Microprogramming Workshop, Chatham, MA, October 1981*, pp. 183–198.

[123] B.R. Rau, V. Kathail and S. Aditya, Machine-description driven compilers for EPIC and VLIW processors, *Design Automation Embedded Syst.*, 4(2/3), 1999.

[124] B.R. Rau, M. Lee, P.P. Tirumalai and M.S. Schlansker, Register Allocation for Software Pipelined Loops, in *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation, San Francisco, CA, June 1992*, pp. 283–299.

[125] B.R. Rau, Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops, in Proceedings of the 27th Annual International Symposium on Microarchitecture, San Jose, CA, November 1994, pp. 63–74.

[126] B.R. Rau, D.W.L. Yen, W. Yen and R.A. Towle, The Cydra 5 departmental supercomputer — design philosophies, decisions, and trade-offs, *Computer*, 22(1), 12–35, January 1989.

[127] R. Reiter, Scheduling parallel computations, *J. ACM*, 15(4), 590–599, October 1968.

[128] J.C. Ruttenberg, Delayed-Binding Code Generation for a VLIW Supercomputer, Ph.D. thesis, Yale University, New Haven, CT, 1985.

[129] J. Sánchez and A. González, Modulo Scheduling for a Fully-Distributed Clustered VLIW Architecture, in Proceedings of the 33rd Annual International Symposium on Microarchitecture, Monterey, CA, December 2000, pp. 124–133.

[130] R. Sethi and J.D. Ullman, The generation of optimal code for arithmetic expressions, *J. ACM*, 17(4), 715–728, October 1970.

[131] H. Sharangpani and H. Arora, Itanium processor microarchitecture, *IEEE Micro*, 20(5), 24–43, September 2000.

[132] D. Shin, S. Lee and J. Kim, Intra-task voltage scheduling for low-energy hard real-time applications, *IEEE Design Test Comput.*, March 2001.

[133] R. Silvera, J. Wang, G.R. Gao and R. Govindarajan, A Register Pressure Sensitive Instruction Scheduler for Dynamic Issue Processors, in *Proceedings of the 1997 International Conference on Parallel Architectures and Compilation Techniques, San Francisco, CA, November 1997*, pp. 78–89.

[134] M. Slater, AMD's K5 designed to outrun Pentium, *Microprocessor Rep.*, October 1994.

[135] J.E. Smith, G.E. Dermer, B.D. Vanderwarn, S.D. Klinger, C.M. Rozewski, D.L. Fowler, K.R. Scidmore and J.P. Laudon, The ZS-1 Central Processor, in Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems, Palo Alto, CA, October 1987, pp. 199–204.

[136] J.E. Smith and G.S. Sohi, The microarchitecture of superscalar processors, *Proc. IEEE*, 83(12), 1609–1624, December 1995.

[137] M. Smotherman, S. Krishnamurthy, P.S. Aravind and D. Hunnicutt, Efficient DAG Construction and Heuristic Calculation for Instruction Scheduling, in Proceedings of the 24th Annual International Symposium on Microarchitecture, Albuquerque, NM, November 1991, pp. 93–102.

[138] B. Su, S. Ding, J. Wang and J. Xia, Microcode Compaction with Timing Constraints, in *Proceedings of the 20th Annual Workshop on Microprogramming, Colorado Springs, CO, December 1987*, pp. 59–68.

[139] C.-L. Su, C.-Y. Tsui and A.M. Despain, Low Power Architecture and Compilation Techniques for High-Performance Processors, in *Proceedings of the IEEE COMPCON, 1994*, pp. 489–498.

[140] Texas Instruments, TMS320C62xx CPU and Instruction Set Reference Guide, 1998. *www.ti.com/sc/c6x*.

[141] M.D. Tiemann, The GNU Instruction Scheduler — cs343 Course Report, Technical report, Computer Science, Stanford University, Stanford, CA, 1989.

[142] A. Timmer, M. Strik, J. van Meerbergen and J. Jess, Conflict Modelling and Instruction Scheduling in Code Generation for in-House DSP cores, in Proceedings of the 32nd ACM/IEEE Design Automation Conference, June 1995, pp. 593–598.

[143] V. Tiwari, S. Malik and A. Wolfe, Power analysis of embedded software: a first step towards software power minimization, *IEEE Trans. Very Large Scale Integration Syst.*, 2, 437–445, 1994.

[144] M.C. Toburen, T.M. Conte and M. Reilly, Instruction Scheduling for Low Power Dissipation in High Performance Processors, in Proceedings of the Power Driven Microarchitecture Workshop, June 1998, in conjunction with the 25th International Symposium on Computer Architecture (ISCA'98), Barcelona, Spain.

[145] M. Tokoro, T. Takizuka, E. Tamura and I. Yamamura, A Technique for Global Optimization of Microprograms, in Proceedings of the Tenth Annual Microprogramming Workshop, Niagara Falls, NY, October 1977, pp. 41–50.

[146] R. Venugopal and Y.N. Srikant, Scheduling expression trees with reusable registers on delayed-load architectures, *Comput. Languages*, 21(1), 49–65, 1995.

[147] M.G. Valluri and R. Govindarajan, Modulo-Variable Expansion Sensitive Software Pipelining, in Proceedings of the 5th International Conference on High Performance Computing, Chennai, India, 1998, pp. 334–341.

[148] M.G. Valluri and R. Govindarajan, Evaluating Register Allocation and Instruction Scheduling Techniques in out-of-order Issue Processors, in Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques, Newport Beach, CA, October 1999.

[149] M.G. Valluri, Evaluation of Register Allocation and Instruction Scheduling Methods in Multiple Issue Processors, M.Sc.(Eng.) thesis, Supercomputer Education and Research Centre, Indian Institute of Science, Bangalore, India, January 1999.

[150] J. Wang and C. Eisenbeis, Decomposed software pipelining: a new approach to exploit instruction level parallelism for loop programs, in *Proceedings of the IFIP WG 10.3 Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, A-23 in IFIP Transactions, M. Cosnard, K. Ebcioğlu and J.-L. Gaudiot, Eds., North-Holland, Amsterdam, 1993, pp. 3–14.

[151] H.S. Warren, Jr., Instruction scheduling for the IBM RISC System/6000 processor, *IBM J. Res. Dev.*, 34(1), 85–92, January 1990.

[152] N.J. Warter, G.E. Haab, J.W. Bockhaus and K. Subramanian, Enhanced Modulo Scheduling for Loops with Conditional Branches, in Proceedings of the 25th Annual International Symposium on Microarchitecture, Portland, OR, December 1992, pp. 170–179.

[153] N.J. Warter, S.A. Mahlke, W.W. Hwu and B.R. Rau, Reverse If-Conversion, in Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation, Albuquerque, NM, June 1993, pp. 290–299.

[154] N.J. Warter-Perez and N. Partamian, Modulo Scheduling with Multiple Initiation Intervals, in Proceedings of the 28th Annual International Symposium on Microarchitecture, Ann Arbor, MI, November 1995, pp. 111–118.

[155] K. Wilken, J. Liu and M. Heffernan, Optimal Instruction Scheduling Using Integer Programming, in Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation, Vancouver, Canada, June 2000, pp. 121–133.

[156] K.C. Yeager, The MIPS R10000 superscalar microprocessor, *IEEE Micro*, 16(2), 28–40, April 1996.

[157] J. Zalamea, J. Llosa, E. Ayguadé and M. Valero, Improved Spill Code Generation for Software Pipelined Loops, in Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation, Vancouver, Canada, June 2000, pp. 134–144.

[158] C. Zhang, R. Govindarajan, S. Ryan and G.R. Gao, Efficient State-Diagram Construction Methods for Software Pipelining, CAPSL Technical Memo 28, Department of Electrical and Computer Engineering, University of Delaware, Newark, DE, March 1999; in *ftp.capsl.udel.edu/pub/doc/memos*.

18

Software Pipelining

Vicki H. Allan
Utah State University

Stephen J. Allan
Utah State University

18.1 Introduction

Software pipelining is an excellent method for improving the parallelism in loops even when other methods fail. Emerging architectures often have support for software pipelining [36, 41, 65].

Many approaches exist for improving the execution time of an application program. One approach involves improving the speed of the processor, whereas another, termed *parallel processing*, involves using multiple processing units. Often, both techniques are used. Parallel processing takes various forms, including processors that are physically distributed, processors that are physically close but asynchronous, synchronous multiple processors or multiple functional units. Fine grain, or instruction level, parallelism deals with the utilization of synchronous parallelism at the operation level.

Several methods are available for creating code to take advantage of the power of the parallel machines. From a theoretical perspective, forcing the user to redesign the algorithm is a superior

0-8493-1240-7/03/$0.00+$1.50

choice. However, there is always a need to take sequential code and parallelize it. Regular loops, such as **for** loops, lend themselves to parallelization techniques. Many techniques are available for parallelizing nested loops [79]. Techniques such as loop distribution, loop interchange, skewing, tiling, loop reversal and loop bumping are readily available [75, 76]. However, when the dependences of a loop do not permit vectorization or simultaneous execution of iterations, other techniques are required. Software pipelining restructures loops so that code from various iterations are overlapped in time. This type of optimization does not unleash massive amounts of parallelism, but creates modest amounts of parallelism.

18.2 Background Information

18.2.1 Modeling Resource Usage

Two operations conflict if they require the same resource. For example, if operations O_1 and O_2 each need the floating point adder (and there is only one floating point adder), the operations cannot execute simultaneously. Any condition that disallows the concurrent execution of two operations can be modeled as a conflict. This is a fairly simple view of resources. A more general view uses the following to categorize resource usage:

1. *Homogeneous or heterogeneous.* The resources are homogeneous if they are identical, and the operation does not need to specify which resource is needed. Otherwise, the resources are heterogeneous.
2. *Specific or general.* If resources are heterogeneous and duplicated, we say the resource request is specific if the operation requests a specific resource instead of any one of a given class. Otherwise, the resource request is general.
3. *Persistent or nonpersistent.* A resource request is persistent if one or more resources are required after the cycle in which an instruction is issued (the issue cycle). Otherwise, the request is nonpersistent.
4. *Regular or irregular.* We say a resource request is regular if it is persistent, but the resource use is such that only conflicts at the issue cycle need to be considered. In other words, two operations that begin at different instruction cycles cannot conflict, so no history of operations previously scheduled is required. Otherwise, the request is irregular.

A common model of resource usage (heterogeneous, specific, persistent, regular) indicates which resources are required by each operation over the duration of the operation. The resource reservation table proposed by some researchers models persistent, irregular resources [19, 70]. A reservation table is illustrated in Figure 18.1 in which the needed resources for a given operation are modeled as a table in which the rows represent time (relative to instruction issue) and the columns represent resources (adapted from [55]). For this reservation table, a series of multiplies or adds can proceed one after another, but an add cannot follow a multiply by two cycles because the result bus cannot be shared. This low-level model of resource usage is extremely versatile in modeling a variety of machine conflicts.

18.2.2 Data Dependence Graph

In scheduling, it is important to know which operations must follow other operations. We say a conflict exists if two operations cannot execute simultaneously, but it does not matter which one executes first. A dependence exists between two operations if interchanging their order changes the results. Dependences between operations constrain what can be done in parallel. A data dependence graph (DDG) is used to illustrate the must follow relationships between various operations. Let the DDG be represented by $DDG(N, A)$, where N is the set of all nodes (operations) and A is the set of

			ALU		Multiplier				
Time	Source 1	Source 2	Stage 0	Stage 1	Stage 0	Stage 1	Stage 2	Stage 3	Result Bus
0	X	X							
1			X						
2				X					
3									X

(a)

			ALU		Multiplier				
Time	Source 1	Source 2	Stage 0	Stage 1	Stage 0	Stage 1	Stage 2	Stage 3	Result Bus
0	X	X							
1					X				
2						X			
3							X		
4								X	
5									X

(b)

FIGURE 18.1 Possible reservation tables for (a) pipelined add and (b) pipelined multiply.

all arcs (dependences). Each directed arc represents a must follow relationship between the incident nodes. The DDG for software-pipelining algorithms contain true dependences, antidependences and output dependences. Let O_1 and O_2 be operations such that O_1 precedes O_2 in the original code. O_2 must follow O_1 if any of the following conditions hold: (1) O_2 is data dependent on O_1 if O_2 reads data written by O_1; (2) O_2 is antidependent on O_1 if O_2 destroys data required by O_1 [12]; or (3) O_2 is output dependent on O_1 if O_2 writes to the same variable as does O_1. The term *dependence* refers to data dependences, antidependences and output dependences.

There is another reason one operation must wait for another operation. A control dependence exists between a and b if the execution of statement a determines whether statement b is executed [79]. Thus, although b is able to execute because all the data are available, it may not execute because it is not known whether it is needed. A statement that executes when it is not supposed to execute could change information used in future computations. Because control dependences have some similarity with data dependences, they are often modeled in the same way [24].

When several copies of an operation exist (representing the operation in various iterations), several modeling choices present themselves. We can let a different node represent each copy of an operation, or we can let one node represent all copies of an operation. We use the latter method. As suspected, this convention makes the graph more complicated to read and requires the arcs be annotated.

Dependence arcs are categorized as follows. A loop-independent arc represents a must follow relationship among operations of the same iteration. A loop-carried arc shows relationships between the operations of different iterations. Loop-carried dependences may turn traditional DDGs into cyclic graphs [79]. (Obviously, dependence graphs are not cyclic when operations from each iteration are represented distinctly. Cycles are caused by the representation of operations.)

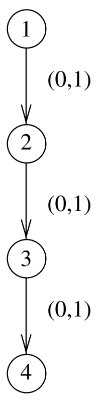
18.2.3 Generating a Schedule

Consider the loop body of Figure 18.2(a).[1] Although each operation of an iteration depends on the previous operation, as shown by the DDG of Figure 18.2(b), no dependence exists between the various

[1]We use pseudo code to represent the operations. Even though array accessing is not normally available at the machine operation level, we use this high-level machine code to represent the dependences because it is more readable than reduced instruction set computer (RISC) code or other appropriate choices. We are not implying our target machine has such operations.

```
for i
    O1:  a[i] = a[i] + 1
    O2:  b[i] = a[i] / 2
    O3:  c[i] = b[i] + 3
    O4:  d[i] = c[i]
end for
```

(a)



(b)

```
                 ITERATIONS

 T    I1     1    1    1 . . .
 I    I2     2    2    2 . . .
 M    13     3    3    3 . . .
 E    I4     4    4    4 . . .
```

(c)

FIGURE 18.2 (a) Loop body pseudo code; (b) data dependence graph in which arcs are labeled with dependence information (dif, min); (c) schedule.

iterations; the dependences are loop independent. This type of loop is termed a *doall* loop because each iteration can be given an appropriate loop control value and may proceed in parallel [18, 42]. The assignment of operations to a particular time slot is termed a *schedule*. A schedule is a rectangular matrix of sets of operations in which the rows represent time and the columns represent iterations. Figure 18.2(c) indicates a possible schedule. A set of operations that executes concurrently is termed an *instruction*. All copies of operation 1 from all iterations execute together in the first instruction. Similarly, all copies of operation 2 execute together in the second instruction. Although it is not required that all iterations proceed in lockstep, it is possible if sufficient functional units are present.

Doall loops often represent massive parallelism and hence are relatively easy to schedule. A *doacross* loop is one in which some synchronization is necessary between operations of various iterations [75]. The loop of Figure 18.3(a) is an example of such a loop. Operation O_1 of one iteration must precede O_1 of the next iteration because the $a[i]$ used is computed in the previous iteration. Although a doall loop is not possible, some parallelism can be achieved between operations of various iterations. A doacross loop allows parallelism between operations of various loops when proper synchronization is provided.

The idea behind software pipelining is that the body of a loop can be reformed so that one iteration of the loop can start before previous iterations finish executing, potentially unveiling more parallelism. Numerous systems completely unroll the body of the loop before scheduling to take advantage of parallelism between iterations. Software pipelining achieves an effect similar to unlimited loop unrolling without as much of a code space penalty.

Because adjacent iterations are overlapped in time, dependences between operations of different iterations must be identified. To see the effect of the dependences in Figure 18.3(a), it is often helpful (for the human reader) to unroll a few iterations as in Figure 18.3(b). Figure 18.3(c) shows the DDG of the loop body. In this example, all dependences are true dependences. The arcs $1 \rightarrow 2$, $2 \rightarrow 3$ and $3 \rightarrow 4$ are loop independent whereas the arc $1 \rightarrow 1$ is a loop-carried dependence. The difference (in iteration number) between the source operation and the target operation is denoted as the first value of the pair associated with each arc, and is termed the *dif*.[2] Figure 18.4 shows a similar example in

[2]The second value of the pair is termed *min* and is to be explained shortly.

```
for (i=1;i<=n;i++)

 O1: a[i + 1] = a[i] + 1
 O2: b[i] = a[i + 1] / 2
 O3: c[i] = b[i] + 3
 O4: d[i]=c[i]
```

(a)

```
O1: a[2] = a[1] + 1
O2: b[1] = a[2] / 2
O3: c[1] = b[1] + 3
O4: d[1]=c[1]

O1: a[3] = a[2] + 1
O2: b[2] = a[3] / 2
O3: c[2] = b[2] + 3
O4: d[2]=c[2]

O1: a[4] = a[3] + 1
O2: b[3] = a[4] / 2
O3: c[3] = b[3] + 3
O4: d[3]=c[3]
```

(b)

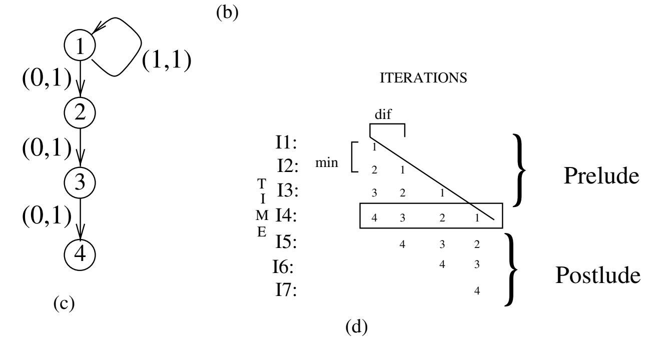
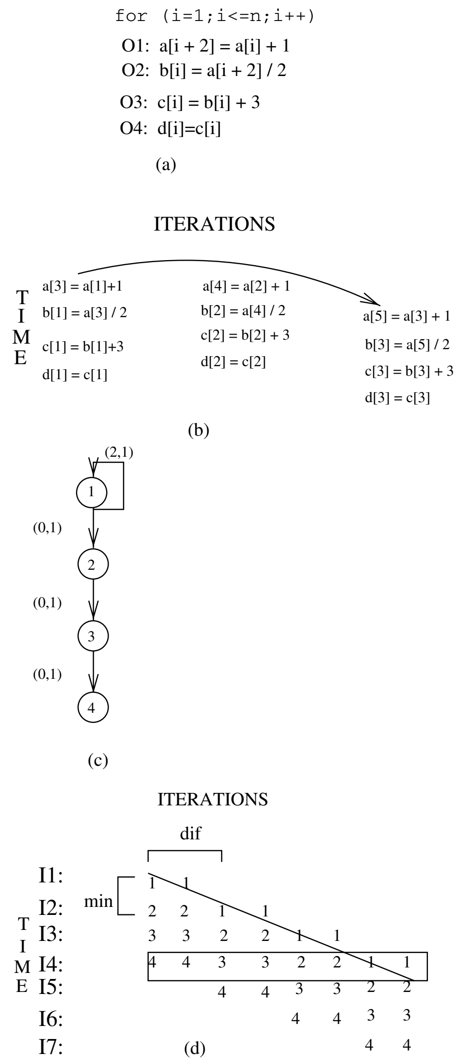


FIGURE 18.3 (a) Loop body pseudo code; (b) first three iterations of unrolled loop; (c) DDG; (d) execution schedule of iterations. Time (*min*) is vertical displacement. Iteration (*dif*) is horizontal displacement. In this example, *min* $= 1$ and *dif* $= 1$. The slope (*min/dif*) of the schedule is then 1. The new loop body is shown in the rectangle.

which the loop-carried dependence is between iterations that are two apart. With this less restrictive constraint, the iterations can be overlapped more.

It is common to associate a delay with an arc, indicating that a specified number of cycles must elapse between the initiation of the incident operations. Such delay is used to specify that some operations are multicycle, such as a floating point multiply. An arc $a \rightarrow b$ is annotated with a *min* time that is the time that must elapse between the time the first operation is executed and the time the second operation is executed. Because each node represents the operation from all iterations, a dependence from node a in the first iteration to b in the third iteration must be distinguished from a dependence between a and b of the same iteration. Thus, in addition to annotation with *min* time, each dependence is annotated with the *dif* that is the difference in the iterations from which the operations come. To characterize the dependence, a dependence arc, $a \rightarrow b$, is annotated with a (*dif*, *min*) dependence pair. The *dif* value indicates the number of iterations the dependence spans, termed the *iteration difference*. If we use the convention that a^m is the version of a from iteration m, then ($a \rightarrow b$, *dif*, *min*) indicates a dependence exists between a^m and b^{m+dif}, $\forall m$. For loop independent arcs, *dif* is zero. The minimum delay intuitively represents the number of instructions that an operation takes to complete. More precisely, for a given value of *min*, if a^m is placed in instruction t (denoted I_t) then b^{m+dif} can be placed no earlier than I_{t+min}.



```
for (i=1;i<=n;i++)
O1: a[i + 2] = a[i] + 1
O2: b[i] = a[i + 2] / 2
O3: c[i] = b[i] + 3
O4: d[i]=c[i]
```

FIGURE 18.4 (a) Loop body code; (b) first three iterations of unrolled loop; (c) DDG; (d) execution schedule of iterations.

Table 18.1 shows examples of code that contain loop carried dependences. For the code shown, *a* precedes *b* in the loop. The loop control variable is *i*, *y* is a variable, *m* is an array and *x* is any expression. For example, the first row indicates that $m[i]$ is assigned a value two iterations before it is used. Thus, it has a true dependence with a *dif* of 2.

If each iteration of the loop in Figure 18.3(a) is scheduled without overlap, four instructions are required for each iteration because no two operations can be done in parallel (due to the dependences). However, if we consider operations from several iterations, there is a dramatic improvement.[3] In

[3]Scheduling in a parallel environment is sometimes called *compaction* because the schedule produced is shorter than the sequential version.

TABLE 18.1 Dependence Examples

Instruction Label	Instruction	DDG Arc	Arc Type	*Dif*
a	$m[i+2]=x$			
b	$y=m[i]$	$a \to b$	True	2
a	$y=m[i+3]$			
b	$m[i]=x$	$a \to b$	Anti	3
a	$m[i]=x]$			
b	$y=m[i-2]$	$a \to b$	True	2
a	$y=m[i]$			
b	$m[i-3]=x$	$a \to b$	Anti	3
a	$y=t$	$a \to b$	Anti	0
b	$t=x+i$	$b \to a$	True	1
a	$t=x+i$	$a \to b$	True	0
b	$y=t$	$b \to a$	Anti	1
a	$y=x+i$			
b	$y=t$	$a \to b$	Output	0

Figure 18.3(d), we assume four operations can execute concurrently, allowing all four operations (from four different iterations) to execute concurrently in I_4.

One seeks to minimize the code needed to represent the improved schedule by locating a repeating pattern in the newly formed schedule. The instructions of a repeating pattern are called the *kernel*, $\mathcal{K}$, of the pipeline. In this chapter, we indicate a kernel by enclosing the operations in a box as shown in Figure 18.3(d). The kernel is the loop body of the new loop. Because the work of one iteration is divided into chunks and executed in parallel with the work from other iterations, it is termed a *pipeline*.

Numerous complications can arise in pipelining. In Figure 18.5(a), O_3 and O_4 from the same iteration can be executed together as indicated by 3, 4 in the schedule of Figure 18.5(c). Note, that no loop carried dependence occurs between the various copies of O_1. When scheduling operations from successive iterations as early as dependences allow (termed *greedy* scheduling), as shown in Figure 18.5(c), O_1 is always scheduled in the first time step I_1. Thus, the distance between O_1 and the rest of the operations increases in successive iterations. A cyclic pattern (such as those achievable in other examples) never forms. In the example of Figure 18.6(b),[4] a pattern does emerge and is shown in the box. Notice, it contains two copies of every operation. The double-sized loop body is not a serious problem, but does increase code size. Note that delaying the execution of every operation to once every second cycle would eliminate this problem without decreasing throughput as shown in Figure 18.6(d).

In Figure 18.6(c), a random (nondeterministic) scheduling algorithm prohibits a pattern from forming quickly. Care is required when choosing a scheduling algorithm for use with software pipelining.

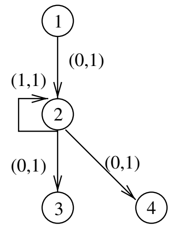
18.2.4 Initiation Interval

In Figure 18.3(d), a schedule is achieved in which an iteration of the new loop is started in every instruction. The delay between the initiation of iterations of the new loop is called the *initiation*

[4]It is assumed that a maximum of three operations can be performed simultaneously. General resource constraints are possible, but we assume homogeneous functional units in this example for simplicity of presentation.

```
for i
    O1: a[i] = i*i
    O2: b[i] =  a[i]*b[i-1]
    O3:c[i] = b[i]/n
    O4:d[i] = b[i]%n
```

(a)



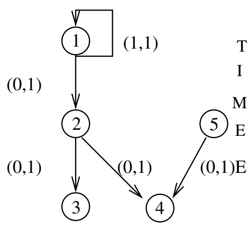
(b)

ITERATIONS (TIME ↓)

TIME			
I1:	1	1	1
I2:	2		
I3:	3,4	2	
I4:		3,4	2
I5:			3,4

(c)

FIGURE 18.5 (a) Loop body code; (b) DDG; (c) schedule.



(a)

ITERATIONS (TIME ↓)

TIME				
I1:	1			
I2:	2	1		
I3:	3,5	2		
I4:	4	3,5		
I5:		4	1	
I6:			2	1
I7:			3,5	2
I8:			4	3,5

(b)

ITERATIONS

I1:	1				
I2:	2	1			
I3:	3,5	5			
I4:	4	2	1		
I5:		3,4	2		
I6:			5	1,5	
I7:			3,4	2	
I8:				3,4	1
I9:					2,5

(c)

ITERATIONS (TIME ↓)

TIME		
I1:	1	
I2:	2	
I3:	3,5	1
I4:	4	2
I5:		3,5
I6:		4

(d)

FIGURE 18.6 (a) DDG; (b) schedule that forms a pattern; (c) schedule that does not form a pattern; (d) shorter schedule achieved by delaying execution.

interval (II) and has length $\mid \mathcal{K} \mid$. This delay is also the slope of the schedule that is defined to be *min/dif*, where *min* and *dif* are the labels on the arcs that control the II. The new loop body must contain all the operations in the original loop. When the new loop body is shortened, execution time is improved. This corresponds to minimizing the effective initiation interval that is the average time one iteration takes to complete. The effective initiation interval is $(\frac{II}{iteration\_ct})$, where *iteration_ct* is the number of copies of each operation in the loop on length $\mid \mathcal{K} \mid$.

Figure 18.4(d) shows a schedule in which two iterations can be executed in every time cycle (assuming functional units are available to support the 8 operations). The slope of this schedule is *min/dif* $= \frac{1}{2}$. Notice that this slope indicates how many time cycles it takes to perform an iteration (on average).

Because $\mathcal{K}$ does not start or finish in exactly the same manner as the original loop L, instruction sequences α (for *prelude*) and Ω (for *postlude*) are required to fill and empty the pipeline, respectively. Prelude and postlude are sometimes referred to as *prologue* and *epilogue*. If the earliest iteration represented in the new loop body is iteration c, and the last iteration represented in the new loop body is iteration d, the span of the pipeline is $d - c + 1$. If $\mathcal{K}$ spans n iterations, the prelude must start

$n - 1$ iterations preparing for the pipeline to execute, and the postlude must finish $n - 1$ iterations from the point where the pipeline terminates. Thus, $L^k = \alpha\ \mathcal{K}^m \Omega$ where k is the number of times L is executed, m is the number of times $\mathcal{K}$ is executed ($m = (k - n + 1)/iteration\_ct$, for $k \geq n$) and α and Ω together execute $n - 1$ copies of each operation.

18.2.5 Factors Affecting the Initiation Interval

18.2.5.1 Resource Constrained *II*

Some methods of software pipelining require an estimate of the initiation interval. The initiation interval is determined by both data dependences and the contention for resources that exists between operations. The resource usage imposes a lower bound on the initiation interval (II_{res}). For each resource, we compute the schedule length necessary to accommodate uses of that resource without regard to dependences or other resource usage; a lower bound on *II* due to resources. For the example of Figure 18.7(a), O_1 requires resource 1 at cycle 1 (from the time of issue) and resource 3 at cycle 3. If we count all resource requirements for all nodes, it is clear that resource 1 is required 3 times, resource 2 is required 4 times, and resource 3 is required 4 times. Thus, at least four cycles are required for a kernel containing all nodes. The relative scheduling of each operation of the original iteration is termed a *flat schedule*, denoted $\mathcal{F}$ and is shown in Figure 18.7(b). The schedule with a kernel size of 4 is shown in Figure 18.7(c).

18.2.5.2 Dependence Constrained *II*

Another factor that contributes to an estimate of the lower bound on the initiation interval is cyclic dependences. Several approaches exist for estimating the cycle length due to dependences, II_{dep}. We extend the concept of (*dif*, *min*) to a path. Let θ represent a cyclic path from a node to itself. Let min_θ be the sum of the *min* times on the arcs that constitute the cycle and let dif_θ be the sum of the *dif* times on the constituent arcs. In Figure 18.8, we see that the time between the execution of a node and itself (over three iterations, in this case) depends on *II*. In general, the time that elapses between the execution of a and another copy of a that is dif_θ iterations away is $II{\cdot}dif_\theta$. *II* must be large enough so that $II{\cdot}dif_\theta \geq min_\theta$. Because each iteration is offset dif_θ, after *II* iterations, $II{\cdot}dif_\theta$ time steps have passed. Arcs in the DDG follow the transitive law; therefore, a path θ containing a series

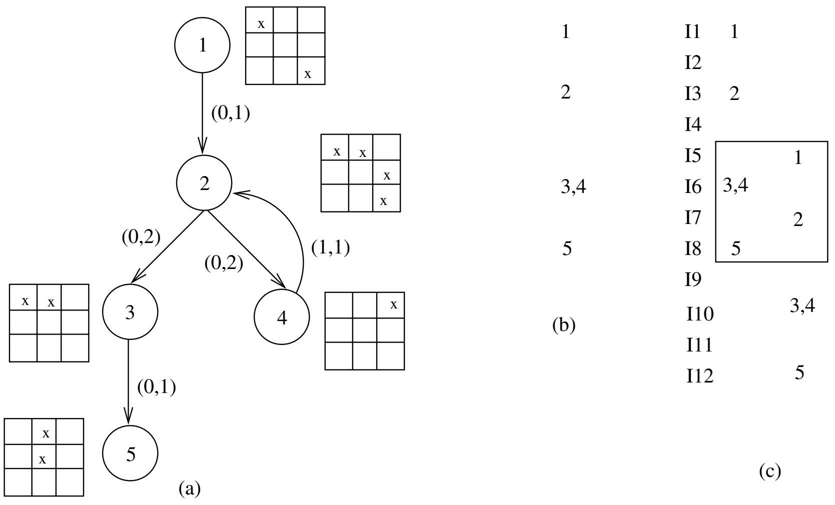


FIGURE 18.7 (a) DDG with reservation style resource constraints denoted by boxes; (b) flat schedule; (c) final schedule, stretched because of resource constraints.

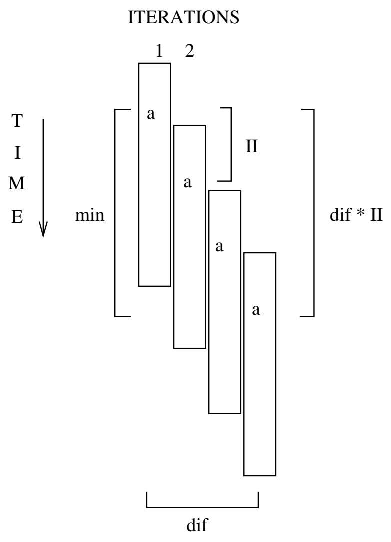


FIGURE 18.8 The effect of *II* on cyclic times.

of arcs with a sum of the minimum delays (min_θ) and a sum of the iteration differences (dif_θ) is functionally equivalent to a single arc from the source node of the path to the destination node with a dependence pair (dif_θ, min_θ). Becuase the cyclic dependences must also be satisfied, the transitive arc $a \to a$, representing a cycle, must satisfy the dependence constraint inequality (below) where the function $\sigma(x)$ returns the sequence number of the instruction in which the operation sequence x begins in $\mathcal{F}$. Let $Time(x^i)$ represent the actual time in which operation x from iteration i is executed. Then, the following formula results:

$$\forall\ cycles\ \theta,\ \ Time(a^{i+dif_\theta}) - \ Time(a^i) \geq \ min_\theta$$

In other words, the time difference in which cyclically dependent operations are scheduled must not be less than the min time. Let $i = 1$. Because $Time(a^1) = \sigma(a)$ and $Time(a^{1+dif_\theta}) = \sigma(a)+$ $II{\cdot}dif_\theta$, this formula becomes:

$$\forall\ cycles\ \theta,\ \ \sigma(a) + II \cdot dif_\theta - \sigma(a) \geq min_\theta$$

This can be rewritten as:

$$\forall\ cycles\ \theta,\ \ 0 \geq min_\theta - II \cdot dif_\theta$$

The lower bound on the initiation interval due to dependence constraints (II_{dep}) can be found by solving for the minimum value of *II*.

$$\forall\ cycles\ \theta,\ \ 0 \geq min_\theta - II_{dep} \cdot dif_\theta \tag{18.1}$$

$$II_{dep} = \max_{(\forall\ cycles\ \theta)} \left\lceil \frac{min_\theta}{dif_\theta} \right\rceil \tag{18.2}$$

For the example of Figure 18.7, $II_{dep} = 3$ because the only cycle has a *min* of 3 and a *dif* of 1. The actual lower bound on the initiation interval is then $II = \max(II_{dep}, II_{res})$, which is four for this example. Any cycle having *min* or *dif* equal to *II* is termed a *critical cycle*.

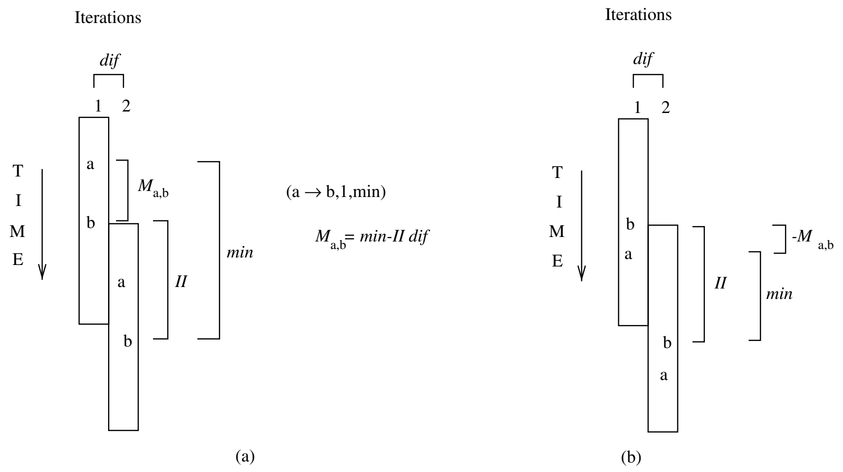


FIGURE 18.9 The effect of *II* on minimum times between nodes. Each rectangle represents the schedule of one iteration of the original loop. (a) A positive value for $M_{a,b}$ indicates *a precedes b*; (b) a negative value for $M_{a,b}$ indicates *a follows b*.

18.2.6 Methods of Computing *II*

18.2.6.1 Enumeration of Cycles

One method of estimating II_{dep} simply enumerates all the simple cycles [47, 68]. The maximum $\frac{min}{dif}$ for all cycles is then the II_{dep} [20].

18.2.6.2 Iterative Shortest Path

The method for computing II_{dep} can be simplified if one is willing to recompute the transitive closure for each possible *II*. For a given *II*, it is clear which of two (*dif*, *min*) pairs is more restrictive. Thus, the processing is much simpler as the cost table needs to contain only *one* (*dif*, *min*) pair [37, 77]. Because the cost of computing transitive closure grows as the square of the number of values at a cost entry, this is a sizable savings.

Path algebra is an attempt to formulate the software pipelining problem in rigorous mathematical terms [77]. Zaky constructs a matrix M that indicates for each entry $M_{i,j}$ the *min* time between the nodes i and j. This construction is simple in the event that the *dif* value between two nodes is zero. Let all nodes be labeled n_i. Assume $a = n_i$ and $b = n_j$. If there is an arc $(a \rightarrow b, 0, min)$, $M_{i,j} = min$. As is shown in Figure 18.9, we see that an arc $(a \rightarrow b, 1, min)$ implies that b must follow a by $min - II$ time units. In general, an arc $(n_i \rightarrow n_j, dif, min)$ represents the distance $min - dif \cdot II$. This computation gives the earliest time n_j can be placed with respect to n_i in the flat schedule. The drawback is that before we are able to construct this matrix, we must estimate *II*. The technique allows us to tell if the estimate for *II* is large enough and iteratively try larger *II* until an appropriate *II* is found.

Consider the graph of Figure 18.10. For an estimate of $II = 2$, the matrix M is shown in Figure 18.11(a). According to this matrix (for restrictions due to paths of length one), n_3 and n_4 can execute together ($M_{3,4} = 0$). Even though there must be a *min* time of 2 between n_3 of one iteration and n_4 from the next, as given by $(n_3 \rightarrow n_4, 1, 2)$, the delay between iterations (*II*) is two. Hence, no farther distance between n_3 and n_4 is required in the flat schedule.

Zaky defines a type of matrix multiply operation, termed *path composition*, such that $M^2 = M \otimes M$ represents the minimum time difference between nodes that is required to satisfy paths of length two. For two vectors (a_1, a_2, a_3, a_4) and (b_1, b_2, b_3, b_4), $(a_1, a_2, a_3, a_4) \otimes (b_1, b_2, b_3, b_4) = a_1 \otimes b_1 \oplus a_2 \otimes b_2 \oplus a_3 \otimes b_3 \oplus a_4 \otimes b_4$. Notice this is similar to an inner product. Symbol $\otimes$ has precedence

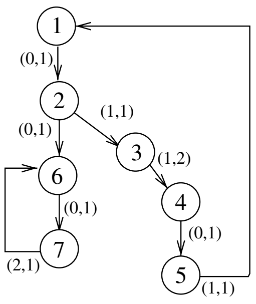


FIGURE 18.10 Sample graph.

	1	2	3	4	5	6	7
1	$-\infty$	1	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
2	$-\infty$	$-\infty$	-1	$-\infty$	$-\infty$	1	$-\infty$
3	$-\infty$	$-\infty$	$-\infty$	0	$-\infty$	$-\infty$	$-\infty$
4	$-\infty$	$-\infty$	$-\infty$	$-\infty$	1	$-\infty$	$-\infty$
5	-1	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
6	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	1
7	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	-3	$-\infty$

(a) Original Matrix M

	1	2	3	4	5	6	7
1	$-\infty$	$-\infty$	0	$-\infty$	$-\infty$	2	$-\infty$
2	$-\infty$	$-\infty$	$-\infty$	-1	$-\infty$	$-\infty$	2
3	$-\infty$	$-\infty$	$-\infty$	$-\infty$	1	$-\infty$	$-\infty$
4	0	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
5	$-\infty$	0	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
6	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	-2	$-\infty$
7	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	-2

(b) M^2

	1	2	3	4	5	6	7
1	0	1	0	0	1	2	3
2	-1	0	-1	-1	0	1	2
3	0	1	0	0	1	2	3
4	0	1	0	0	1	2	3
5	-1	0	-1	-1	0	1	2
6	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	-2	1
7	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	-3	-2

(c) Closure

FIGURE 18.11 Closure computation, in which infinity is represented by ∞.

```
for (k = 0; k < nodect; k++)
    for (i = 0; i < nodect; i++)
        if ($M$[i][k] > $-\infty$)
            for (j = 0; j < nodect; j++)
            { $t$ = $M$[i][k] + $M$[k][j];
                if (t > $M$[i][j])
                    $M$[i][j] = t;
            }
```

FIGURE 18.12 A variant of Floyd's algorithm for path closure.

over $\oplus$. Symbol $\otimes$ is addition, and $\oplus$ is maximum.[5] For example, to get $M^2(1, 6)$ we compose row 1 of the matrix of Figure 18.11(b) with column 6 as follows:

$$\begin{aligned}&[-\infty, 1, -\infty, -\infty, -\infty, -\infty, -\infty] \otimes [-\infty, 1, -\infty, -\infty, -\infty, -\infty, -3]\\&\quad = max(-\infty, 2, -\infty, -\infty, -\infty, -\infty, -\infty) = 2\end{aligned}$$

Thus, a path composed of two edges (indicated by the superscript on M) is between n_1 and n_6 such that n_6 must follow n_1 by two time steps. We can verify this result by noting the path that requires n_6 to follow n_1 by two is the path $1 \to 2 \to 6$ that has a (*dif*, *min*) of (0, 2).

The matrix M is a representation of the graph in which all *dif* values have been converted to zero. Therefore, edges of the transitive closure are formed from adding the *min* times of the edges that compose the path. Path composition, as just defined, adds transitive closure edges. Edges of the transitive closure are added by summing both the *dif* and *min* values composing the path. Zaky does the same thing by simply adding the minimum values on each arc of the path. This is identical as *dif*s in Zaky's method are always zero. Because multiple paths can be between the same two nodes, we must store the maximum distance between the two nodes. Thus, the matrix M^2 is shown in Figure 18.11(b). Formally, we perform regular matrix multiplication, but replace the operations $(\cdot,+)$ with $(\otimes,\oplus)$, where $\otimes$ indicates the *min* times must be added, and $\oplus$ indicates the need to retain the largest time difference required.

Clearly, we need to consider constraints on placement dictated by paths of all lengths. Let the closure of M be $\Gamma(M) = M \oplus M^2 \oplus M^3 \oplus \cdots \oplus M^{n-1}$ where n is the number of nodes and M^i indicates i copies of M path multiplied together. Only paths of length $n - 1$ need to be considered, because paths that are composed of more arcs must contain cycles and give no additional information. We propose using a variant of Floyd's algorithm as shown in Figure 18.12 to make closure more efficient. $\Gamma(M)$ represents the maximum distance between each pair of nodes after considering paths of all lengths.

A legal *II* produces a closure matrix in which entries on the main diagonal are nonpositive. For this example, an *II* of 2 is clearly minimal because of II_{dep}. The closure matrix contains nonpositive entries on the diagonal, indicating an *II* of 2 is sufficient. If an *II* of 1 is used, the matrix of Figure 18.13(b) results. The positive values on the diagonal indicate *II* is too small.

Suppose we repeat the example with $II = 3$ as shown in Figure 18.14. All diagonals are negative in the closure table. For instance, O_1 must follow O_1 by at least -4 time units. In other words, O_1 can precede O_1 from the next iteration by 4 time units. Because all values along the diagonal are nonpositive, $II = 3$ is adequate. Methods that use an iterative technique to find an adequate *II* try various values for *II* in increasing order until an appropriate value is found.

[5]It may seem strange that $\otimes$ is addition, but the notation was chosen to show the similarity between path composition and inner product.

	1	2	3	4	5	6	7
1	$-\infty$	1	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
2	$-\infty$	$-\infty$	0	$-\infty$	$-\infty$	1	$-\infty$
3	$-\infty$	$-\infty$	$-\infty$	1	$-\infty$	$-\infty$	$-\infty$
4	$-\infty$	$-\infty$	$-\infty$	$-\infty$	1	$-\infty$	$-\infty$
5	0	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
6	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	1
7	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	-1	$-\infty$

(a)

	1	2	3	4	5	6	7
1	3	4	4	5	6	8	9
2	2	3	3	4	5	7	8
3	2	3	3	4	5	7	8
4	1	2	2	3	4	6	7
5	3	4	4	5	6	8	9
6	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	0	1
7	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	-1	0

(b)

FIGURE 18.13 (a) Original matrix; (b) closure for $II = 1$.

	1	2	3	4	5	6	7
1	$-\infty$	1	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
2	$-\infty$	$-\infty$	-2	$-\infty$	$-\infty$	1	$-\infty$
3	$-\infty$	$-\infty$	$-\infty$	0	$-\infty$	$-\infty$	$-\infty$
4	$-\infty$	$-\infty$	$-\infty$	$-\infty$	-1	$-\infty$	$-\infty$
5	-2	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
6	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	1
7	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	-5	$-\infty$

(a)

	1	2	3	4	5	6	7
1	-4	1	-1	-1	-2	2	3
2	-5	-4	-2	-2	-3	1	2
3	-3	-2	-4	0	-1	-1	0
4	-3	-2	-4	-4	-1	-1	0
5	-2	-1	-3	-3	-4	0	1
6	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	-4	1
7	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	-5	-4

(b)

FIGURE 18.14 (a) Original matrix; (b) closure for $II = 3$.

```
For i = 1 to 4      a        For i = 1 to 2
  a                 b          a
  b                 a’         b
                    b’         a’
                    a”         b’
                    b”
                    a”’
                    b”’

  (a)              (b)         (c)
```

FIGURE 18.15 (a) Loop code; (b) completely unrolled loop; (c) replicated loop.

18.2.6.3 Linear Programming

Yet another method for computing II_{dep} is to use linear programming to minimize II given the restrictions imposed by the (*dif*, *min*) pairs [31].

18.2.7 Unrolling and Replication

The term *unrolling* has been used by various researchers to mean different transformations. A loop is completely unrolled if all iterations are concatenated as in Figure 18.15(b) in which primes are added to indicate which iteration the operation is from. We use the term *replicated* when the body of the loop is copied a number of times and the loop count adjusted as in Figure 18.15c. We use the term unrolling to represent complete unrolling and replication to represent making copies of the loop body. All replicated copies must exist in the newly formed schedule.

Replication allows fractional initiation intervals by letting adjacent iterations be scheduled differently. Time optimality is possible because the new loop body can include more than one copy of each operation. This is an advantage that can be achieved by any technique by using simple replication, but is complicated by the fact that (1) any replication increases complexity and (2) it is not known how much replication is helpful. Unrolling is used to find a schedule (see Section 18.5) in many methods. Many iterations may be examined to find a naturally occurring loop, but it is not required that more than one copy of each operation be in the new loop body.

18.2.8 Support for Software Pipelining

Software-pipelining algorithms sometimes require that the loop limit be a runtime constant. Thus, the pipeline can be stopped before it starts to execute operations from a future iteration that should not be executed. *Speculative execution* refers to the execution of operations before it is clear that they should be executed. For example, consider the loop of Figure 18.16 that is controlled by **for (i = 0;d[i] < MAX; i++)**. Suppose that five iterations execute before d[i] is greater than MAX. The operations from the succeeding iterations *should not have been executed*. Because we are executing operations from several iterations, when the condition becomes false, we have executed several operations that would not have executed in the original loop. Because these operations change variables, there must be some facility for “backing out” of the computations. When software pipelining is applied to general loops, the parallelism is not impressive unless support exists for speculative execution. Such speculative execution is supported by various mechanisms, including variable renaming or delaying speculative stores until the loop condition has been evaluated.

```
for (i = 0; d[i]<MAX;i++)
    O1: a[i + 1] = a[i] + 1
    O2: b[i] = a[i + 1] / 2
    O3: c[i] = b[i] + 3
    O4: d[i] = c[i]
```

(a)

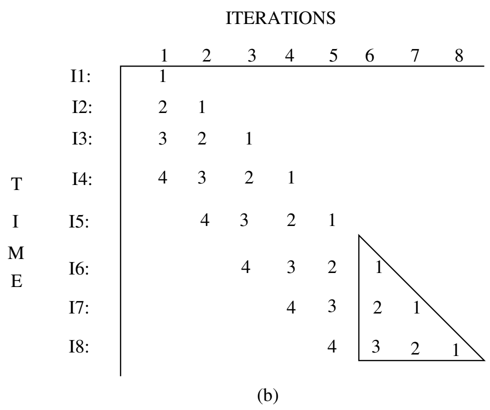


FIGURE 18.16 (a) Loop body code; (b) schedule (part enclosed in triangle should not have been executed).

18.3 Modulo Scheduling

Historically, early software pipelining attempts consisted of scheduling operations from several iterations together and looking for a pattern to develop. Modulo scheduling uses a different approach in that operation placement is done so that the schedule is legal in a cyclic interpretation [60, 61]. In other words, when operation a is placed at a given location, one must ensure that if the schedule is overlapped with other iterations, no resource conflicts or data dependence violations occur. In considering the software pipeline of Figure 18.17(b), a schedule for one iteration (shown in Figure 18.17(a)) is offset and repeated in successive iterations. If the schedule for one iteration is of length f, there are $\lceil f/II \rceil$ different iterations represented in the kernel (new loop body). Recall, this is termed the span. For the example of Figure 18.17(b), the span is 3 ($\lceil 6/2 \rceil$) because operations in the kernel come from three different iterations. The difficulty is in making sure the placement of operations is legal given that successive iterations are scheduled identically. In making that determination, it is clear that the offset (which is just the initiation interval) is known before scheduling begins. Because of the complications due to resource conflicts, we can only guess at an achievable initiation interval. Because the problem is a difficult one, no polynomial time algorithm exists for determining an optimal initiation interval; the problem has been shown to be NP complete [35, 43]. This problem is solved by estimating II (using II_{res} and II_{dep}) and then repeating the algorithm with increasing values for II until a solution is found.

Locations in the flat schedule (the relative schedule for the original iteration) are denoted F_1, $F_2, \ldots, F_f$. The pipelined loop, $\mathcal{K}$, is formed by overlapping copies of $\mathcal{F}$ that are offset by II. Figure 18.17(a) illustrates a flat schedule and Figure 18.17(b) shows successive iterations offset by the initiation interval to form a pipelined loop body of length two. This is termed modulo scheduling in that all operations from locations in the flat schedule that have the same value modulo II are executed simultaneously. This type of pipeline is called a *regular pipeline* in that each iteration of the loop is scheduled identically (i.e., $\mathcal{F}$ is created so that if a new iteration is started every II instructions, no resource conflicts exists and all dependences are satisfied).

Most scheduling algorithms use list scheduling in which some priority is used to select which of the ready operations is scheduled next. Scheduling is normally as early as possible in the schedule, though some algorithms have tried scheduling as late as possible or alternating between early and late placement [37] (which some term *bidirectional slack scheduling*). In modulo scheduling, operations are placed one at a time. Operations are prioritized by difficulty of placement (a function of the number of legal locations for an operation). Operations that are more difficult to place are scheduled

F1:	1
F2:	
F3:	2,3
F4:	4,5
F5:	6
F6:	7

(a)

I1:	1		
I2:			
I3:	2,3	1	
I4:	4,5		
I5:	6	2,3	1
I6:	7	4,5	
I7:		6	2,3
I8:		7	4,5
I9:			6
I10:			7

(b)

FIGURE 18.17 (a) Flat schedule $\mathcal{F}$ with $II = 2$; (b) The resulting regular pipeline.

first to increase the likelihood of success. Conceptually, when you place operation a into a partially filled flat schedule, you consider that the partial schedule is repeated at the specified offset. A legal location for a must not violate dependences between previously placed operations and a. In addition, resource conflicts must not be between operations that execute simultaneously in this schedule.

Consider the example of Figure 18.18 in which the dependence graph governing the placement is shown along with the schedule. Suppose operation 6 is the last operation to be placed. We determine a range of locations in the flat schedule in which 6 can be placed. Clearly, operation 6 cannot be placed earlier than F_5 (I_5 and I_9) because it must follow operation 4 that is located in F_4 (I_4 and I_8). However, this is also the latest it can be scheduled. When we consider iteration 2 (that is offset by the initiation interval of 4), operation 1 from iteration 2 (scheduled in I_5) must not precede operation 6 from iteration 1. Thus, only one legal location exists for operation 6 (assuming all other operations have been scheduled). All loop-carried dependences and conflicts between operations are considered as the schedule is built. The newly placed operation must be legal in the series of offset schedules represented by the previously placed operations. This is much different from other scheduling techniques. Other techniques schedule an operation from a particular iteration with previously scheduled operations from specific iterations. This technique schedules an operation from all iterations with previously scheduled operations from all iterations. In other words, one cannot schedule operation a from iteration 1 without scheduling operation a from all iterations.

Several different algorithms have been derived from the initial framework laid out by Rau and Glaeser [60] and Rau, Glaeser and Greenwalt [61].

18.3.1 Modulo Scheduling via Hierarchical Reduction

Several important improvements over the basic modulo-scheduling technique were proposed by Lam [44]. Her use of modulo variable expansion, in which one variable is expanded into one variable per overlapped iteration, has the same motivation as architectural support of rotating registers (which is discussed later). Rau originally included the idea as adapted to polycyclic architectures as part of the Cydra 5, but the ideas were not published until later due to proprietary considerations [14, 59]. The handling of predicates by taking the *or* (rather than the sum) of resource requirements (termed hierarchical reduction) of disjoint branches is a goal incorporated into state-of-the-art algorithms.

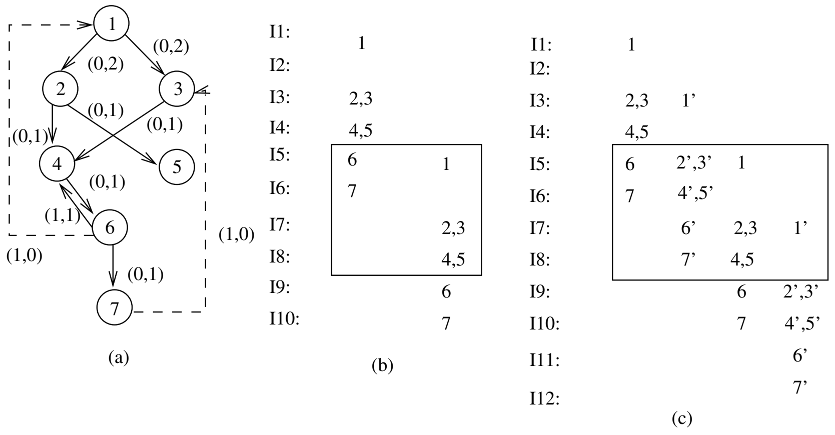


FIGURE 18.18 (a) DDG; (b) schedule; (c) schedule after renaming to eliminate loop-carried antidependence.

Hsu's stretch scheduling developed concurrently [34]. This algorithm is a variant of modulo scheduling in which strongly connected components[6] are scheduled separately [43, 44]. Although Lam uses a traditional list-scheduling algorithm, several modifications must be made to create the flat schedule.

Lam's model allows multiple operations to be present in a given node of the dependence graph. Because her method breaks the problem into smaller problems that are scheduled separately, she needs a way to store the schedule for a subproblem at a node. Each strongly connected component is reduced to a single node, representing its resulting schedule; this graph is termed a *condensation*. Because the condensed graph is acyclic, a standard list-scheduling algorithm is used to finish scheduling the loop body.

18.3.2 Path Algebra

Path algebra is an attempt to formulate the software-pipelining problem in rigorous mathematical terms [77]. In Section 18.2.6.2, path algebra was used to determine a viable *II* using the matrix M. This same matrix can also be used to determine a modulo schedule for software pipelining. Nodes that are on the critical cycle (having maximum min/dif) have a zero on the diagonal of $\Gamma(M)$ indicating the node must be exactly zero locations from itself.[7]

18.3.3 Predicated Modulo Scheduling

Predicated modulo scheduling has all the advantages of other techniques discussed in this section, but represents an improvement of known defects. It is an excellent technique that has been implemented in commercial compilers.

[6]A strongly connected component of a digraph is a set of nodes such that a directed path exists from every node in the set to every other node in the set. Strongly connected components can be found with Tarjan's algorithm [67].

[7]This is a little confusing in that it seems obvious that every node must be exactly zero locations from itself or *II* locations from itself in the next iteration. The point is that if dependence constraints force this distance, the techniques of path algebra can compute the required schedule.

Many researchers have embraced modulo scheduling for architectures with hardware support for modulo scheduling [20, 37, 46, 56–59, 69, 73] and have modified the resulting code to work on architectures without hardware support [74]. The Cydra 5 work is described in [20, 21]. We use the term *predicated modulo scheduling* to represent this general category of algorithms. In all but [37], the precise method for scheduling operations is not discussed, probably because of the complexity of explaining the process. One must assume the method used is similar to that employed by Lam except that the hierarchical reduction of schedules produced for strongly connected components (which generates suboptimal results) is circumvented.

18.3.3.1 Register Renaming

When iterations are overlapped, the reuse of registers becomes a concern. In the example of Figure 18.18(a) suppose operation 1 writes to a register (call it x) that is not used for the last time until operation 6 of the same iteration, and operation 3 writes to a register (call it y) that is not used for the last time until operation 7. In the DDG, these lifetimes manifest themselves not only in the dependence chain from 1 to 6 and from 3 to 7 but also in the antidependences of $7 \rightarrow 3$ and $6 \rightarrow 1$ (shown by dotted arcs in the graph). The antidependences have a (1, 0) annotation indicating they are loop carried ($dif > 0$) and the operation that writes to the register can be executed in the same instruction as the last use ($min = 0$). This is possible if we assume the fetch of a value precedes the write within a machine cycle. The dependence from operation 1 to 2 has a (0, 2) annotation indicating that the operation takes two cycles to complete ($min = 2$). The antidependences force the maximum min/dif to be 4. Thus, the schedule shown in Figure 18.18(b) is of length 4.

Let II_{anti} be the length of the longest cycle involved in a dependence cycle containing a loop-carried antidependence. Let II be the initiation interval for the schedule when antidependences are ignored. If we replicate the loop so that II_{anti}/II copies of the loop body exists, we can use different registers in each copy. This replicated loop is scheduled and shown in Figure 18.18(c). Operations that write to different registers are indicated with a prime (e.g., $1'$). In this case, because there are two copies of each operation, two versions exist for each register whose lifetime extends beyond II.

Table 18.2 shows the same schedule with renamed versions of a register differing by a prime. Writes of an operation are shown by the register name appearing to the left of an equal sign. Reads from a register are shown by the register name appearing to the right of an equal sign. For simplicity, only registers x and y are shown. Operation 1 writes to x in the first iteration and x' in the second iteration. In the third iteration, x is used again. Similarly operation 3 writes to y in the first and third iterations and writes to y' in the second iteration. Even though I_5 uses x and writes x, there is no problem as fetches precede stores within the cycle. Instead of two registers, four (x, x', y, y') are required and code space has increased, but the effective initiation interval is halved because two versions of each original operation are in the kernel. This is termed *modulo variable expansion* [44].

Hardware support for modulo scheduling simplifies register renaming. With the advent of rotating register files ([58, 59]), loop-carried antidependences can be ignored without code expansion. Variables that are not redefined in the loop or whose lifetime is less than II can be assigned static general purpose registers. Variables involved in loop-carried dependence cycles can take advantage of rotating register files. A rotating register file is a file whose current pointer rotates. Thus, register specifier n does not always refer to the same physical register, but rotates over the set of registers. This is accomplished by treating the register specifier as an offset from the iteration control pointer (ICP) that points to the beginning of the registers for the current iteration. Every register reference is computed as the sum of the register specifier and the ICP (modulo the register file size). The ICP is decremented (modulo register file size) at the end of each iteration execution. In this way, each iteration accesses different registers even though the code remains identical for each iteration. This provides hardware-managed renaming.

TABLE 18.2 Modulo Variable Expansion

Time	Iterations			
I_1	$1(x =)$			
I_2				
I_3	$2, 3(y =)$	$1'(x' =)$		
I_4	$4, 5$			
I_5	$6(= x)$	$2', 3'(y' =)$	$1(x =)$	
I_6	$7(= y)$	$4', 5'$		
I_7		$6'(= x')$	$2, 3(y =)$	$1'(x' =)$
I_8		$7'(= y')$	$4, 5$	
I_9			$6(= x)$	$2', 3'(y' =)$
I_{10}			$7(= y)$	$4', 5'$
I_{11}				$6'(= x')$
I_{12}				$7'(= y')$

By using a rotating register file for our example, the schedule of Figure 18.17(b), instead of Figure 18.18(c), is achieved. Operation 1 always writes the same register specifier (0 in this case), but because the registers rotate there is no problem. Similarly, operation 2 always writes to register specifier 1.[8]

18.3.3.2 Predicated Execution

When code contains conditionally executed code, modulo scheduling becomes more complicated. Consider the example of Figure 18.17. Suppose that operation 2 computes a predicate (Boolean value). If the predicate is true, operations 4 and 6 are executed. If the predicate is false, operations 5 and 7 are executed. Clearly, the schedule of Figure 18.17(b) is misleading because 4 and 5 are never both executed. We would need two versions of the code for each iteration of the replicated schedule. Because code from three iterations is overlapped, eight combinations could result in complicated code expansion. Let the notation (true, false, true) correspond to the values of the predicate in successive iterations that are true, then false, then true. Clearly, eight combinations of three Boolean values exists. One solution to this problem, hierarchical reduction, has already been mentioned [43]. Instead of scheduling each branch of a conditional separately, the code for both branches is scheduled with the understanding that only one of the branches is actually executed at runtime. In other words, the schedule is created so that it is legal regardless of which branch is taken; the needs of both branches are considered. The resource conflicts must be adjusted so that at any point in time, the union[9] of the resources required by different branches is available instead of requiring the sum of the resources to be available. The disadvantage is that even when we can use the union of the resource requirements, we are limited to the scheduling length of the longest path.

A hardware implementation of this idea involves the use of predicated execution. Predicated execution makes it possible to execute an operation conditionally. Instead of jumping around an operation that should not be executed, the hardware can just ignore the effects of the operation. For example, the floating point multiply specified by $r1 = \mathit{fmpy}(r2, r3)\ \langle p1 \rangle$ is executed only if predicate $p1$ is true. If $p1$ is false, either the operation is ignored (treated as a NO-OP) or is executed but the

[8]In general, register specifiers are not adjacent but are spaced to reflect the lifetime of the registers.

[9]The term *union* is used to indicate that two operations that cannot both execute (due to opposite values of their predicates) do not compete for resources.

result register is not changed. The former implementation saves effort, but the latter implementation allows the operation to be executed in the same instruction as the predicate is computed. Because the results of predicate evaluation are known before the target register is written, such overlap is possible. Such hardware support may eliminate a physical jump. It also makes it possible to modulo schedule loops containing conditionals without code expansion as well as reduce the code expansion caused by a distinct prelude and postlude.

For the same reasons registers are stored in a rotating file, predicates are also stored in a rotating file. Predicates can either share the regular rotating register file or have a dedicated predicate register file [58]. The process of converting code into predicate code is termed *if-conversion* and is an integral part of enhanced modulo scheduling (EMS) [73]. Conditional branches are removed and control dependences become data dependences because conditionally executed operations are data dependent on the operation that generates the predicate on which they depend. The proposed technique has more flexibility than hierarchical reduction in that hierarchical reduction completely schedules the conditional code before it attempts to schedule other operations. An arbitrary decision made in scheduling the branch can negatively impact the placement of other operations that are not even considered during this prescheduling phase.

In predicated execution schemes, all operations in an instruction are fetched and only those with true predicates complete execution. However, in EMS and hierarchical reduction, one is limited by the number of operations that physically execute for a given predicate value instead of the number of operations for all predicate values. In both cases, operations that execute under disjoint predicate values can be scheduled at the same time even if they require the same nonsharable resource. Proponents of this method are so enthusiastic they even recommend using this same technique for processors that do not support predicated execution. The proposed technique, called *reverse if-conversion*, simplifies the process of global scheduling [74]. EMS has an advantage over hierarchical reduction in that no prescheduling of paths in done. This increased flexibility produces superior code.

A side effect of predicated execution is that one avoids having special instructions for prelude and postlude. This is termed *kernel-only code* [57].

18.3.4 Enhanced Modulo Scheduling

Although all modulo-scheduling techniques are basically the same, they differ in how they handle predicates. Hierarchical reduction schedules operations on each branch of a conditional construct before combining, using the union of the requirements. This prescheduling and unioning of requirements creates complicated pseudo operations that are difficult to schedule efficiently with other operations. EMS uses if-conversion to convert all operations into straight-line, predicated code. In this form, scheduling is done, noting that disjoint operations do not conflict. There is no need to preschedule the various parts of the conditional construct. After modulo scheduling, modulo variable expansion is used to rename registers lifetimes from distinct iterations.

Predicated execution has the disadvantage that all operations from taken and untaken branches are scheduled for execution (even if the results are just thrown away). Thus, the resource requirements required is the sum of the requirements of each branch. EMS recreates the branching structure, reverse if-conversion, by inserting conditional branch instructions to eliminate predicated execution. In other words, predicated execution is used to allow operations to be scheduled independently of the branching structure and then the branching structure is reinserted. This method has some real benefits in terms of simplicity, but is also hampered by the fact that such a technique is prone to code explosion. If a predicated operation (predicated by p) happens to be scheduled early, all operations that are between the first predicated operation scheduled and the last operation predicated on p must be cloned to appear on each branch. If code that is predicated by p is overlapped n times, there can be code expansion of order 2^n. Clearly, this is unacceptable. Various techniques are employed to limit code explosion, the most important being the restriction of which blocks are scheduled together. The term *hyperblock* is used to denote which set of blocks are scheduled together [73, 74].

The initial simplicity of scheduling without regard to predicates results in the complexity of introducing conditionals back into the code. Because the insertion of branches is done after code scheduling, various code inefficiencies can be introduced as is common whenever phases are segmented. According to [73], EMS performs 18% better than hierarchical reduction with up to 105% increase in code size. Although some of the increase in code size undoubtedly results from the fact that tighter code overlaps more conditional constructs, some of the increase results from unnecessary elongation of the predicated region.

18.3.5 Other Techniques

Calland, Darte and Robert [17] revisit the technique of decomposed software pipelining [30, 72] in which the NP-complete problem of software pipelining is divided into two pieces to simplify the algorithms. One schedule considers cyclic scheduling ignoring resource requirements, whereas another schedule considers resources but ignores the cyclic requirements of the graph. Both approaches preprocess the graph by removing some of the data-dependence edges. Calland, Darte and Robert use a very simple resource model (nonpipelined and identical) but justify this approach because of the need to give an upper bound on the worst initiation interval that they may derive.

The basic approach is to create a schedule that is concerned with cyclic dependencies (but not resources) and then to modify it to deal with resources.

Govindarajan, Altman and Gao [33] combine the design of hardware pipeline schedules with software pipelining. In [32], they use a mathematical formulation of the problem using integer programming to compute a periodic schedule. They use a graph-coloring technique to form a schedule using minimal registers. This technique is time consuming and might only be used for time critical loops. It may also be used to evaluate other heuristic methods.

López et al. [45] describe a compilation technique that packs independent loads and stores of adjacent memory locations into a wide load and store. They indicate that doubling the width of the bus is as effective as doubling the number of buses, but is much cheaper. However, this option does require sophisticated compilation support. Although it was feared that wide buses would increase register pressure and would hence be undesirable, this work indicates that the negative effects are minimal.

18.3.6 Evaluation

Instead of waiting for a pattern to form, modulo techniques analyze the DDG and create a desirable schedule. The tight coupling of scheduling and pipelining constraints results in a pipeline that is near optimal. The ability to adjust the scheduling technique to control register pressure or prioritize by various attributes gives great flexibility. The regular pipeline that is produced simplifies the formation of the prelude, kernel and postlude. With the replication suggested, fractional rates can be achieved [39]. The trial-and-error approach of finding the achievable *II* increases the compile time, but no software pipelining algorithm has a tight bound on compile time.

Zaky uses a technique that produces results very similar to other modulo-scheduling techniques, but uses path algebra as a framework [77]. Zaky's algorithm is impractical because it cannot handle resource constraints, but is important because of the elegant way it formulates and solves the problem. Because of the concise algorithm, this technique provides an excellent conceptual model.

The modulo-scheduling techniques have continued to improve by using hardware support to reduce code expansion and allow tighter schedules by the use of rotating register files. These methods represent an excellent choice for software pipelining, with their only drawback the fact that fractional rates are not achieved without replication before scheduling.

The work of Ruttenberg et al. [62] presents a valuable comparison between heuristic modulo-scheduling techniques (implemented at Silicon Graphics, Inc. [SGI]) and the exhaustive integer linear programming (ILP) approach of Gao. Both SGI and ILP are variants of modulo scheduling but the former is heuristic whereas the latter is exhaustive. Although the ILP version is known to be NP complete (and thus too inefficient for practical consideration), it does serve as a means of evaluating other techniques.

Ruttenberg et al. use the term *rate optimal* to indicate a schedule with a minimal II and a valid register assignment. The heuristics suggested by SGI were shown to be highly effective. Software pipelining was improved by the following analysis and optimizations before software pipelining:

1. Array dependence analysis, loop interchange and outer loop unrolling
2. Common subexpression elimination, copy propagation, constant folding and strength reduction
3. If-conversion, common memory reference elimination and construction of the data dependence graph

Several improvements are used in the SGI variant of modulo scheduling. Instead of increasing the *II* by one each time a failure is encountered, a system of exponential backoff is used. In other words, the system successively tries II, $II+1$, $II+2$, $II+4$, $II+8$, etc. Once a successful schedule is found, it uses a binary search between the first success and the last failure to narrow down the correct II. This method assumes that if you can find a schedule at II, you can find a schedule at $II+1$. Although it can be shown that exceptions to this assumption exist, these researchers indicate that they never found a real case that violated this principle.

The SGI method uses regular branch-and-bound scheduling with severe pruning. When unscheduling, they do not unschedule operations with identical resource requirements that are not related by data dependencies. Similarly, the failure of operation X does not trigger the unscheduling of operations that do not overlap with X in resource requirements. Pruning of the search space is facilitated by limiting which operations can trigger unscheduling. An operation that is tried at the next possible location (after unscheduling all its successors in the scheduling list) is termed a *catch point*. Thus, pruning is accomplished by limiting what the catch points can be. Such restrictions include (1) only the first element in a strongly connected component can catch; (2) operation j may catch the backtrack caused by operation i failing to be scheduled, if i and j do not require identical operations and unscheduling i makes scheduling of j possible.

This research indicates that the ordering of operations for scheduling exerts a profound effect on the efficiency of the schedule produced, but which order was best varied from loop to loop. Instead of committing themselves to one ordering, they try four different orderings for each loop. The orderings used are variants of the following:

1. *Folded depth-first ordering*. This technique primarily uses the roots (stores of the calculations) as a beginning point to regular depth-first ordering; however, when some operations are very difficult to schedule, they can be treated as a root and scheduling proceeds outward from these new roots (hence the term, folded depth-first ordering).
2. *Order based on height*. The height in the DDG, in terms of the sum of all latencies on the path from the node in question to the root, is used to determine preference.
3. *Reversal of ordering*. Simply reverse the ordering generated by another technique.
4. *Delayed scheduling*. Schedule stores with no successors or loads with no predecessors at the end of the list (giving them lowest priority).

Ruttenberg et al. combined these ideas to come up with four orderings.

When the number of available registers is exceeded, one option is to reschedule with a longer initiation interval, but this solution normally results in less efficiency. A second solution is to move a register value to memory (and then restore it when needed, termed *spilling*) to save registers. The code generated due to spilling is termed *spill code*. Earlier research indicates that rescheduling after increasing the *II* tends to generate worse schedules, but this is not always the case.

Ruttenberg et al. [62] performed some valuable tests with spill candidates. Spills are added in an exponential manner (on failure): first spill one value, then two, then four and so on. They decide which value to spill based on the shortest legal schedule for which no register allocation could be done. For each live value, they compute the number of cycles for which the value is live divided by the number of times it is referenced in that range. The larger the references per cycle ratio are, the less the benefit of keeping the value in a register. Thus, values with the largest such ratio are picked for spilling.

ILP was adjusted slightly before the two methods were compared. The maximum running time of 3 minutes for ILP was used, but this did not appear to change the results much. Finding a register optimal schedule was too time consuming because of the involved mathematical specifications, so the goal was adjusted to finding a legal resource-constrained schedule. In addition, ILP was allowed to try several orders of instruction scheduling, just as SGI does.

Tests indicate that once a schedule has a valid register assignment, no real advantage results from reducing the number of registers used. The size of the prelude and postlude are particularly important for loops with a small loop count. In these results, they show that among loops having fewer registers there is not necessarily less pipelining overhead. Thus, as long as you can assign registers, minimizing the number of register used is not important to optimization.

In comparing the SGI heuristic method with ILP, SGI was obviously more efficient (because it is near linear time compared with exponential time of ILP). However, ILP generated a superior schedule to SGI in only one case, and that case could be eliminated with slight increases in allowed backtracking. ILP was not able to guarantee register optimality for many loops. With short loop counts and severe register pressure, ILP may have something to offer, but more tests would be needed.

18.4 Architectural Support for Software Pipelining

Software pipelining can be dramatically improved if architectures provide support for it. In [36], a description of the IA-64 architecture is given that can be used to provide a guide as to the ways architecture can support software pipelining. In addition to having 128 general-purpose registers, each with 65-bit (64-bit of data plus 1-bit for control), the IA-64 has 128 floating point registers each with 32-bit, space for 128 special-purpose registers each with 64-bit (for a register stack, for example), and 64 predicate registers each with 1-bit. The architecture also provides for a deferred exception flag (termed *not a thing* (NaT) bit) to aid with exceptions thrown during out-of-order execution.

Hardware in modern processors can perform branch prediction, determine instruction dependences, extract parallelism, modify the order of instructions, decide when to execute each (and even on which functional unit) and manage caches (controlling prefetching, instruction caching and data caching). However, such out-of-order processors still require significant software support to achieve the best speeds. In the case of the IA-64, the compiler provides instruction groups that can all be executed simultaneously. The instruction set of the IA-64 allows the expression of such parallelism [36]. Not only can independent instructions be performed in parallel, but the clauses of a compound condition can be executed simultaneously and their results combined in a single instruction.

Many processors use branch prediction to support speculative execution: operations that occur after the branch are moved before the branch and executed (but not committed) before the result of the branch is known. Once the results of the branch instruction are known, the results that are speculatively executed are either committed or discarded (depending on whether the prediction was correct). Obviously, if the prediction is incorrect, a heavy penalty is paid because all the speculative work is wasted.

The high degree of parallelism of the IA-64 allows operations from both branches to be executed simultaneously. The compiler is able to help by determining whether there are sufficient resources to execute instructions from both the true and the false branch.

Load instructions often have a longer latency than many other operations. If a load moves before a branch (to minimize the wasted time caused by the delay required to wait for the results), a problem can exist if the early load accesses data that the program does not yet have access to. Such an illegal action can cause an exception to be thrown, which is a serious problem for a speculatively executed operation because the executed version of the program must not have exceptions that would not have occurred in the original instruction ordering. The IA-64 prevents this problem by allowing a speculative load to set an NaT (deferred exception) bit in the target register. Then when the code tries to use that register (which was not successfully loaded), it branches to code to reload the value.

Similarly, the IA-64 allows a possible dependence to be ignored until it is actually determined that a dependence exists. When a load follows a store from the same location, the load must wait for the store to complete. However, with pointer arithmetic, it is difficult to reorder any loads and stores because they may possibly point to the same location — even though that aliasing is unlikely. Various memory disambiguation techniques are used to try to determine whether two locations could be the same, but many times it is impossible to determine.

With the IA-64, the loads are allowed to move before stores (via an advanced load instruction [ld.a]) and if it is discovered that the store wrote to the same location as the load, the load and its dependent operations are reexecuted. The advanced load instruction is made possible by using an advanced load address table (ALAT) to record speculatively executed instructions. The ALAT is a cache with content addressable memory. Whenever a store is executed, the hardware compares the address to all the ALAT entries. Any address that conflicts with a speculatively executed load causes the load to be removed from the ALAT table.

When an inquiry is made about the validity of an operation relying on an advanced load, the ALAT is searched. When no entry is present, the operation chain leading to the check must be recomputed via fix-up code. (The same action takes place if an item is removed from the ALAT because no room is available to store it because of too many subsequent advanced loads.) If the entry is found, the ld.a has succeeded and nothing needs to be done.

Note that this fix-up code is generated at compile time so no special hardware is required to support fix-up code. The ability to ignore a potential data dependence makes a huge difference in software pipelining. For example, consider the loop adapted from [41] that is illustrated in Figure 18.19. This example assumes a load delay of five cycles. The code using an advance load allows an *II* of 2 instead of 7.

The predicate register file allows for up to 64 predicates to be set and used to control the execution of statements, including branches. The rotating predicate file allows the engine to treat them as "isValidStage" bits, which allow the pipe to be filled and drained without special prelude and postlude code. Thus, little code expansion occurs with software pipelining. This sophisticated predication support is a huge benefit to software pipelining because small basic blocks are replaced by a large composite block that can be scheduled as a basic block.

The IA-64 uses a rotating register file to give the user the illusion of an infinite register stack. The first 32 registers are static, and the last 96 are used as a stack. Actually, hardware saves and restores on-chip registers to and from memory. Allocated registers are divided into local and output registers. At call time, the registers are renamed so that the output registers from the previous call become the first local registers (starting with r32). See [36] for more information. A similar register renaming can also be used for software pipelining via modification of the register rename base (rrb), thus eliminating the overhead of software register renaming [41].

Stotzer and Leiss [65] cite a example of software pipelining for the Texas Instrument TMS320C6X (hereafter referred to as C6X) for digital signal processing (DSP). DSP includes such areas as disk

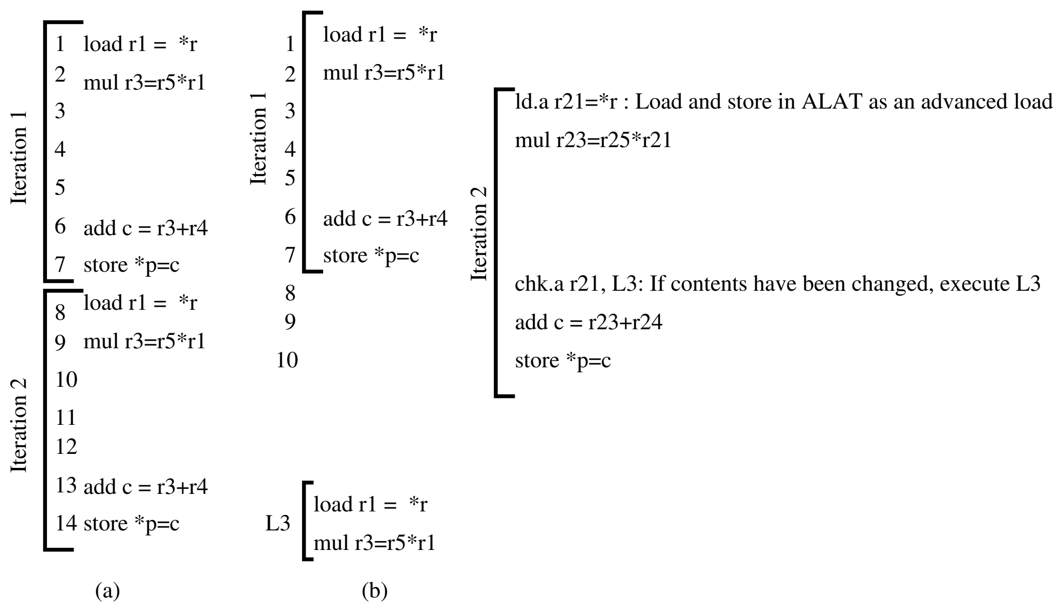


FIGURE 18.19 Speculative loads allow for a tighter loop. (a) Possible data dependence forces the load to execute after the store of the preceding loop. (b) Speculative executed load allows the load and multiply to be executed before the store as long as the nondependence of the load and store is checked before the actions are committed.

drive controllers, video conferencing, adaptive filtering and communication, as well as medical, sonar and equipment health monitoring. Speech, audio, image and video compression are also popular DSP applications.

The C6X processor is designed to be used in embedded systems, and hence has increased requirements both in terms of code size and register limitations. The register file is medium sized making reducing register pressure a priority. In addition, the code size is restricted, motivating the control of code expansion. Because changes to code are infrequent, more time can be taken to get an optimal schedule.

The C6X processor has a variety of features that make it an ideal candidate for software pipelining. The processor allows for eight operations to be performed simultaneously. Such power allows the processor to be used for embedded real-time applications that had previously required supercomputers to run.

There are two identical data paths in the C6X, each having 16 registers and four functional units. The four functional units (on each data path) are a multiplier, a logical adder, a shifter/logical and an address generator/adder. Each path can access a 32-bit data value from/to memory in each cycle. There is a limited ability to pass information between data paths.

The C6X supports predicated execution by allowing 5 of the 32 general registers to be used as registers whose value controls whether a given instruction executes. As discussed earlier, this capability allows for larger basic blocks because instructions from multiple blocks can be executed concurrently with proper predication.

Instructions on the C6X require various amounts of time to complete. For example, a multiply takes two cycles, a load takes five cycles, a branch takes six cycles and a shift takes one cycle. Once an instruction begins, it is considered to be in-flight until it finishes. To reduce the number of registers required, the C6X allows in-flight instructions to write to the same register as long as they actually do the write in different cycles. This is enormously beneficial in reducing register pressure. For

```
ldh *a5, a1
nop           ldh *a6, a0
nop           nop
nop           nop
nop           nop
add a1,a2,a2  nop
              add a0,a3,a3
```

(a) two intermediate registers

```
ldh *a5, a1  : a1=*a5
nop  }
nop  }
nop  } due to load latency
nop  }
add a1,a2,a2 : a2 +=a1
ldh *a6, a1  : a1=*a6
nop  }
nop  }
nop  } due to load latency
nop  }
add a1,a3,a3 : a3 +=a1
```

(b) one intermediate register

```
ldh *a5, a1
nop           ldh *a6, a1
nop           nop
nop           nop
nop           nop
add a1,a2,a2  nop
              add a1,a3,a3
```

(c) one register, multiple assignment

FIGURE 18.20 Multiple assignment reduces register pressure without increasing code length. (a) Using two intermediate registers (a0 and a1); (b) using one intermediate register (a1); (c) using one intermediate register (a1) but relying on the fact that it can be assigned in different cycles.

example, in Figure 18.20 we are attempting to perform the following: add the contents of the address stored in register a5 to a2 and add the contents of the address stored in register a6 to the contents of a3. Figure 18.20(a) shows the schedule resulting from the five-cycle load latency when we use two different intermediate registers to store the loaded values. Note that the two load operations are overlapped in time. Figure 18.20(b) shows the increased schedule length required if a single register is used as a temporary register, allowing no overlap of the load instructions. Figure 18.20(c) shows the same code only utilizing the processor feature of two in-flight instructions that are allowed to write to the same register. The example illustrates that the multiple assignment feature can yield results equivalent to the two register case while only requiring one intermediate register. In this example, a1 is loaded and used before the second load is completed.

In the work of Stotzer and Leiss [65], instructions are assigned, by hand, to one of the two data paths to simplify the scheduling. However, their approach does not assign a specific functional unit until scheduling. This gives needed flexibility in that functional units that are in less demand can be used. For example, because an add can be performed by the logical/shift, the logical/adder or the address generation/adder, the functional unit that competing operations do *not* need can be selected to avoid scheduling conflicts. An add is needed but a commitment as to how the add is performed is not made until the resource needs of other operations are known.

In adapting the scheduling algorithm to this architecture, Stotzer and Leiss [65] use the modulo-scheduling algorithm, modified to eject all operations of the same type when a conflict arises. We term this policy *eager ejection*. This decision allows for maximum flexibility because the scheduler is then able to use a different functional unit when those operations are rescheduled. They use bidirectional slack scheduling.

In addition, the priority an operation is given in the list-scheduling algorithm is increased if an operation uses a critical resource (one used 90% of the time in the initiation interval) and is based on the range of legal locations in which it can be placed. However, when the functional unit is unknown, it is not clear how this determination is made.

In the results of Stotzer and Leiss, the scheduler always finds optimal results. The ability to find optimal schedules is as much a function of the input as of the scheduler. Many of the loop kernels in the test suite had trivial cyclic dependences.

One interesting measurement taken in this research [65] is to tally the number of times an operation is placed, which measures the amount of effort performed in unscheduling operations. As compared with other software-pipelining algorithms, this method results in a much higher average attempt ratio (2.65 compared with 1.03). This is attributed partly to the difference in the latencies of the actual operations scheduled in the two tests, but is likely a direct result of the eager ejection used to permit the late binding of an operation to a specific functional unit. Thus, we have a compromise — a tighter schedule at the cost of greater effort.

18.5 Kernel Recognition

Although modulo-scheduling algorithms create a kernel by scheduling one iteration such that it is legal when overlapped by *II* cycles, other techniques schedule various iterations and must recognize when a kernel has been formed. Some authors term this type of software-pipelining algorithm an *unrolling algorithm* [56], but the term *kernel recognition* is more accurate because no physical unrolling may be present. Proponents of modulo scheduling point to the need to search for a kernel as a flaw, whereas proponents of kernel recognition counter that searching for a pattern can be done with hashing[10] and is much more efficient than repeating the scheduling for various goal initiation intervals. Kernel recognition proponents also argue that the ability to achieve fractional rates effortlessly makes their algorithms superior. Modulo-scheduling algorithm enthusiasts claim that the *II* rarely has to be incremented over its original minimum initiation interval [55]. Obviously, modulo scheduling can achieve fractional rates by replicating the loop body before scheduling. However, this increases complexity and may result in full iterations in prelude and postlude (that would be removed).

A first attempt at kernel recognition techniques is as follows:

1. Unroll the loop and note dependences.
2. Schedule the various operations as early as data dependences allow.
3. Look for a block of consecutive instructions that are identical to the blocks after it. This block represents the new loop body. Rewrite the unrolled iterations as a new loop containing the repetitive block as the loop body.

18.5.1 Perfect Pipelining

Perfect pipelining combines code motion with scheduling. It achieves fractional rates and handles general (*dif*, *min*) pairs. Techniques to assist the formation of a pattern are somewhat *ad hoc*.

Historically, the effectiveness of local scheduling has been limited due to the small size of basic blocks. Architectural models in which multiple tests can be performed within a single instruction greatly enhance the degree of parallelism achieved. Aiken and Nicolau introduce the perfect pipelining algorithm[11] for use with this more general machine model [2, 4–6, 16, 51]. This method is important in that it reframes the problem by changing the parameters. It answers the question, “How would software pipelining be effected if the architectures were modified to support it better?”

Perfect pipelining is somewhat similar to the method of Su, Ding and Xia [66] in that the loop is prescheduled, unrolled and overlapped, but it is more sophisticated in that operations may move

[10]For the general resource model, hashing must be done on an encoding of the entire state of the scheduler at each point in time.

[11]Aiken does not consider perfect pipelining to be an algorithm but instead a framework upon which algorithms can be built. Thus, for every reference to “the perfect pipelining algorithm,” the reader should substitute “one of many possible algorithms using the perfect pipelining framework.” The algorithm referenced is one that Aiken considers in [2].

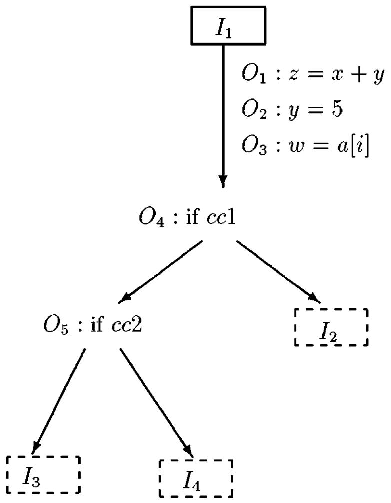


FIGURE 18.21 Perfect pipelining instruction model.

independently after the prescheduling and loops that span multiple blocks are easily accommodated. Perfect pipelining (as EMS) is more powerful than previous techniques in that it can take advantage of an architecture with multiway branching. A multiway branching architecture allows an instruction to have several branch target locations based on multiple Boolean conditions. Figure 18.21 illustrates a basic instruction executable in one time unit in this model. In one cycle, O_1, O_2 and O_3 are executed and control is transferred to one of I_2, I_3 or I_4, depending on the values of $cc1$ and $cc2$ (which are predicates set before this instruction). Such instructions are called *tree instructions* [23]. All the assignment operations are executed and a destination is selected simultaneously. To take advantage of this type of architecture, a type of global code motion, called *migration*, is implemented [3]. Migration is an improvement over early trace scheduling [25] in that copies can be merged, and code motion is tied to code-correcting compensation for that motion directly so that the cost benefit can be considered. Later versions of trace scheduling have adopted these improvements [27].

The types of code motion are enumerated in [26]. The addition of the ability to join multiple copies of an operation as they move past a branch point, termed *unification*, is important in that it reduces code explosion. The importance of unification is that the multiple copies of an operation generated by moving past branch points can often be recombined.

Aiken and Nicolau perform global code motion within the loop, before software pipelining, to simplify the initial schedule. Once global code motion has been performed, the loop is unrolled an unspecified number of times, and the result is scheduled assuming infinite resources. In a loop, performing code motion before unrolling significantly speeds up pipelining. The pipelining algorithm does not need to repeatedly perform similar code motions within each copy.

The assumption of infinite resources in the initial scheduling step is made because the algorithm requires unhampered motion. If the constraint of finite resources is enforced and I_1, I_2 and I_3 are sequentially ordered, motion of an operation between instructions I_3 and I_1 would be limited by that fact that the operation may not be able to temporarily reside in instruction I_2, because of resource conflicts with the operations that are placed there first. Thus, instead of allowing free code motion, the algorithm would be hindered by the race condition: which operation got to node I_2 first.

Perfect pipelining allows the pattern to form naturally. As each successive instruction is scheduled, one must determine whether the schedule has begun to repeat itself. Let the state of the schedule at a specific instruction represent the set of information that controls which operations may be scheduled in succeeding instructions. For a general resource model, state must include resources committed by the previous scheduling of operations with persistent resource requirements. For operations with nonunit latencies, the state must include the concept of elapsed time between dependent operations. If two nodes can be reduced to the same state, they are said to be *functionally equivalent*. An instruction that is functionally equivalent to an earlier node can be replaced with a branch to the first instruction, thus creating a loop. The problem of determining when two nodes are functionally equivalent is discussed in [9].

18.5.2 Petri Net Model

The Petri net algorithm uses the rich graph theoretical foundation of Petri nets to solve the problem of kernel recognition. It achieves fractional rates, works for general (*dif*, *min*) pairs and is extendible. It has the power of perfect pipelining, but has replaced ad hoc techniques with mathematically sound approaches.

The Petri net model of Allan, Rajagopalan and Lee provides a valuable solution to the problems associated with the formation of a pattern, both in terms of forcing a pattern to occur and recognizing a pattern has formed [10, 54]. The ability to recognize when a pattern has formed and to aid the efficient formation of such a pattern is essential to kernel recognition-type software pipelining. In other techniques, kernel development needs to be assisted by manipulating the final schedule or look-alike instructions may masquerade as loop entry points when a repeating pattern has not been achieved [39]. Both problems are elegantly eliminated using Petri nets. This algorithm is an improvement over the Gau, Wong and Ning [28, 29] algorithm, which suffers from the following limitations:

1. *Dif* values greater than one are not handled in the Gao algorithm except by replicating the code so all *dif*s are zero or one.
2. Initiation intervals of less than two cannot be achieved without replication because the acknowledgment arcs create cycles and thus force an initiation interval of two.
3. Gao's method is complicated by the addition of superfluous arcs and frequently achieves non-optimal initiation intervals due to the fact that cycles having $min_\theta / dif_\theta > II$ are inadvertently created.

A Petri net $G(P, T, A, M)$ is a bipartite graph having two types of nodes, places P and transitions T, and arcs A between transitions and places. Figure 18.22(a) shows a Petri net. The transitions are represented by horizontal bars whereas places are represented by circles. An initial mapping M associates with each place p, $M(p)$ number of tokens such that $M(p) \geq 0$. A place p is said to be marked if $M(p) > 0$. Associated with each transition t is a set of input places $S_i(t)$ and a set of output places $S_o(t)$. The set $S_i(t)$ consists of all places p such that there is an arc from p to t in the Petri net. Similarly $S_o(t)$ consists of all places p such that there is an arc from t to p in the Petri net.

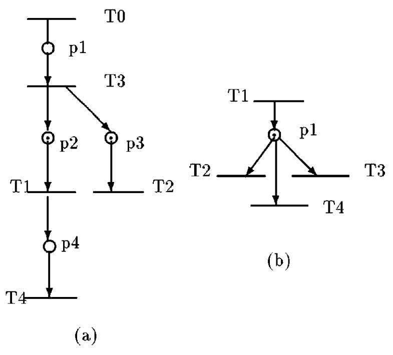


FIGURE 18.22 Petri net. (a) Concurrency: transitions T_1 and T_2 are independent of each other and can fire simultaneously; (b) conflict: transitions T_2, T_3 and T_4 cannot fire simultaneously because the input place p_1 can pass the token to only one successor.

The marking at any instant defines the state of the Petri net. The Petri net changes state by *firing* transitions. A transition t is ready to fire if for all p belonging to $S_i(t)$, $M(p) \geq w_p$, where w_p is the weight of the arc between p and t. The reader may see some similarity between transitions and runners in a relay race. One runner cannot run (fire) until that runner has been given the baton (token). However, in this case, a runner can pass a baton to several teammates simultaneously and one runner may have to receive a baton from each of several teammates before continuing.

When a transition fires, the number of tokens in each input place is decremented by the weight of the input arc whereas the number of tokens in each output place is incremented by the weight of the arc from the transition to that place. All transitions fire according to the earliest firing rule; that is, they fire as soon as all their input places have sufficient tokens. In Figure 18.22(a), no arcs exist between transitions T_1 and T_2. These transitions are independent of each other and hence can be fired concurrently. However, T_4 cannot fire until T_1 has fired and placed a token in place p_4. Therefore, T_4 is dependent on T_1. In Figure 18.22(b), place p_1 contains only one token that can be used to fire one of transitions T_2, T_3 or T_4. This represents a conflict that can be resolved using a suitable algorithm.

The Petri net models the cyclic dependences of a loop. A DDG shows the must-follow relationship between operations (nodes). However, a DDG cannot show which operations are ready to be scheduled at a given point in time. A Petri net is like a DDG with the current scheduling status embedded. Each operation is represented by a transition. Places show the current scheduling status. Each pair of arcs between two transitions represents a dependence between the two transitions. When the place (between the transitions) contains a token, it is a signal that the first operation has executed, but the second has not.

The firing of a transition can be thought of as passing the result of an operation performed at a node to other nodes that are waiting for this result. If the Petri net is cyclic, a state may be reached when a series of firings of the Petri net take it to a state through which it has already passed. The state of the Petri net is represented by the token count at each place. Because all decisions are deterministic, an identical state (with an identical reservation table) means the behavior of the Petri net repeats.

Min values of greater than one are handled by inserting dummy nodes so that no *min* is greater than one. For example, an arc ($a \rightarrow b$, *dif*, 3) is replaced by arcs ($a \rightarrow t_1$, *dif*, 1), ($t_1 \rightarrow t_2$, 0, 1) and ($t_2 \rightarrow b$, 0, 1), where t_1 and t_2 are dummy nodes. This implementation increases the node count, but greatly simplifies the recognition of equivalent states because the marking contains all delay information. This algorithm handles *min* values of zero by doing a special firing check for nodes connected to predecessors by *min* times of zero. Thus, compile time is increased by accommodating *min* times of zero.

Arcs are added to the DDG to make the graph strongly connected. The benefit is that the rate of each firing is controlled; no node is allowed to sustain a rate faster than the slowest cycle dictates. As arcs are added, new cycles are created. If a new cycle θ' has a larger $min_{\theta'}/dif_{\theta'}$ than that contained in the original graph, the schedule is necessarily slowed down (*II* increased).

18.5.3 Vegdahl's Technique

Unlike all other techniques discussed in this survey, Vegdahl's [71] represents an exhaustive method in which all possible solutions are represented and the best is selected. Because software pipelining is NP complete, this makes the algorithm impractical for real code.

18.5.4 Evaluation

Perfect pipelining is important from a historical perspective due to the early work in exploration of pipelining involving branches within the body of the loop. Unification is an important addition to earlier global code motion techniques.

Conceptually, perfect pipelining has an advantage in that it is not forced to consider all paths through the loop simultaneously. In other words, various paths through the loop are able to achieve a different initiation interval. The frequency of achieving optimal overall results is hampered by the ad hoc nature of the scheduling.

The main disadvantage of perfect pipelining is the difficultly of determining when two nodes are functionally equivalent. Other problems include the generation of loops that contain more copies of the original iteration than needed and the need to help the pattern develop.

The Petri net model is another excellent choice for software pipelining due to its strong mathematical orientation and flexibility in adapting to a wide variety of constraints. General reservation models pose no problem for this adaptable method. It produces excellent schedules, with its only drawback the need to search for a pattern.

Vegdahl's method is an interesting theoretical tool. Because of its exponential complexity, it is not a practical technique, but serves as an "optimal" solution for small code size. The method could be adapted for use with persistent resources, but would significantly increase the runtime. Perhaps Vegdahl's method could be used in combination with other techniques. For example, if modulo scheduling was used to determine span and *II*, Vegdahl's exhaustive technique could be greatly restricted to explore solutions with span and *II* slightly better than the achieved. A parallel implementation of a greatly reduced search space could make the algorithm reasonable for a broader class of problems.

18.6 Enhanced Pipeline Scheduling

18.6.1 The Algorithm

Enhanced pipeline scheduling (EPS) [22, 23, 49] integrates code transformation with scheduling to produce excellent results. One important benefit is the ability to schedule various branches of a loop with different *II*.

Nakatani and Ebcioğlu [50] propose an algorithm unlike either modulo scheduling or kernel recognition algorithms. EPS uses a completely different approach to software pipelining, building on code motion (like perfect pipelining), but retaining the loop structure so no kernel recognition is required. This algorithm is quite complicated both to understand and to produce the code. However, it has benefits no other algorithm achieves because it combines code transformation with software pipelining. In a more recent version of the algorithm [49], the algorithm is termed *selective scheduling with software pipelining*.

Nonnumerical programs typically have many branches, resulting in small basic blocks. Some techniques [38] use branch probability techniques to determine which parts of the code to optimize. Although a loop control branch is generally taken (because we spent more time in the loop than skipping it), other branches are executed in approximately half the time, making optimization more difficult. For some types of algorithms static branch prediction falls far short as a reliable tool. Mispredicted branches can actually slow execution because of the speculative code that was generated to make a nontaken branch more efficient. Thus, selective scheduling tries to limit the amount of speculatively executed code.

Techniques with a fixed initiation interval may be less than desirable if one path through the loop is significantly longer than another path (due to data dependences). In traditional software pipelining, all paths through the loop have the same length. The goal of the selective scheduling algorithm is to allow various paths through the loop to take different amounts of time, instead of penalizing all paths because of the length of the longest one. Requiring all paths to take the same amount of time makes the initiation interval the worst case (based on dependences and resources) and is, therefore, inefficient. For example, consider the example of Figure 18.23, adapted from [49]. In this case, when

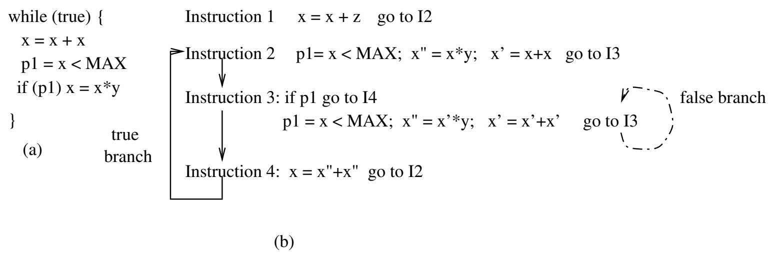

while (true) {
x = x + x
p1 = x < MAX
if (p1) x = x*y
}
(a)
true
branch
Instruction 1 x = x + z go to I2
Instruction 2 p1= x < MAX; x" = x*y; x' = x+x go to I3
Instruction 3: if p1 go to I4
p1 = x < MAX; x" = x'*y; x' = x'+x' go to I3
false branch
Instruction 4: x = x"+x" go to I2
(b)

FIGURE 18.23 (a) Original loop; (b) results of pipelining.

the condition is true, the loop takes three instruction cycles (instructions 2, 3 and 4); however, when the condition is false, the loop only takes one instruction (instruction 3).

Code motion pipelining can be described as the process of shifting operations forward in the execution schedule. Because operations are shifted through the loop control predicate, they move ahead of the loop, into the loop prelude and they move back into the bottom of the loop, as operations from a later iteration. The degree to which shifting is able to compact the loop depends on both the resources available and the data dependence structure of the original loop. The algorithm has some similarity to perfect pipelining, in that it uses a modification of the perfect pipelining global code motion algorithm, but differs significantly because EPS manipulates the original flow graph to create the pipelined loop and does not require searching for a pattern in unrolled code. Because the loop is manipulated as a whole instead of individual iterations, EPS does not encounter the problems of pattern identification inherent in perfect pipelining.

Selective scheduling is accomplished by the following steps:

1. "Cut" selected edges in the loop so that the code becomes acyclic.
2. Compute the set of all right-hand sides that can move to the top of the DDG. Right-hand sides are used instead of entire operations because more parallelism is exposed. Thus, for example, a sequence of operations $m = x + 1; x = y + z; n = x + 2$ might be transformed as $x' = y + z; m = x + 1; x = x'; n = x + 2$. Separating $x = y + z$ into $x' = y + z; x = x'$ allows the $y + z$ to move ahead of the assignment to m (perhaps much earlier). The residual copy operations (such as $x = x'$) can often be done cheaply or even eliminated.
   Notice that we can move the assignment to n past the copy operation ($x = x'$) by renaming. This operation is very important due to the large number of copy operations generated. Thus, the code sequence becomes $x' = y + z; n = x' + 2; m = x + 1; x = x'$.
3. Compute a set of available right-hand sides over all execution paths, combining those that occur on all successors of a point.
4. Evaluate the available right-hand sides in terms of how speculative they are.

Consider the example of Figure 18.24 in which available expressions have been accumulated at the beginning of the block. In this example, the operation $if\, cc1$ is nonspeculative because it is always executed. Similarly, $d + 1$ is nonspeculative. The operations a^*b and b^*b are both speculative. The operation a^*b is used if the true branch is taken. However, if the false branch is taken, the value of a has been reassigned. By substituting the assignment to a along the false branch, the expression becomes b^*b.

The computation of available instructions is computed each time a parallel instruction is to be scheduled. However, the recomputation of available expressions is inexpensive because of incremental computation. To reduce overhead, an intermediate availability set (which is computed

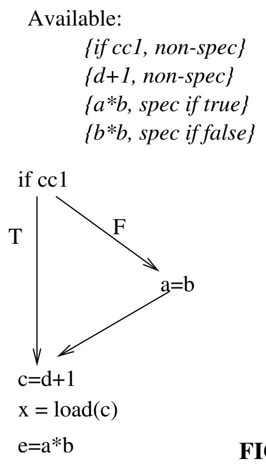


FIGURE 18.24 Original code labeled with the set of available expressions.

as part of the data flow computations for available sets) is left at the beginning of every basic block, making incremental computation of available sets simpler. Also, because operations from different branches are combined, code explosion is reduced. In this example, the available expression $d + 1$ is replicated as it passes the join point, but the copies are recombined as it moves past the branch point.

To determine the scheduling preference between operations of the availability set, a numerical speculative attribute (*spec*) is assigned; the higher the value assigned, the more speculative is the operation (and the less desirable for scheduling). Initially all operations are assigned a *spec* value of 0. Available operations are computed using standard data flow techniques, examining the blocks in reverse topological order.

As the operations are moved in the data flow graph, the speculative attribute is updated as follows: all operations in the basic block n that are not data dependent on each other are moved to the availability set for n with their *spec* attribute unchanged. If an operation exists in all the available sets of n's immediate successor blocks, it can be moved to *available(n)* (the available set for basic block n) and is assigned the highest of its *spec* attributes in the successor blocks. If an operation is present in only one of the successor blocks, its *spec* attribute is incremented by one. For example, consider Figure 18.25. Notice that operation $y + 1$ has a speculative value of 0 because it is useful all on paths. Operation $a + f$ has a higher speculative value than $b + e$ because it is useful on fewer paths. The idea is that nonspeculative operations (*spec* value is 0) should be scheduled first (as they are guaranteed to be useful). Operations that are more speculative are less desirable, because their chance to be productive is less.

Other methods [15] define the degree of speculativeness of an operation as the number of branches that control its execution. However, this is not accurate. For example, in Figure 18.25, operation $a + e$ is not control dependent on any branch, yet it becomes speculative as it moves past branch points and values are substituted.

Each basic block header stores the set of live registers (using regular data flow techniques). To reduce the amount of code expansion (as well as scheduling time), a software look ahead window is used to limit the distance operations that are moved to reach the availability set. This also reduces register pressure because code that moves long distances tends to increase register lifetimes.

Once an operation from the available set has been scheduled, only the available sets and live register sets on that path from which the operation came need to be recomputed. Restricting the updates to the paths involved is a significant reduction in effort.

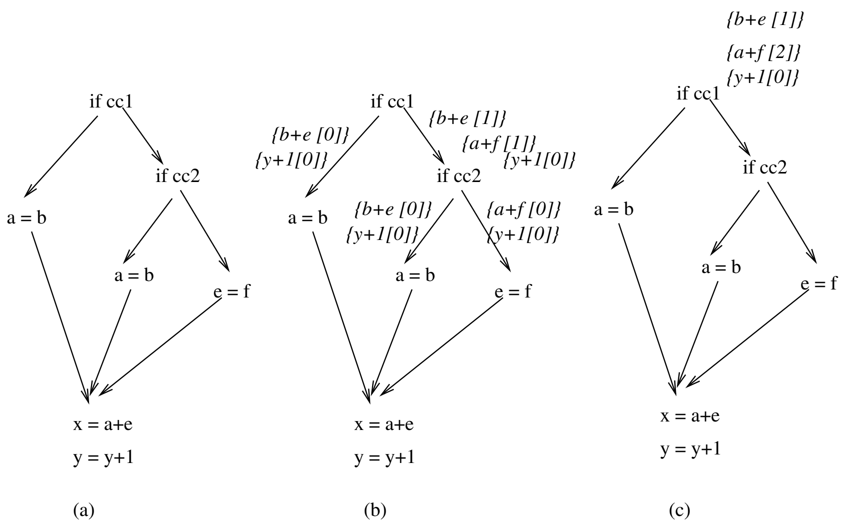


FIGURE 18.25 (a) Original code; (b) availability sets along branches, with speculative values in square brackets; (c) availability sets for initial basic block.

In the original form of this algorithm, software pipelining occurred when operations were moved across the back edge,[12] allowing them to enter the loop body as operations from future iterations. The pipelining algorithm iterated until all the loop-carried dependences had been considered. The pipeline prelude and postlude were generated by the automatic code duplication that accompanied the global code motion.

In the current form [49], this algorithm attacks the problem more aggressively by deciding which edges of a loop to consider as back edges. The algorithm cuts some edges to turn the cyclic graph into an acyclic one. The availability set is computed and a parallel operation group is formed. This parallel operation group is placed on each edge that was cut. Then, the old edges are restored, new edges are cut and the process repeats again. Forward and backward edges are identified elegantly by manipulating the sequence number of each operation. An edge from a higher to a lower sequence number is considered to be a back edge. After a parallel group is scheduled, all the operations of the group are given new (higher) sequence numbers, effectively moving the instructions at the end of the next iteration's acyclic graph.

The prelude and postlude is generated automatically as code is moved to cut edges. The technique can be extended to nested loops by first applying software pipelining to the inner loop, and then treating the inner loop as fixed, allowing the prelude and postlude code to become software pipelined into the outer loop.

This model can handle *min* times of greater than 1 by inserting dummy nodes. However, this increases the complexity of the algorithm because of the increased number of nodes. It should be noted that the basic algorithm cannot handle persistent, irregular resources, but the authors contend that it could be done using reservation tables [49].

[12]A back edge of a data dependence graph is an edge whose head dominates its tail in the flow graph and is used to locate a natural loop [1]. A node a dominates a node b in a graph if every path from the source to b must contain a.

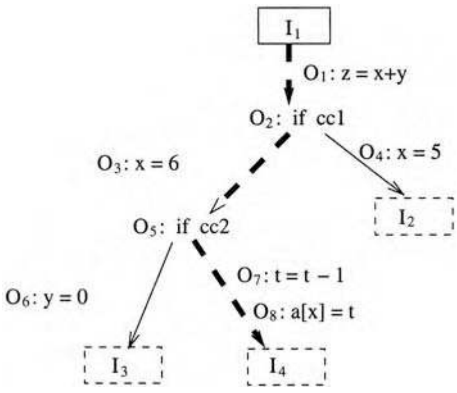


FIGURE 18.26 A tree instruction used in enhanced pipeline scheduling.

18.6.2 Instruction Model

EPS is a powerful algorithm that can utilize a multiway branching architecture with conditional execution. The conditional execution feature makes this machine model more powerful than the original perfect pipelining machine model that includes only multiway branching. Recent versions of perfect pipelining have included multipredicated execution [51]. To take advantage of multi-predicated execution, a more powerful software pipelining algorithm is required.

Figure 18.26 shows a typical control flow graph (CFG) node, I_1,[13] termed a *tree instruction.* I_2 through I_4 are labels of other CFG nodes. The condition codes $cc1$ and $cc2$ are computed before the node is entered, and only those statements along one path from the root to a leaf are executed. All the operations on the selected path are executed concurrently, using the old values for all operands. Operations in a tree instruction have no dependences on each other. For example, in Figure 18.26, one possible path is shown with dashed arrows. If $cc1$ is true (left branch) and $cc2$ is false when I_1 is executed, operations O_1, O_3, O_7 and O_8 are executed simultaneously, then control is transferred to I_4. (This path is shown by a heavy, dashed path in the diagram.) Thus, the assignment to t by O_7 does not affect the use of t by O_8. Two types of resource constraints are placed on this machine model. One is limited by the total number of operations that can be contained within the tree instruction. Another is limited to a fixed number of different paths through the tree instruction.

With conditional execution, operations can be placed on any branch of a CFG node, unlike a traditional machine model in which all operations must precede any conditional branch operations. Although EPS can utilize such features, the scheduling technique does not require this powerful of an architecture. Other architectures can be represented by the tree instruction by restricting the form of the instruction.

18.6.3 Global Code Motion with Renaming and Forward Substitution

Nakatani and Ebcioğlu use renaming and forward substitution to move operations past predicates and shorten dependence chains involving antidependences [50]. Renaming involves replacing an operation such as $x = y\ op\ z$ with two operations $x' = y\ op\ z; x = x'$. Because $x' = y\ op\ z$ defines a variable that is only used by the copy statement $x = x'$, the assignment to x' is free to move outside the predicate. When $x' = y\ op\ z$ moves before the predicate that controls its execution, the

[13]A control flow graph (CFG) is a graph in which nodes represent computations and edges represent the flow of control [1].

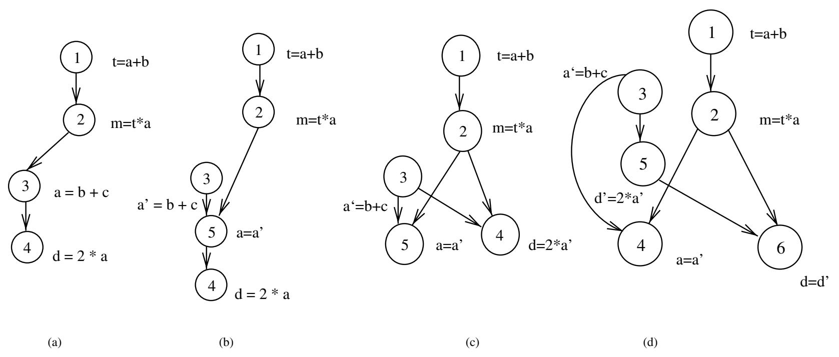


FIGURE 18.27 (a) Original dependence graph; (b) dependence graph after renaming $a = b + c$; (c) after forward substitution; (d) dependence graph after renaming $d = 2 \cdot a$.

statement is said to be speculative. Figure 18.27 illustrates the shortening of a data dependence chain (involving an antidependence) and a control dependence chain using renaming of a. Figure 18.27(a) shows the original dependence graph whereas Figure 18.27(b) shows the graph after renaming. The dependence between n_5 and n_4 can be eliminated by forward substitution. For an assignment statement *var* = *expr*, when uses of *var* in subsequent instructions are replaced by *expr*, it is termed *forward substitution*. This is particularly useful if, as a result of the forward substitution, the operations can be executed simultaneously. In Figure 18.27(c), forward substitution is used to change $d = 2 \cdot a$ to $d = 2 \cdot a'$ because the a referenced has the value of a'. Figure 18.27(d) shows that $d = 2 \cdot a$ is renamed. Notice that in this case, the length of the dependence chain for the graph is decreased.

To see the flexibility of this model, node 2 could also represent a predicate and the same transformations used on the data dependences can also be applied to the control dependences.

Forward substitution may collapse true dependences that prevent code motion. A true dependence between two operations O_1 and O_2 may be collapsible with forward substitution if (1) O_1 is a copy operation or (2) both operations involve a constant immediate operand. Forward substitution is performed if a true dependence can be collapsed.

Figure 18.28 shows an example in which both forward substitution and renaming are required to move an operation. The diagram shows part of a CFG before and after operation O_6 (shown in bold type) is moved from instruction I_4 to instruction I_2. Multiple tree instructions are shown in the figure. Each rectangular node is the entry point of a tree instruction. O_6 has a true dependence on O_4 that involves an immediate operand, making this dependence a candidate for forward substitution. First, y is renamed giving $O_6 : y_a = z + 6$ and $O_{6a} : y = y_a$. Next, forward substitution is performed giving $O_6 : y = z + 4 + 2$. Constant folding is then performed giving $O_6 : y = z + 6$.

18.6.4 Pipelining the Loop

To perform pipelining, code is moved backward along the control flow arcs. The algorithm consists of two phases that repeat until all operations have an opportunity to move. In phase one, code from within the current loop body is moved as early as dependences allow. The first parallel instructions of the loop are termed *fences*. The name is derived from the fact that code is moved from the rest of the loop to a position as early as possible in the loop, but not past the fence. Hence, the fence bounds the code motion. In phase two, the fence instruction is duplicated and moved. Code from the loop body is duplicated as it moves past the top of the loop because there are two control predecessors.

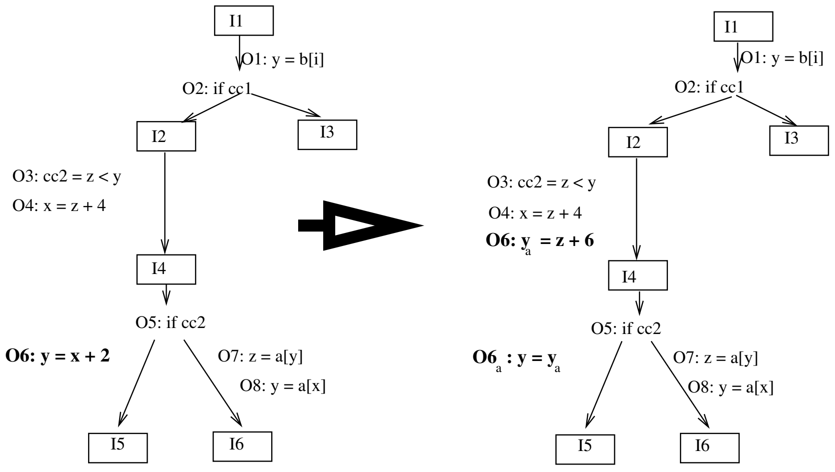


FIGURE 18.28 Example of code motion of O_6 with renaming and forward substitution of $x = z + 4$ for the x in O_{6a}.

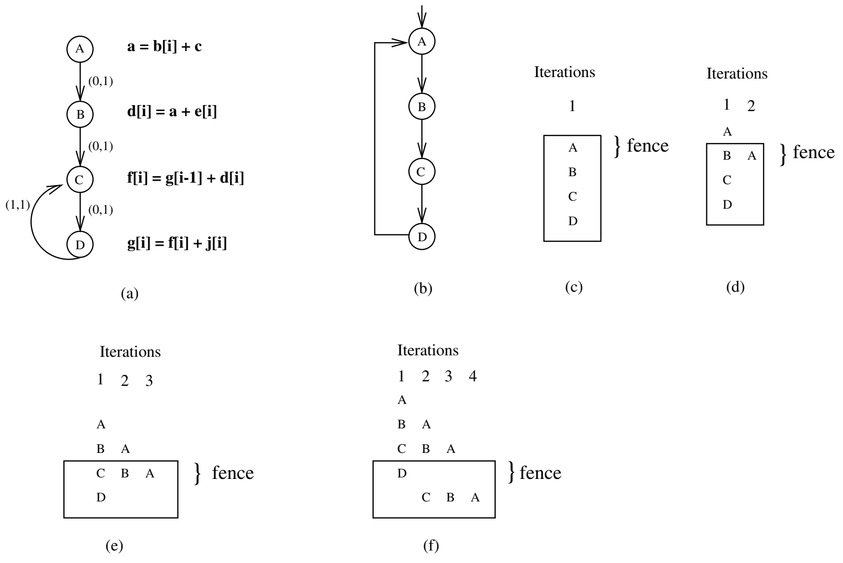


FIGURE 18.29 Selective scheduling. (a) Data dependence graph of loop; (b) control flow graph of loop; (c) after filling first instruction; (d) after filling second instruction; (e) after filling third instruction; (f) final software pipeline.

The code that moved out of the loop forms a prelude. The code that joins code at the bottom of the loop body represents work originally performed in a different iteration.

18.6.5 Extensions

The original EPS work is modified in [40] to remove some of the extra copies that are generated by EPS. The first step performs the code motion (which generates many copy instructions). The second step unrolls the loop a sufficient number of times so that the copies can be combined. Coalescing

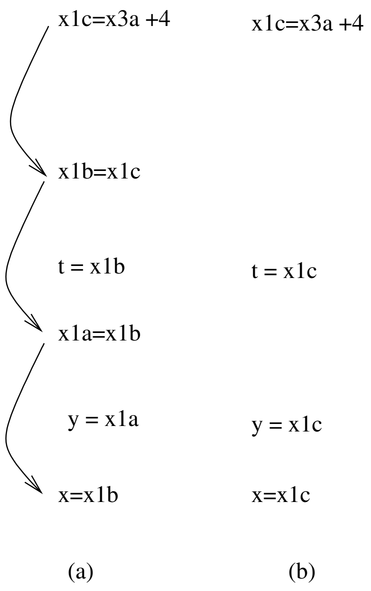


FIGURE 18.30 (a) Situation where coalescing can be done; (b) resulting code after coalescing.

eliminates copies by allocating the same register to both the source and the target of a copy. Obviously this can only be done if the live tracks do not overlap. For example, in Figure 18.30(a), various adjacent live chains are shown that are produced during the course of code motion, and have subsequently been unrolled. In Figure 18.30(b), the live tracks are combined and the variable names are substituted. All the variables involved in the chains can be stored in the same register if we rename all the uses. This technique has quite a special purpose, in that it eliminates the copies generated by this technique. One would also wonder if the unrolling were done before the code motion were performed, whether the need for coalescing would be reduced.

The work of Shim and Moon [64] adjusts EPS by duplicating common blocks to permit more efficient schedules when the same path is executed repeatedly. They use the term *cross-path pipelining* to refer to a schedule that is tuned to changes between the execution paths of a loop and the term *intrapath pipelining* for schedules that are tuned to repeated execution of the same path. Earlier versions of the EPS algorithm focused more on tight cross-path pipelining to the detriment of intrapath pipelining. The new algorithm, termed split-path enhanced pipeline scheduling (SP-EPS), first computes a modulo schedule for each path separately and then the schedules are combined. To facilitate separate pipelining of paths, the paths need to be separate — so they are first split via code duplication until the code becomes a treelike graph. This step can cause considerable code expansion. Intrapath pipelining is achieved through code motion. Cross-path pipelining is done only when the scheduling does not conflict with intrapath operations. Loop unrolling is performed to reduce antidependences caused from overlapping of iterations. Consistent with the goal, this method produces better schedules (approximately 6% speedup) than EPS when the transition between paths is low, but this is accompanied by a code expansion of about 15%. Interestingly, the improvement over EPS disappears when the number of processors is doubled.

18.6.6 Evaluation

EPS is unique. Not only does it deal with multiblock loops but also it always maintains a legal loop structure instead of trying to rebuild the loop. EPS handles general loops. Although renaming and forward substitution have been utilized for years, the degree to which they have been successfully employed in this technique is noteworthy.

EPS has several advantages over perfect pipelining. The algorithm increases the speed of convergence by retaining the loop construct and reducing the resulting code size. These techniques can also cause problems, because prematurely forcing operations to remain together can restrict the parallelism. These disadvantages are lessened if a majority of the loop dependences can be removed by the automatic renaming and combining. The most serious drawback is the unsuitability of the method for use with persistent resources. Because instructions are inserted or removed during the scheduling, persistent resources (that require a fixed set of resources a given offset from instruction initiation) cannot be accommodated. Fractional rates are not achieved.

18.7 Adapting Software Pipelining to Other Concerns

18.7.1 Reducing Cache Misses

Many of the pipelining techniques isolate the need for exploiting parallelism from the requirements of a real machine. Prefetching of data can minimize the effects of cache misses. However, capacity misses (which occur because the needed data exceed the capacity of the cache) can be reduced by considering the contents of the loop, referred to as *loop blocking* or *loop tiling*. Panda et al. [53] examine this issue. Although they do not specifically consider loop tiling with software pipelining, much could be gained by combining the techniques. Conflict misses occur in limited-associativity caches when several data elements compete for the same cache location. Conflict misses can seriously hamper the effectiveness of software pipelining by increasing the initiation interval.

Panda et al. [53] introduce a technique for data alignment that is termed *padding*. Basically, elements that compete for the same cache location are controlled. Thus, the concern is with conflicts between data in the same array (self-interference) and data between arrays.

Sánchez and González [63] consider the interaction between software prefetching and software pipelining. They point out that, although achieving high throughput or low-register pressure has been considered by a variety of researchers, the effect of cache memory is often disregarded. The cache performance is a critical factor in performance due to the fact that memory speeds are so much slower than processor clock speeds.

Whereas a lockup free cache is advantageous because it allows the processor to continue after a cache miss, the advantages are limited because in software pipelining, stalls occur after the cache miss due to data dependencies on the missing data. Cache misses are normally dealt with via prefetching, either by moving the memory operations away from the code that uses the fetched data (termed *binding prefetching*) or by inserting special prefetch instructions that perform a cache lookup but do not actually store the data in a register (terming *nonbinding prefetching*). Binding prefetching increases the lifetime of the variable and hence affects register pressure. Nonbinding prefetching increases the number of memory requests and may cause the initial interval of the software pipeline to increase to accommodate the extra instructions. Sánchez and González illustrate that common software-pipelining techniques that fail to account for normal cache memory behavior produce far from optimal results.

Sánchez and González propose a technique that is based on early scheduling of carefully chosen operations. Operations involved in recurrences are monitored closely because modifications to the recurrence code can increase register pressure as well as increase the length of the initiation interval due to the cyclic dependencies. They compute two alternative schedules, one of which affects the stall time whereas another affects the length of the prologue and epilogue. If the number of times the loop is executed (termed the *trip count*) is large, having more prologue and epilogue code is not as serious because the code expansion is considered to be worth the space penalty to obtain loop code that is efficient due to the amortized cost. Thus, the number of iterations to be performed affects the choice of which schedule is preferred, because the costs are amortized.

To determine the best way of reducing cache misses, locality analysis is conducted by considering the following factors:

1. Determine which type of reuse is present. Self-dependence is when the same operation repeatedly accesses the same cache location, whereas group dependence is when elements are repeatedly accessed that were most recently used by different instructions.
2. Determine whether two static instructions always interfere with each other in the cache.
3. Determine which references cannot take advantage of reuse because they have been displaced from the cache.

From this information, a schedule that minimizes the effect of recurrences is constructed.

Sánchez and González compare their proposed technique with various strategies for using prefetch instructions: (1) always insert prefetch instructions, (2) insert prefetch instructions for references without temporal locality and (3) insert prefetch instructions only for references with no locality. They show that their technique is superior to others with prefetching and that among the prefetching strategies, one is not superior to another, but varies with the particular program and architecture. Thus, it appears that more work is needed to identify when prefetching is advantageous.

18.7.2 Itemization of All States

Milićev and Jovanović [48] propose a formal model for software pipelining in the presence of conditionals. Predicated execution allows for code from multiple blocks to be scheduled as one block. Their observation is as follows. Suppose operation O_j from iteration 2 is dependent on operation O_i that is conditionally executed. If O_i does not really need to be executed, a better schedule could likely be produced.

In the movement of predicated code, it could happen that the execution of an operation is conditional on the value of a predicate computed for a previous iteration. Similarly, if we allow speculative execution, a predicated operation in one iteration could be predicated on a predicate computed in a different iteration. They use a predicate matrix in which the column indicates the iteration for which the predicate is of interest, and the row is the number of different predicates contained in the code.

In the code segment Figure 18.31, let $\langle p1\ 0\rangle$ indicate that the operation is predicated on predicate $p1$ from the current iteration and $\langle p2 - 1\rangle$ indicate that the operation is predicated on predicate $p2$ from the prior iteration. A predicate matrix (also termed a *state matrix*) for a single iteration that depends on two predicates ($p1$ and $p2$) from three iterations (previous, current and next) may look something like that of Figure 18.32.

The next state matrix is obtained by shifting all values one place to the left: the left edge elements are discarded and right edge values become the next predicate values. Some values may be blank. For example, if the value of $p2$ from a previous iteration was not used, $\langle p2\ \ -1\rangle$ would be blank.

Operation	Predicate
$x = a + b$	$\langle p1\ \ 0\rangle$
$a = b$	$\langle p2\ \ -1\rangle$
$b = t$	$\langle p2\ \ -1\rangle$
$m = b - 1$	$\langle p1\ \ 1\rangle$

FIGURE 18.31 Example of predicated code.

$$\begin{array}{ccc} \langle p1\ \ -1\rangle & \langle p1\ 0\rangle & \langle p1\ \ 1\rangle \\ \langle p2\ \ -1\rangle & \langle p2\ 0\rangle & \langle p2\ \ 1\rangle \end{array}$$

FIGURE 18.32 Predicate matrix.

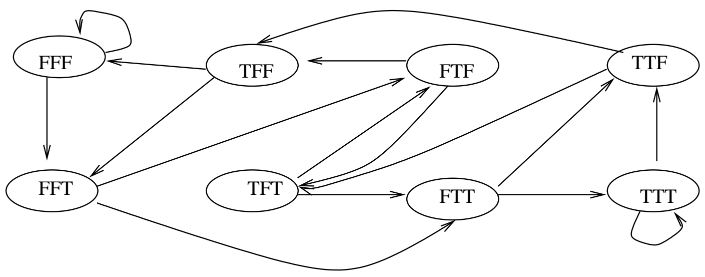


FIGURE 18.33 Finite state machine for a single predicate.

There are 2^k possible state matrices, where k is the number of nonblank values in the matrix. The idea is to let each possible state matrix be a node in a finite state machine (FSM) so that we can schedule each unique state individually.

For a simple example, consider a single predicate used over three iterations:

$$\langle \texttt{p1} \ \ -1\rangle \ \ \langle \texttt{p1} \ \ 0\rangle \ \ \langle \texttt{p1} \ \ 1\rangle$$

The FSM is shown in Figure 18.33, in which each node is labeled with the contents of the predicate matrix. Note that arcs mark possible next states for the following iteration.

Associated with each node in the FSM is the DDG given those values for the predicates. Thus, operations that would not be performed due to the specific predicate assignment of an iteration are eliminated from the DDG. Now code motion between nodes of the FSM is performed. For example, if the dependence graphs of two adjacent nodes are concatenated, the critical path may be shorter than the sum of their individual critical paths. In this case, moving code between them may reduce the critical path of one or both of them. As is common with code motion, one needs to be careful to preserve proper semantics. Consider an operation from state TTF named O_i that is predicated by $\langle p1\ 0\rangle$. If this operation is moved to a successor of TTF, it would be predicated by $\langle p1 - 1\rangle$. For this to be legal, we must move O_i to both successors of TTF (states TFF and TFT).

Similarly, consider an operation from state FTF named O_j that is predicated by $\langle p1\ \ 0\rangle$. If this operation is moved to a predecessor of FTF, it would be predicated by $\langle p1\ \ 1\rangle$. For this to be legal, we must move O_j to both predecessors of FTF (states FFT and TFT).

Some of the resulting states may have some code in common that can be combined with a type of unification. Although this technique may be very expensive, due to the exponential code explosion, it is an interesting idea. The key idea is that particular paths through the code should be optimized. In this case, every path is allowed to be optimized separately.

18.7.3 Register Pressure

One major concern with software pipelining is the high register demand. Although each loop invariant needs only to be stored once for all iterations, loop-dependent variables have a separate value for each iteration. When multiple iterations overlap, more registers are needed to store the variable values. In the work by Zalamea et al. [78], the problem of using spill code to reduce register pressure is considered.

Many software-pipelining methods are not sensitive to the limited number of registers available. When the number of available registers is exceeded, one option is to reschedule with a longer initiation interval, but this solution normally results in less efficiency. A second solution (spilling) is to move a register value to memory (and then restore it when needed) to save registers. Earlier research indicates that rescheduling after increasing the *II* tends to generate worse schedules, but this is not always the case.

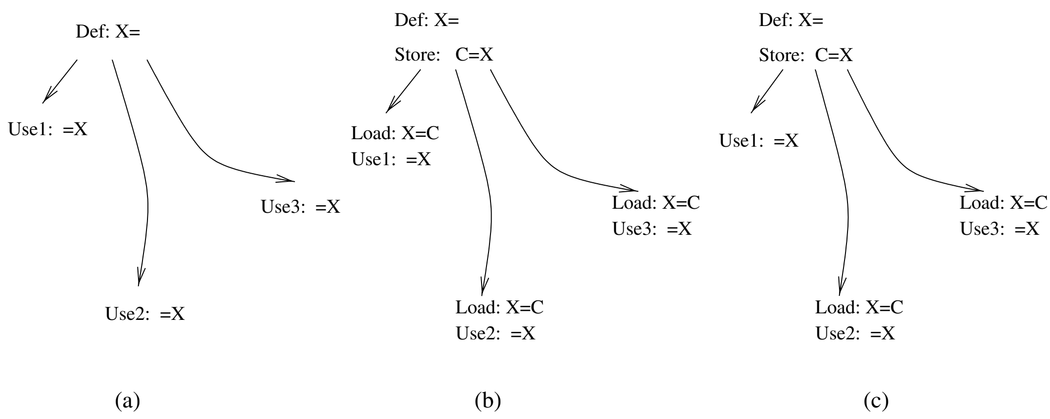


FIGURE 18.34 (a) Original code; (b) total spilling of variable; (c) spilling based on use.

One way of inserting spill code is to spill a variable totally, which means that there is a store after the generation of a variable value and a load of the value before every use of that value. Another, less aggressive technique spills individual uses of a variable. Thus, instead of requiring every use of a spilled value to have an associated load, one might decide to let some uses of the value use the original value and only restore values for uses that are further from the definition. For example, in Figure 18.34(b) when spilling X, all uses of register X have an associated load of the stored value of register X; whereas in Figure 18.34(c), we let *Use*1 use the value of X that is still in memory (because data flow analysis does not allow X to be reused until *Use*1 is done with it) but *Use*2 must load a stored value of X.

In producing spill code, one must decide how many variables to spill and how to decide which ones to spill. This is a bit tricky to do, because the introduction of spill code may increase the *II*, but a larger *II* may make the spill code less necessary. Thus, the process is normally iterative. You generate spill code and then reschedule to make sure the *II* is not increased. If the *II* is increased, you remove the spill code and determine anew what spill code is required at the new *II*.

Some techniques try to spill as many variables as possible without increasing the *II*. The problem with this approach is that memory traffic may be increased unnecessarily, and little benefit may result from reducing register use below that required for scheduling.

Zalamea et al. [78] propose several ideas for spill value determination. They find the instruction cycle during which the largest number of registers are simultaneously live, and then spill only registers that are live during this critical instruction. Instead of spilling all the registers needed at once, they spill only part of them (as controlled by a quantity factor, which is a parameter to the algorithm). The idea is that you do not introduce so much spill code at one time that *II* is increased. When *II* is increased, the spilling needs are altered, and you have to start the scheduling over anyway. Similarly, a traffic factor is used to allow the algorithm to experiment with various amounts of increased memory traffic.

The idea in the Zalamea work is to allow the compiler to try several approaches to handling register pressure and pick the best solution. The results of this approach are good, but the scheduler can take almost twice as long to perform the scheduling. In addition, some evidence shows that a dynamic way of adjusting specific values of the parameters to their algorithm would be beneficial.

Altemose and Morris [11] have modified modulo scheduling for an IA-64 type of architecture to incorporate sensitivity to register pressure. In their experiments on 880 loop bodies, they were able to reduce the number of registers required in 52% of the 880 loop bodies. Many modulo schedulers place operations as early as possible, causing an increase in the number of values that are live simultaneously. Their pressure responsive pipelining (PRP) looks at the changes in register pressure that result from various placement locations. Basically, they just minimize the length of the lifetime of the variable affected by the scheduled operation based on predecessors or successors (in the dependence relationship) that have already been scheduled. Unscheduled operations are ignored. Evidently, scheduling based solely on lifetime length may increase the failure rate as they double the number of attempts at operation placement over what is normally allowed. They monitor the number of placements that have been attempted. When half of the total number of placements have been made, they turn off the PRP scheduling.

Because values that are loop invariant (stored in the static register file) are not affected by PRP scheduling, Altemose and Norris [11] chose only to consider the registers in the rotating files instead of all registers. Although the rotating register usage of many loops was unaffected by PRP scheduling, the average decrease over all loops in register usage was 6.6%. A 14% decrease occurred for those loops that had reduced register requirements. Interestingly enough, however, they state that the floating point register usage was unaffected in the vast majority of the cases because of the "higher difficulty in scheduling floating point operations" caused by increased latency of floating point operations and fewer floating point functional units.

In scheduling for embedded very long instruction word (VLIW) machines, both register and memory space are at a premium. Akturan and Jacome [7] propose a technique, termed the register-sensitive force-directed retiming algorithm (RS-FDRA), for coping with register constraints and code space constraints. The algorithm is similar in spirit to the modulo-scheduling algorithms in that a specific initiation interval is attempted, but the initiation interval is increased in the face of failure. The main focus is in the pipelining of inner loops of single basic blocks.

The RS-FDRA algorithm deals with two optimization problems:

1. Minimize register pressure and initiation delay.
2. Minimize register pressure and number of overlapped iterations.

The solutions manager generates several good solutions, which are evaluated based on smallest code size. Next possible schedules are considered based on reducing register pressure. Several heuristics are used to facilitate this selection. Register pressure is reduced by selecting solutions that have larger delays on edges connecting shared objects. They look at the earliest possible scheduling location for a node and the latest possible scheduling location (similar to what other algorithms do) but prune the search by identifying infeasible locations and eliminating locations with no available resources. Each operation is given an associated force function, which measures the change of concurrency associated with scheduling the operation in a given cycle. This function helps the scheduler decide which operation to schedule next. Those operations that need to be scheduled in the most sought after instructions are scheduled first. The locations that give minimal register requirements are given first priority. When the scheduler is unable to meet the demands of the schedule, either the total number of iterations executing concurrently is incremented or the initiation interval is incremented, depending on priority. Early tests indicate that this technique is very effective in reducing both code space and register pressure.

18.8 Conclusions

Software pipelining is an effective technique for extracting parallelism from loops that thwart attempts to vectorize or divide the work across processors. Although the speedup is modest, it should be noted that software pipelining succeeds where other methods fail and can be applied after other techniques have extracted coarse grain parallelism. The variety of architectures benefiting from software pipelining underscores its importance.

Although an NP-complete scheduling problem, software pipelining has numerous effective heuristics that have been developed. Both resource conflicts and cyclic dependences produce a lower bound on the initiation interval. Such lower bounds can be computed in polynomial time. In considering algorithmic features such as low complexity, ability to deal with conditionals, accommodation of resource conflicts and achievement of fractional initiation intervals, the current methods have various degrees of success.

Some researchers are applying artificial intelligence techniques to the problem of software pipelining. Beaty uses genetic algorithms for instruction scheduling [13]. O'Neill and Allan use genetic algorithms and simulated annealing to solve the problem of software pipelining [8, 52].

Acknowledgment

Portions of this chapter were excerpted from V.A. Allan et al., *ACM Comput. Sur.,* 27(3), 367–432, September 1995. With permission.

References

[1] A.V. Aho, R. Sethi and J.D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA, 1988.

[2] A. Aiken, Compaction-Based Parallelization, Ph.D. thesis, Department of Computer Science, Cornell University, Ithaca, NY, 1988.

[3] A. Aiken and A. Nicolau, A development environment for horizontal microcode, *IEEE Trans. Software Eng.*, 14(5), 584–594, May 1988.

[4] A. Aiken and A. Nicolau, Optimal Loop Parallelization, in Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation, Atlanta, GA, June 1988, pp. 308–317.

[5] A. Aiken and A. Nicolau, Perfect Pipelining: A New Loop Optimization Technique, in Proceedings of the 1988 European Symposium on Programming, Springer Verlag Lecture Notes in Computer Science, No. 300, Atlanta, GA, March 1988, pp. 221–235.

[6] A. Aiken and A. Nicolau, A Realistic Resource-Constrained Software Pipelining Algorithm, Technical report RJ 7743, IBM Research Division, San Jose, CA, October 1990.

[7] C. Akturan and M.F. Jacome, RS-FDRA: A Register Sensitive Software Pipelining Algorithm for Embedded VLIW Processors, in Proceedings of 9th International Conference on Hardware Software Codesign CODES '01, Copenhagen, Denmark, 2001, pp. 67–72.

[8] V.H. Allan and M.R. O'Neill, Software Pipelining: A Genetic Algorithm Approach, in PACT 94, Montreal, Canada, August 23– 26, 1994, pp. 311–314.

[9] V.H. Allan, R. Jones, R. Lee and S.J. Allan, Software pipelining, *ACM Comput. Surv.*, 27(3), 367–432, September 1995.

[10] V.H. Allan, M. Rajagopalan and R.M. Lee, Software Pipelining: Petri Net Pacemaker, in Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism, Orlando, FL, January 1993, pp. 15–26.

[11] G. Altemose and C. Morris, Register Pressure Responsive Software Pipelining, in Proceedings of the Symposium on Applied Computing SAC 2001, Las Vegas, March 2001, pp. 626–631.

[12] U. Banerjee, S. Shen, D.J. Kuck and R.A. Towle, Time and parallel processor bounds for Fortran-like loops, *IEEE Trans. Comput.*, C28(9), 660–670, September 1979.

[13] S.J. Beaty, Instruction Scheduling Using Genetic Algorithms, Ph.D. thesis, Colorado State University, Fort Collins, CO, 1991.

[14] G.R. Beck, D.W.L. Yen and T.L. Anderson, The Cydra 5 Minicomputer: architecture and implementation, in *J. Supercomput.*, 143–180, May 1993.

[15] D. Berstein and M. Rodeh, Global Instruction Scheduling for Superscalar Machines, in Proceedings of the SIGPLAN 1991 Conference on Programming Language Design and Implementation, PLDI '91, May 1991, pp. 241–255.

[16] M. Breternitz, Jr., Architecture Synthesis of High-Performance Application-Specific Processors, Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA, 1991.

[17] P.-Y. Calland, A. Darate and Y. Robert, Circuit retiming applied to decomposed software pipelining, *IEEE Trans. Parallel Distributed Syst.*, 9(1), 24–35, January 1998.

[18] K.M. Chandy and C. Kesselman, Parallel programming in 2001, *IEEE Software*, 8(6), 11–20, November 1991.

[19] E.S. Davidson, The Design and Control of Pipelined Function Generators, in Proceedings of the 1971 International IEEE Conference on Systems, Networks, and Computers, Oaxtepec, Mexico, January 1971, pp. 19–21.

[20] J.C. Dehnert, P.Y.-T. Hsu and J.P. Bratt, Overlapped Loop Support in the Cydra-5, in Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems, Boston, MA, April 1989, pp. 26–38.

[21] J.C. Dehnert and R.A. Towle, Compiling for the Cydra-5, in *J. Supercomput.*, May 1993, pp. 181–228.

[22] K. Ebcioğlu, A Compilation Technique for Software Pipelining of Loops With Conditional Jumps, in Proceedings of the 20th Microprogramming Workshop (MICRO-20), Colorado Springs, CO, December 1987, pp. 69–79.

[23] K. Ebcioğlu and T. Nakatani, A new compilation technique for parallelizing loops with unpredictable branches on a VLIW architecture, in *Languages and Compilers for Parallel Computing*, D. Gelernter, Ed., MIT Press, Cambridge, MA, 1990, pp. 213– 229.

[24] J. Ferrante, K.J. Ottenstein and J.D. Warren, The program dependence graph and its use in optimization, *ACM Trans. Programming Languages Syst.*, 9(3), 319–349, July 1987.

[25] J. Fisher, Trace scheduling: a technique for global microcode compaction, *IEEE Trans. Comput.*, C-30(7), 478–490, July 1981.

[26] J.A. Fisher, Trace scheduling: a technique for global microcode compaction, *IEEE Trans. Comput.*, C-30(7), 478–490, July 1981.

[27] S.M. Freudenberger, T.R. Gross and P.G. Lowney, Avoidance and suppression of compensation code in a trace scheduler, *ACM Trans. Programming Languages Syst.*, 16(4), 1156–1214, July 1994.

[28] G.R. Gao, W.-B. Wong and Q. Ning, A timed Petri-net model for fine-grain loop scheduling, *SIGPLAN Notices*, 26(6), 204–218, June 1991.

[29] G.R. Gao, W.-B. Wong and Q. Ning, A Timed Petri-Net Model for Fine-Grain Loop Scheduling, Technical report ACAPS Technical Memo 18, School of Computer Science, McGill University, Montreal, Canada, January 1991.

[30] F. Gasperoni and U. Schwiegelsjojn, Generating close to optimum loop schedules on parallel processors, *Parallel Process. Lett.*, 4(4), 391–403, 1994.

[31] R. Govindarajan, E.R. Altman and G.R. Gao, Minimizing Register Requirements under Resource Constrained Rate-Optimal Software Pipelining, in Proceedings of the 27th Annual International Symposium on Microarchitecture, 1994, pp. 85–94.

[32] R. Govindarajan, E.R. Altman and G.R. Gao, A Framework for Resource-Constrained Rate-Optimal Software Pipelining, *IEEE Trans. Parallel Distributed Syst.*, 7(11), 1133–1149, November 1996.

[33] R. Govindarajan, E.R. Altman and G.R. Gao, Co-Scheduling Hardware and Software Pipelines, in Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture (HPCA '96), San Jose, CA, February 1996, pp. 52–61.

[34] P.Y.T. Hsu, Highly Concurrent Scalar Processing. Ph.D. thesis, Department of Computer Science, University of Illinois Urbana-Champaign, Urbana-Champaign, IL, 1986.

[35] P.Y.T. Hsu and E.S. Davidson, Highly concurrent scalar processing, *Comput. Architecture News*, 14(2), 386–395, June 1986.

[36] J. Huck, D. Morris, J. Ross, A. Knies, H. Mulder and R. Zahir, Introducing the IA-64 architecture, *IEEE Micro*, 20(5), 12–23, September-October 2000.

[37] R.A. Huff, Lifetime-Sensitive Modulo Scheduling, in Conference Record of SIGPLAN Programming Language and Design Implementation, Albuquerque, NM, June 1993, pp. 258–267.

[38] J. Ellis, Bulldog: A compiler for VLIW architecture, Ph.D. thesis, Yale University, New Haven, CT, February 1985.

[39] R.B. Jones, Constrained Software Pipelining, Master's thesis, Department of Computer Science, Utah State University, Logan, UT, September 1991.

[40] S. Kim, S.-M. Moon and K. Ebcioğlu, Unroll-Based Register Coalescing, in Proceedings of International Conference on Supercomputing ICS 2000, Sante Fe, NM, May 2000, pp. 296–305.

[41] R. Krishnaiyer, D. Kulkarni, D. Lavery, W. Li, C.-C. Lim, J. Ng and D. Sehr, An advanced optimizer for the IA-64 architecture, *IEEE Micro*, 20(6), 60–68, November–December 2000.

[42] D.J. Kuck and D.A. Padua, High-speed multiprocessors and their compilers, in Proceedings 1979 International Conference on Parallel Processing, 1979, pp. 5–16.

[43] M.S. Lam, A Systolic Array Optimizing Compiler, Ph.D. thesis, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1987.

[44] M.S. Lam, Software Pipelining: An Effective Scheduling Technique for VLIW Machines, in Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation, Atlanta, GA, June 1988, pp. 318–328.

[45] D. López, M. Valero, J. Llosa and E. Ayguadé, Increasing Memory Bandwidth with Wide Buses: Compiler, Hardware, and Performance Trade-offs, in Proceedings of the International Conference on Supercomputing ICS '97, Vienna, Austria, July 1997, pp. 12–19.

[46] S.A. Mahlke, D.C. Lin, W.Y. Chen, R.E. Hank and R.A. Bringmann, Effective Compiler Support for Predicated Execution Using the Hyperblock, in Proceedings of the 25th Annual International Symposium on Microarchitecture (MICRO-25), Portland, OR, December 1992, pp. 45–54.

[47] P. Mateti and N. Deo, On algorithms for enumerating all circuits of a graph, *SIAM J. Computing*, 5(1), 990–99, 1976.

[48] D. Milićev and Z. Jovanović, A Formal Model of Software Pipelining Loops with Conditions, in Proceedings of the 11th International Parallel Processing Symposium IPPS '97, Geneva, Switzerland, April 1997.

[49] S.-M. Moon and K. Ebcioğlu, Parallelizing nonnumerical code with selective scheduling and software pipelining, *ACM Trans. Programming Languages Syst.*, 19(6), 853–898, November 1997.

[50] T. Nakatani and K. Ebcioğlu, "Combining" as a Compilation Technique for VLIW Architectures, in Proceedings of the 22nd Microprogramming Workshop (MICRO-22), ACM Press, Cambridge, MA, 1989, pp. 43–55.

[51] A. Nicolau and R. Potasman, An Environment for the Development of Microcode for Pipelined Architectures, in Proceedings of the 23rd Symposium and Workshop on Microprogramming and Microarchitecture, Orlando, Florida, November 1990, pp. 69–79.

[52] M.R. O'Neill, Software Pipelining with Stochastic Search Algorithms, Master's thesis, Department of Computer Science, Utah State University, Logan, UT, 1994.

[53] P.R. Panda, H. Nakamura, N.D. Dutt and A. Nicolau, Augmenting loop tiling with data alignment for improved cache performance, *IEEE Trans. Comput.*, 48(2), February 1999.

[54] M. Rajagopalan and V.H. Allan, Specification of software pipelining using Petri nets, *Int. J. Parallel Process.*, 3(22), 279–307, 1994.

[55] B.R. Rau, Iterative Modulo Scheduling: An Algorithm for Software Pipelined Loops, in Proceedings of Micro-27, 27th Annual International Symposium on Microarchitecture, San Jose, CA, November 1994, pp. 63–74.

[56] B.R. Rau and J.A. Fisher, Instruction-level parallel processing: history, overview, and perspective, *J. Supercomput.*, 7, 9–50, 1993.

[57] B.R. Rau, M. Lee, P.P. Tirumalai and M.S. Schlansker, Register Allocation for Modulo Scheduled Loops: Strategies, Algorithms, and Heuristics, in Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation, San Francisco, CA, June 1992, pp. 283–299.

[58] B.R. Rau, M.S. Schlansker and P.P. Tirumalai, Code Generation Schema for Modulo Scheduled Loops, in Proceedings of Micro-25, The 25th Annual International Symposium on Microarchitecture, Portland, OR, December 1992, pp. 158–169.

[59] B.R. Rau, D.W.L. Yen, W. Yen and R.A. Towle, The Cydra 5 departmental supercomputer: design philosophies, decisions, and trade-offs, *IEEE Comput.*, January 1989, pp. 12–25.

[60] B.R. Rau and C.D. Glaeser, Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing, in 14th Annual Workshop of Microprogramming, October 1981, pp. 183–198.

[61] B.R. Rau, C.D. Glaeser and E.M. Greenwalt, Architectural Support for the Efficient Generation of Code for Horizontal Architectures, in Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems, March 1982, pp. 96–99.

[62] J. Ruttenberg, G.R. Gao, A. Stoutchinin and W. Lichtenstein, Software Pipelining Showdown: Optimal vs. Heuristic Methods in a Production Compiler, in *SIGPLAN '96 Conference on Programming Languages and Design Implementation*, 31(5), 1–11, May 1996.

[63] F.J. Sánchez and A. Gonzáles, Cache Sensitive Modulo Scheduling, in Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture, Research Triangle Park, December 1997, pp. 338–348.

[64] SM. Shim and S.-M. Moon, Split-Path Enchanced Pipeline Scheduling for Loops with Control Flow, in Proceedings of 31st Annual ACM/IEEE International Symposium on Microarchitecture, Dallas, TX, November 1998, pp. 93–102.

[65] E. Stotzer and E. Leiss, Modulo Scheduling for the TMS320C6X VLIW DSP Architectures, in Proceedings of Language, Compiler and Tool-Support for Embedded Systems, LCTES '99, Atlanta, GA, 1999, pp. 28–34.

[66] B. Su, S. Ding and J. Xia, URPR — An Extension of URCR for Software Pipelining, in Proceedings of the 19th Microprogramming Workshop (MICRO-19), New York, NY, October 1986, pp. 104–108.

[67] R. Tarjan, Depth-first search and linear graph algorithms, *SIAM J. Comput.*, 1(2), 146–160, June 1972.

[68] J.C. Tiernan, An efficient search algorithm to find the elementary circuits of a graph, *Commun. ACM*, 13(1), 722–726, 1970.

[69] P. Tirumalai, M. Lee and M.S. Schlansker, Parallelization of Loops with Exits on Pipelined Architectures, in Proceedings of SuperComputing '90, Amsterdam, November 1990, pp. 200–212.

[70] M. Tokoro, E. Tamura, K. Takase and K. Tamaru, An Approach to Microprogram Optimization Considering Resource Occupancy and Instruction Formats, in Proceedings of the 10th Annual Workshop on Microprogramming, Niagara Falls, NY, November 1977, pp. 92–108.

[71] S. Vegdahl, A Dynamic-Programming Technique for Compacting Loops, in Proceedings of Micro-25, 25th Annual International Symposium on Microarchitecture, Portland, OR, December 1992, pp. 180–188.

[72] J. Wang, C. Einsenbeis, M. Jourdan and B. Su, Decomposed software pipelining, *Int. J. Parallel Programming*, 27(3), 352–373, 1994.

[73] N.J. Warter, G.E. Haab and J.W. Bockhaus, Enhanced Modulo Scheduling for Loops with Conditional Branches, in Proceedings of the 25th Annual International Symposium on Microarchitecture (MICRO-25), Portland, OR, December 1992, pp. 170–179.

[74] N.J. Warter, S.A. Mahlke, W.-M. Hwu and B.R. Rau, Reverse If-Conversion, in Proceedings of the SIGPLAN 1993 Conference on Programming Language Design and Implementation, PLDI '93, Albuquerque, NM, June 1993, pp. 290–299.

[75] M.J. Wolfe, *Optimizing Supercompilers for Supercomputers*, MIT Press, Cambridge, MA, 1989.

[76] M.J. Wolfe, Tiny — A Loop Restructuring Tool, User Manual, Oregon Graduate Institute of Science and Technology, Klamuth Falls, OR, 1990.

[77] A.M. Zaky, Efficient Static Scheduling of Loops on Synchronous Multiprocessors, Ph.D. thesis, Department of Computer and Information Science, Ohio State University, Columbus, OH, 1989.

[78] J. Zalamea, J. Llosa, E. Ayguadé and M. Valero, Improved spill code generation for software pipelined loops, *ACM SIGPLAN Notices, Proceedings of the ACM SIGPLAN '00 Conference on Programming Language, Design and Implementation*, 35(5), 134–144, May 2000.

[79] H. Zima and B. Chapman, *Supercompilers for Parallel and Vector Computers*, ACM Press, Cambridge, MA, 1991.

19
Dynamic Compilation

Evelyn Duesterwald
Hewlett-Packard Laboratories

19.1 Introduction

The term *dynamic compilation* refers to techniques for runtime generation of executable code. The idea of compiling parts or all the application code while the program is executing challenges our intuition about overheads involved in such an endeavor. Yet, recently a number of different approaches have evolved that effectively manage this challenging task.

The ability to dynamically adapt executing code addresses many of the existing problems with traditional static compilation approaches. One such problem is the difficulty for a static compiler to fully exploit the performance potential of advanced architectures. In the drive for greater performance, today's microprocessors provide capabilities for the compiler to take on a greater role in performance delivery, ranging from predicated and speculative execution (e.g., for the Intel Itanium processor) to various power consumption control models. To exploit these architectural features, the static compiler usually has to rely on profile information about the dynamic execution behavior of a program. However, collecting valid execution profiles ahead of time may not always be feasible or practical. Moreover, the risk of performance degradation that may result from missing or outdated profile information is high.

Current trends in software technology create additional obstacles to static compilation. These are exemplified by the widespread use of object-oriented programming languages and the trend toward shipping software binaries as collections of dynamically linked libraries, instead of monolithic binaries. Unfortunately, the increased degree of runtime binding can seriously limit the effectiveness of traditional static compiler optimization, because static compilers operate on the statically bound scope of the program.

Finally, the emerging Internet and mobile communications marketplace create the need for the compiler to produce portable code that can efficiently execute on a variety of machines. In an environment of networked devices, where code can be downloaded and executed on the fly, static

0-8493-1240-7/03/$0.00+$1.50

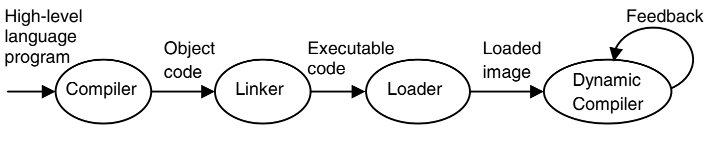


FIGURE 19.1 Dynamic compilation pipeline.

compilation at the target device is usually not an option. However, if static compilers can only be used to generate platform-independent intermediate code, their role as a performance delivery vehicle becomes questionable.

This chapter discusses dynamic compilation, a radically different approach to compilation that addresses and overcomes many of the preceding challenges to effective software implementation. Dynamic compilation extends our traditional notion of compilation and code generation by adding a new dynamic stage to the classical pipeline of compiling, linking and loading code. The extended dynamic compilation pipeline is depicted in Figure 19.1.

A dynamic compiler can take advantage of runtime information to exploit optimization opportunities not available to a static compiler. For example, it can customize the running program according to information about actual program values or actual control flow. Optimization may be performed across dynamic binding, such as optimization across dynamically linked libraries. Dynamic compilation avoids the limitations of profile-based approaches by directly utilizing runtime information. Furthermore, with a dynamic compiler, the same code region can be optimized multiple times should its execution environment change. Another unique opportunity of dynamic compilation is the potential to speedup the execution of legacy code that was produced using outdated compilation and optimization technology.

Dynamic compilation provides an important vehicle to efficiently implement the "write-once-run-anywhere" execution paradigm that has recently gained a lot of popularity with the Java programming language [23]. In this paradigm, the code image is encoded in a mobile platform-independent format (e.g., Java bytecode). Final code generation that produces native code takes place at runtime, as part of the dynamic compilation stage.

In addition to addressing static compilation obstacles, the presence of a dynamic compilation stage can create entirely new opportunities that go beyond code compilation. Dynamic compilation can be used to transparently migrate software from one architecture to a different host architecture. Such a translation is achieved by dynamically retargeting the loaded nonnative guest image to the host machine native format. Even for machines within the same architectural family, a dynamic compiler may be used to upgrade software to exploit additional features of the newer generation.

As indicated in Figure 19.1, the dynamic compilation stage may also optionally include a feedback loop. With such a feedback loop, dynamic information, including the dynamically compiled code itself, may be saved at runtime to be restored and utilized in future runs of the program. For example, the FX!32 system for emulating x86 code on an Alpha platform [27] saves runtime information about executed code, which is then used to produce translations off-line that can be incorporated in future runs of the program. It should be noted that FX!32 is not strictly a dynamic compilation system, in that translations are produced in between executions of the program instead of on-line during execution.

Along with its numerous opportunities, dynamic compilation also introduces a unique set of challenges. One such challenge is to amortize the dynamic compilation overhead. If dynamic compilation is sequentially interleaved with program execution, the dynamic compilation time directly contributes to the overall execution time of the program. Such interleaving greatly changes the cost-benefit compilation trade-off that we have grown accustomed to in static compilation. Although in a static compiler increased optimization effort usually results in higher performance, increasing the dynamic compilation time may actually diminish some or all the performance improvements that were gained by the optimization in the first place. If dynamic compilation takes place in parallel with program execution on a multiprocessor system, the dynamic compilation overhead is less important, because the dynamic compiler cannot directly slow down the program. It does, however, divert resources that could have been devoted toward execution. Moreover, long dynamic compilation times can still adversely affect performance. Spending too much time on compilation can delay the employment of the dynamically compiled code and diminish the benefits. To maximize the benefits, dynamic compilation time should therefore always be kept to a minimum.

To address the heightened pressure for minimizing overhead, dynamic compilers often follow an adaptive approach [24]. Initially, the code is optimized with little or no optimization. Aggressive optimizations are considered only later, when more evidence has been found that added optimization effort is likely to be of use.

A dynamic compilation stage, if not designed carefully, can also significantly increase the space requirement for running a program. Controlling additional space requirements is crucial in environments where code size is important, such as embedded or mobile systems. The total space requirements of execution with a dynamic compiler include not only the loaded input image but also the dynamic compiler itself, plus the dynamically compiled code. Thus, care must be taken to control both the footprint of the dynamic compiler and the size of the currently maintained dynamically compiled code.

19.2 Approaches to Dynamic Compilation

A number of different approaches to dynamic compilation have been developed. These approaches differ in several aspects, including the degree of transparency, the extent and scope of dynamic compilation and the assumed encoding format of the loaded image. On the highest level, dynamic compilation systems can be divided into transparent and nontransparent systems. In a transparent system, the remainder of the compilation pipeline is oblivious to the fact that a dynamic compilation stage has been added. The executable produced by the linker and loader is not specially prepared for dynamic optimization and it may execute with or without a dynamic compilation stage. Figure 19.2 shows a classification of the various approaches to transparent and nontransparent dynamic compilation.

Transparent dynamic compilation systems can further be divided into systems that operate on binary executable code (binary dynamic compilation) and systems that operate on an intermediate platform-independent encoding (just-in-time [JIT] compilation). A binary dynamic compiler starts out with a loaded fully executable binary. In one scenario, the binary dynamic compiler recompiles the binary code to incorporate native-to-native optimizing transformations. These recompilation systems are also referred to as *dynamic optimizers* [3, 5, 7, 15, 36]. During recompilation, the binary is optimized by customizing the code with respect to specific runtime control and data flow values. In dynamic binary translation, the loaded input binary is in a nonnative format, and dynamic compilation is used to retarget the code to a different host architecture [19, 35, 39]. The dynamic code translation may also include optimization.

JIT compilers present a different class of transparent dynamic compilers [11, 12, 18, 28, 29]. The input to a JIT compiler is not a native program binary; instead, it is code in an intermediate, platform-independent representation that targets a virtual machine. The JIT compiler serves as an enhancement to the virtual machine to produce native code by compiling the intermediate input program at runtime, instead of executing it in an interpreter. Typically, semantic information is attached to the code, such as symbol tables or constant pools, which facilitates the compilation.

The alternative to transparent dynamic compilation is the nontransparent approach, which integrates the dynamic compilation stage explicitly within the earlier compilation stages. The static compiler cooperates with the dynamic compiler by delaying certain parts of the compilation to

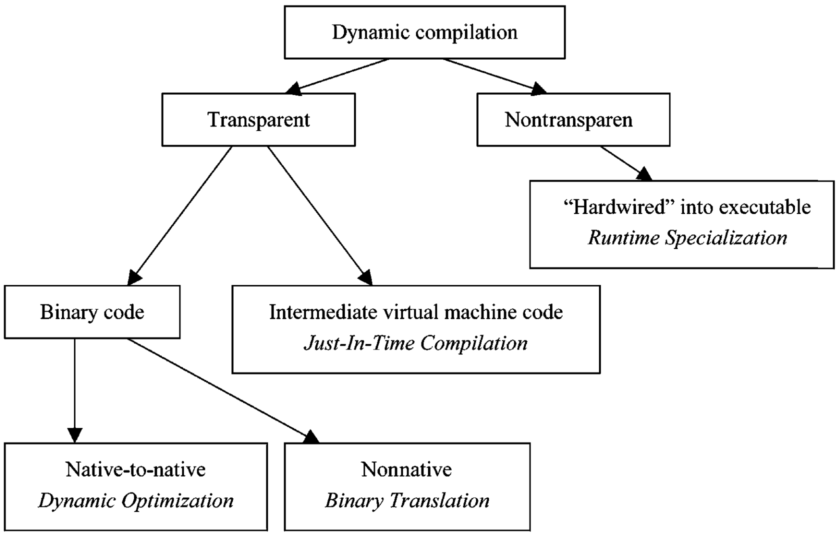


FIGURE 19.2 Software dynamic compilation classification.

runtime, if their compilation can benefit from runtime values. A dynamic compilation agent is compiled (i.e., hardwired) into the executable to fill and link in a prepared code template for the delayed compilation region. Typically, the programmer indicates adequate candidate regions for dynamic compilation via annotations or compiler directives. Several techniques have been developed to perform runtime specialization of a program in this manner [9, 24, 31, 33].

Runtime specialization techniques are tightly integrated with the static compiler, whereas transparent dynamic compilation techniques are generally independent of the static compiler. However, transparent dynamic compilation can still benefit from information that the static compiler passes down. Semantic information, such as a symbol table, is an example of compiler information that is beneficial for dynamic compilation. If the static compiler is made aware of the dynamic compilation stage, more targeted information may be communicated to the dynamic compiler in the form of code annotations to the binary [30].

The remainder of this chapter discusses the various dynamic compilation approaches shown in Figure 19.2. We first discuss transparent binary dynamic optimization as a representative dynamic compilation system. We discuss the mechanics of dynamic optimization systems and their major components, along with their specific opportunities and challenges. We then discuss systems in each of the remaining dynamic compilation classes and point out their unique characteristics.

Also a number of hardware approaches are available to dynamically manipulate the code of a running program, such as the hardware in out-of-order superscalar processors or hardware dynamic optimization in trace cache processors [21]. However, in this chapter, we limit the discussion to software dynamic compilation.

19.3 Transparent Binary Dynamic Optimization

A number of binary dynamic compilation systems have been developed that operate as an optional dynamic stage [3, 5, 7, 15, 35]. An important characteristic of these systems is that they take full control over the execution of the program. Recall that in the transparent approach, the input program is not specially prepared for dynamic compilation. Therefore, if the dynamic compiler does not maintain full control over the execution, the program may escape and simply continue executing natively, effectively bypassing dynamic compilation altogether. The dynamic compiler can afford to relinquish control only if it can guarantee to regain control later, for example, via a timer interrupt.

Binary dynamic compilation systems share the general architecture shown in Figure 19.3. Input to the dynamic compiler is the loaded application image as produced by the compiler and linker. Two main components of a dynamic compiler are the compiled code cache that holds the dynamically compiled code fragments and the dynamic compilation engine. At any point in time, execution takes place either in the dynamic compilation engine or in the compiled code cache. Correspondingly, the dynamic compilation engine maintains two distinct execution contexts: the context of the dynamic compilation engine itself and the context of the application code.

Execution of the loaded image starts under control of the dynamic compilation engine. The dynamic compiler determines the address of the next instruction to execute. It then consults a lookup table to determine whether a dynamically compiled code fragment starting at that address already exists in the code cache. If so, a context switch is performed to load the application context, and to continue execution in the compiled code cache until a code cache miss occurs. A code cache miss indicates that no compiled fragment exists for the next instruction. The cache miss triggers a context switch to reload the dynamic compiler's context and reenter the dynamic compilation engine.

The dynamic compiler decides whether a new fragment should be compiled starting at the next address. If so, a code fragment is constructed based on certain fragment selection policies, which are discussed in the next section. The fragment may optionally be optimized and linked with other previously compiled fragments before it is emitted into the compiled code cache.

The dynamic compilation engine may include an instruction interpreter component. With an interpreter component, the dynamic compiler can choose to delay the compilation of a fragment and instead interpret the code until it has executed a number of times. During interpretation, the dynamic compiler can profile the code to focus its compilation efforts on only the most profitable code fragments [4]. Without an interpreter, every portion of the program that is executed during the current run can be compiled into the compiled code cache.

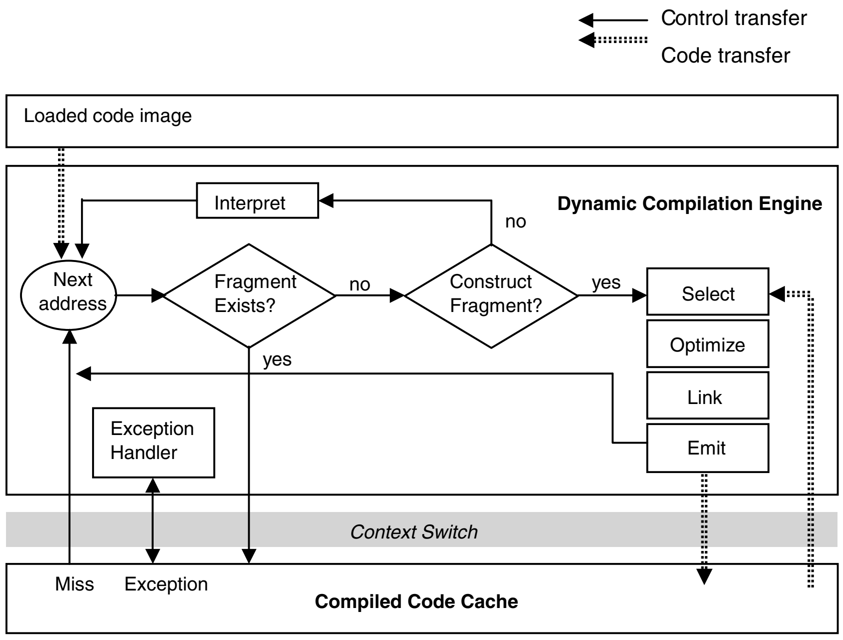


FIGURE 19.3 Transparent dynamic compilation architecture.

Figure 19.3 shows a code transfer arrow from the compiled code cache to the fragment selection component. This arrow indicates that the dynamic compiler may choose to select new code fragments from previously created code in the compiled code cache. Such fragment reformation may be performed to improve fragment shape and extent. For example, several existing code fragments may be combined to form a single new fragment. The dynamic compiler may also reselect an existing fragment for more aggressive optimization. Reoptimization of a fragment may be indicated if profiling of the compiled code reveals that it is a hot (i.e., frequently executing) fragment.

In the following sections, we discuss the major components of the dynamic compiler in detail: fragment selection, fragment optimization, fragment linking, management of the compiled code cache and exception handling.

19.3.1 Fragment Selection

The fragment selector proceeds by extracting code regions and passing them to the fragment optimizer for optimization and eventual placement in the compiled code cache. The arrangement of the extracted code regions in the compiled code cache leads to a new code layout, which has the potential of improving the performance of dynamically compiled code. Furthermore, by passing isolated code regions to the optimizer, the fragment selector dictates the scope and kind of runtime optimization that may be performed. Thus, the goal of fragment selection is twofold: to produce an improved code layout and to expose dynamic optimization opportunities.

New optimization opportunities or improvements in code layout are unlikely if the fragment selector merely copies static regions from the loaded image into the code cache. Regions such as basic blocks or entire procedures are among the static regions of the original program and have been already exposed to, and possibly optimized by, the static compiler. New optimization opportunities are more likely to be found in the dynamic scope of the executing program. Thus, it is crucial to incorporate dynamic control flow into the selected code regions.

Because of the availability of dynamic information, the fragment selector has an advantage over a static compiler in selecting the most beneficial regions to optimize. At the same time, the fragment selector is more limited because high-level semantic information about code constructs is no longer available. For example, without information about procedure boundaries and the layout of switch statements, it is generally impossible to discover the complete control flow of a procedure body in a loaded binary image.

In the presence of these limitations, the units of code commonly used in a binary dynamic compilation system is a partial execution trace, or trace for short BD [4, 7]. A trace is a dynamic sequence of consecutively executing basic blocks. The sequence may not be contiguous in memory; it may even be interprocedural, spanning several procedure boundaries, including dynamically linked modules. Thus, traces are likely to offer opportunities for improved code layout and optimization. Furthermore, traces do not need to be computed; they can be inferred simply by observing the runtime behavior of the program.

Figure 19.4 illustrates the effects of selecting dynamic execution traces. The graph in Figure 19.4(a) shows a control flow graph representation of a trace, consisting of blocks A, B, C, D and E that forms a loop containing a procedure call. The graph in Figure 19.4(b) shows the same trace in a possible noncontiguous memory layout of the original loaded program image. The graph in Figure 19.4(c) shows a possible improved layout of the looping trace in the compiled code cache as a contiguous straight-line sequence of blocks. The straight-line layout reduces branching during execution and offers better code locality for the loop.

19.3.1.1 Adaptive Fragment Selection

The dynamic compiler may select fragments of varying shapes. It may also stage the fragment selection in a progressive fashion. For example, the fragment selector may initially select only basic

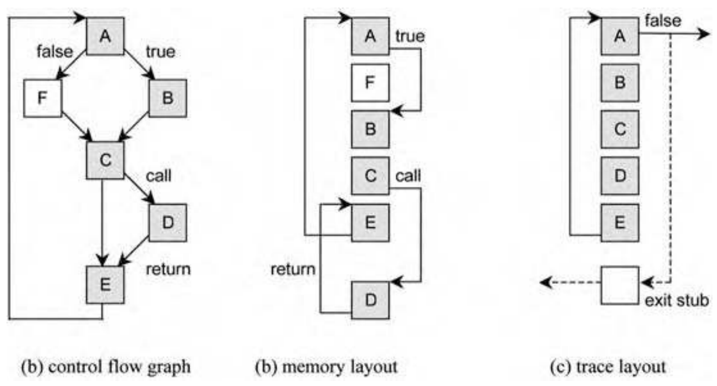


FIGURE 19.4 Traces in the control flow graph and in memory layout.

block fragments. Larger composite fragments, such as traces, are selected as secondary fragments by stringing together frequently executing block fragments [4]. Progressively larger regions, such as tree regions, may then be constructed by combining individual traces [19]. Building composite code regions can result in potentially large amounts of code duplication because code that is common across several composite regions is replicated in each region. Uncontrolled code duplication can quickly result in excessive cache size requirements, the so-called *code explosion* problem. Thus, a dynamic compiler has to employ some form of execution profiling to limit composite region construction to only the (potentially) most profitable candidates.

19.3.1.2 On-line Profiling

Profiling the execution behavior of the loaded code image to identify the most frequently executing regions is an integral part of dynamic compilation. Information about the hot spots in the code is used in fragment selection and for managing the compiled code cache space. Hot spots must be detected on-line as they are becoming hot, which is in contrast to conventional profiling techniques that operate off-line and do not establish relative execution frequencies until after execution. Furthermore, to be of use in a dynamic compiler, the profiling techniques must have very low space and time overheads.

A number of off-line profiling techniques have been developed for use in feedback systems, such as profile-based optimization. A separate profile run of the program is conducted to accumulate profile information that is then fed back to the compiler. Two major approaches to off-line profiling are statistical PC sampling and binary instrumentation for the purpose of branch or path profiling. Statistical PC sampling [1, 10, 40] is an inexpensive technique for identifying hot code blocks by recording program counter hits. Although PC sampling is efficient for detecting individual hot blocks, it provides little help in finding larger hot code regions. One could construct a hot trace by stringing together the hottest code blocks. However, such a trace may never execute from start to finish because the individual blocks may have been hot along disjoint execution paths. The problem is that individually collected branch frequencies do not account for branch correlations, which occur if the outcome of one branch can influence the outcome of a subsequent branch.

Another problem with statistical PC sampling is that it introduces nondeterminism into the dynamic compilation process. Nondeterministic behavior is undesirable because it greatly complicates development and debugging of the dynamic compiler.

Profiling techniques based on binary instrumentation record information at every execution instance. They are more costly than statistical sampling, but can also provide more fine-grained frequency information. Like statistical sampling, branch profiling techniques suffer the same problem of not adequately addressing branch correlations. Path-profiling techniques overcome the correlation problem by directly determining hot traces in the program [6]. The program binary is instrumented to collect entire path (i.e., trace) frequency information at runtime in an efficient manner.

A dynamic compiler could adopt these techniques by inserting instrumentation in first-level code fragments to build larger composite secondary fragments. The drawback of adapting off-line techniques is the large amount of profile information that is collected and the overhead required to process it. Existing dynamic compilation systems have employed more efficient, but also more approximate, profiling schemes that collect a small amount of profiling information, either during interpretation [5] or by instrumenting first-level fragments [19]. Ephemeral instrumentation is a hybrid profiling technique [38] based on the ability to efficiently enable and disable instrumentation code.

19.3.1.3 On-line Profiling in the Dynamo System

As an example of a profiling scheme used in a dynamic compiler, we consider the next executing tail (NET) scheme used in the Dynamo system [17]. The objective of the NET scheme is to significantly reduce profiling overhead while still providing effective hot path predictions. A path is divided into a path head (i.e., the path starting point, and the path tail, which is the remainder of the path following the starting point. For example, in path ABCDE in Figure 19.4(a), block A is the path head and BCDE is the path tail. The NET scheme reduces profiling cost by using speculation to predict path tails, while maintaining full profiling support to predict hot path heads. The rationale behind this scheme is that a hot path head indicates that the program is currently executing in a hot region, and the next executing path tail is likely to be part of that region.

Accordingly, execution counts are maintained only for potential path heads, which are the targets of backward taken branches or the targets of cache exiting branches. For example, in Figure 19.4(a), one profiling count is maintained for the entire loop at the single path head at the start of block A. Once the counter at block A has exceeded a certain threshold, the next executing path is selected as the hot path for the loop.

19.3.2 Fragment Optimization

After a fragment has been selected, it is translated into a self-contained location-independent intermediate representation (IR). The IR of a fragment serves as a temporary vehicle to transform the original instruction stream into an optimized form and to prepare it for placement and layout in the compiled code cache. To enable fast translation between the binary code and the IR, the abstraction level of the IR is kept close to the binary instruction level. Abstraction is introduced only when needed, such as to provide location independence through symbolic labels and to facilitate code motion and code transformations through the use of virtual registers.

After the fragment is translated into its intermediate form, it can be passed to the optimizer. A dynamic optimizer is not intended to duplicate or replace conventional static compiler optimization. On the contrary, a dynamic optimizer can complement a static compiler by exploiting optimization opportunities that present themselves only at runtime, such as value-based optimization or optimization across the boundaries of dynamically linked libraries. The dynamic optimizer can also apply path-specific optimization that would be too expensive to apply indiscriminately over all paths during static compilation. On a given path, any number of standard compiler optimizations may be performed, such as constant and copy propagation, dead code elimination, value numbering and redundancy elimination [4, 15]. However, unlike in static compiler optimization, the optimization algorithm must be optimized for efficiency instead of generality and power. A traditional static optimizer performs an initial analysis phase over the code to collect all necessary data flow information that is followed by the actually optimization phase. The cost of performing multiple passes over the code is likely to be prohibitive in a runtime setting. Thus, a dynamic optimizer typically combines analysis and optimization into a single pass over the code [4]. During the combined pass all necessary

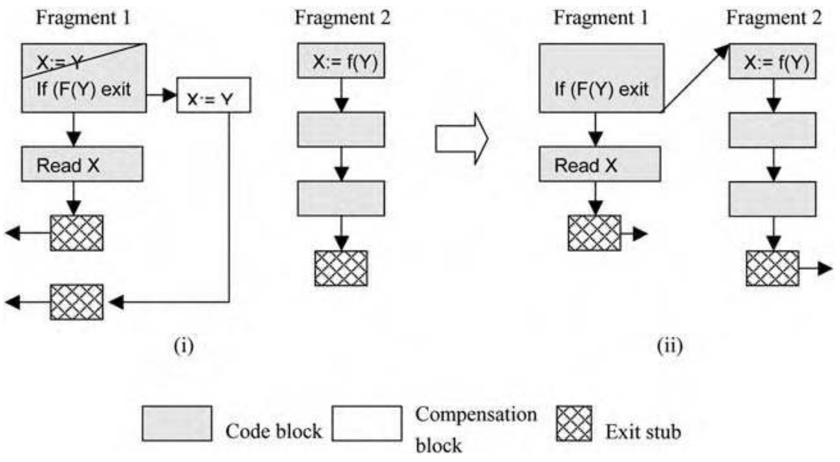


FIGURE 19.5 Code sinking example. Code sinking before (i) and after (ii) fragment linking.

data flow information is gathered on demand and discarded immediately if it is no longer relevant for current optimization [16].

19.3.2.1 Control Specialization

The dynamic compiler implicitly performs a form of control specialization of the code by producing a new layout of the running program inside the compiled code cache. Control specialization describes optimizations whose benefits are based on the execution taking specific control paths. Another example of control specialization is code sinking [4], also referred to as hot–cold optimization [13]. The objective of code sinking is to move instructions from the main fragment execution path into fragment exits to reduce the number of instructions executed on the path. An instruction can be sunk into a fragment exit block if it is not live within the fragment. Although an instruction appears dead on the fragment, it cannot be removed entirely because it is not known whether it is also dead after exiting the fragment.

An example of code sinking is illustrated in Figure 19.5. The assignment X: = Y in the first block in fragment 1 is not live within fragment 1 because it is overwritten by the read instruction in the next block. To avoid useless execution of the assignment when control remains within fragment 1, the assignment can be moved out of the fragment and into a so-called compensation block at every fragment exit at which the assigned variable may still be live, as shown in Figure 19.5(i). Once the exit block is linked to a target fragment (fragment 2 in Figure 19.5) the code inside the target fragment can be inspected to determine whether the moved assignment becomes dead after linking. If it does, the moved assignment in the compensation block can safely be removed, as shown in Figure 19.5(ii).

Another optimization is prefetching, which involves the placement of prefetch instructions along execution paths prior to the actual usage of the respective data to improve the memory behavior of the dynamically compiled code. If the dynamic compiler can monitor data cache latency, it can easily identify candidates for prefetching. A suitable placement of the corresponding prefetch instructions can be determined by consulting collected profile information.

19.3.2.2 Value Specialization

Value specialization refers to an optimization that customizes the code according to specific runtime values of selected specialization variables. The specialization of a code fragment proceeds like a general form of constant propagation and attempts to simplify the code as much as possible.

Unless it can be established for certain that the specialization variable is always constant, the execution of the specialized code must be guarded by a runtime test. To handle specialization variables that take on multiple values at runtime, the same region of code may be specialized multiple times. Several techniques, such as polymorphic in-line caches [25], have been developed to efficiently select among multiple specialization versions at runtime.

A number of runtime techniques have been developed that automatically specialize code at runtime given a specification of the specialization variables [9, 24, 31]. In code generated from object-oriented languages, virtual method calls can be specialized for a common receiver class [25]. In principle, any code region can be specialized with respect to any number of values. For example, traces may be specialized according to the entry values of certain registers. In the most extreme case, one can specialize individual instructions, such as complex floating point instructions, with respect to selected fixed input register values [34].

The major challenge in value specialization is to decide when and what to specialize. Overspecialization of the code can quickly result in code explosion, and may severely degrade performance. In techniques that specialize entire functions, the programmer typically indicates the functions to specialize through code annotations prior to execution [9, 24]. Once the specialization regions are determined, the dynamic specializer monitors the respective register values at runtime to trigger the specialization. Runtime specialization is the primary optimization technique employed by nontransparent dynamic compilation systems. We revisit runtime specialization in the context of nontransparent dynamic compilation in Section 19.6.

19.3.2.3 Binary Optimization

The tasks of code optimization and transformation are complicated by having to operate on executable binary code instead of a higher level intermediate format. The input code to the dynamic optimizer has previously been exposed to register allocation and possibly also to static optimization. Valuable semantic information that is usually incorporated into compilation and optimization, such as type information and information about high-level constructs (i.e., data structures), is no longer available and is generally difficult to reconstruct.

An example of an optimization that is relatively easy to perform on intermediate code but difficult on the binary level is procedure inlining. To completely inline a procedure body, the dynamic compiler has to reverse engineer the implemented calling convention and stack frame layout. Doing this may be difficult, if not impossible, in the presence of memory references that cannot be disambiguated from stack frame references. Thus, the dynamic optimizer may not be able to recognize and entirely eliminate instructions for stack frame allocation and deallocation or instructions that implement caller and callee register saves and restores.

The limitations that result from operating on binary code can be partially lifted by making certain assumptions about compiler conventions. For example, assumptions about certain calling or register usage conventions help in the procedure inlining problem. Also, if it can be assumed that the stack is only accessed via a dedicated stack pointer register, stack references can be disambiguated from other memory references. Enhanced memory disambiguation may then in turn enable more aggressive optimization.

19.3.3 Fragment Linking

Fragment linking is the mechanism by which control is transferred among fragments without exiting the compiled code cache. An important performance benefit of linking is the elimination of unnecessary context switches that are needed to exit and reenter the code cache.

The fragment-linking mechanism may be implemented via exit stubs that are initially inserted at every fragment exiting branch, as illustrated in Figure 19.6. Prior to linking, the exit stubs direct control to the context switch routine to transfer control back to the dynamic compilation engine.

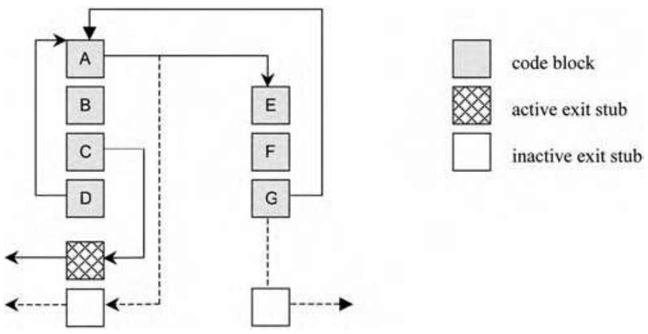


FIGURE 19.6 Fragment linking.

If a target fragment for the original exit branch already exists in the code cache, the dynamic compiler can patch the exiting branch to jump directly to its target inside the cache. For example, in Figure 19.6, the branches A to E and G to A have been directly linked, leaving their original exit stubs inactive. To patch exiting branches, some information about the branch must be communicated to the dynamic compiler. For example, to determine the target fragment of a link, the dynamic compiler must know the original target address of the exiting branch. This kind of branch information may be stored in a link record data structure, and a pointer to it can be embedded in the exit stub associated with the branch [4].

The linking of an indirect computed branch is more complicated. If the fragment selector has collected a preferred target for the indirect branch, it can be inlined directly into the fragment code. The indirect target is inlined by converting the indirect branch into a conditional branch that tests whether the current target is equal to the preferred target. If the test succeeds, control falls through to the preferred target inside the fragment. Otherwise, control can be directed to a special lookup routine that is permanently resident in the compiled code cache. This routine implements a lookup to determine whether a fragment for the indirect branch target is currently resident in the cache. If so, control can be directed to the target fragment without having to exit the code cache [4].

Although its advantages are obvious, linking also has some disadvantages that need to be kept in balance when designing the linker. For example, linking complicates the effective management of the code cache, which may require the periodic removal or relocation of individual fragments. The removal of a fragment may be necessary to make room for new fragments, and fragment relocation may be needed to periodically defragment the code cache. Linking complicates both the removal and relocation of individual fragments because all incoming fragment links have to be unlinked first. Another problem with linking is that it makes it more difficult to limit the latency of asynchronous exception handling. Asynchronous exceptions arise from events such as keyboard interrupts and timer expiration. Exception handling is discussed in more detail in Section 19.3.5.

Linking may be performed on either a demand basis or a preemptive basis. With on-demand linking, fragments are initially placed in the cache with all their exiting branches targeting an exit stub. Individual links are inserted as needed each time control exits the compiled code cache via an exit stub. With preemptive linking, all possible links are established when a fragment is first placed in the code cache. Preemptive linking may result in unnecessary work when links are introduced that are never executed. On the other hand, demand-based linking causes additional context switches and interruptions of cache execution each time a delayed link is established.

19.3.4 Code Cache Management

The code cache holds the dynamically compiled code and may be organized as one large contiguous area of memory, or it may be divided into a set of smaller partitions.

Managing the cache space is a crucial task in the dynamic compilation system. Space consumption is primarily controlled by a cache allocation and deallocation strategy. However, it can also be influenced by the fragment selection strategy. Cache space requirements increase with the amount of code duplication among the fragments. In the most conservative case, the dynamic compiler selects only basic block fragments, which avoids code duplication altogether. However, the code quality and layout in the cache is likely to be unimproved over the original binary. A dynamic compiler may use adaptive strategy that permits unlimited duplication if sufficient space is available but moves toward shorter, more conservatively selected fragments as the available space in the cache diminishes. Even with an adaptive strategy, the cache may eventually run out of space and the deallocation of code fragments may be necessary to make room for future fragments.

19.3.4.1 Fragment Deallocation

A fragment deallocation strategy is characterized by three parameters: the granularity, the timing and the replacement policy that triggers deallocation. The granularity of fragment deallocation may range from an individual fragment deallocation to an entire cache flush. Various performance tradeoffs are to be considered in choosing the deallocation granularity. Individual fragment deallocation is costly in the presence of linking because each fragment exit and entry has to be individually unlinked. To reduce the frequency of cache management events, one might choose to deallocate a group of fragments at a time. A complete flush of one of the cache partitions is considerably cheaper because individual exit and entry links do not have to be processed. Moreover, complete flushing does not incur fragmentation problems. However, uncontrolled flushing may result in loss of useful code fragments that may be costly to reacquire.

The timing of a deallocation can be demand or preemptive based. A demand-based deallocation occurs simply in reaction to an out-of-space condition of the cache space. A preemptive strategy is used in the Dynamo system for cache flushing [4]. The idea is to time a cache flush so that the likelihood of losing valuable cache contents is minimized. The Dynamo system triggers a cache flush when it detects a phase change in the program behavior. When a new program phase is entered, a new working set of fragments is built, and it is likely that most of the previously active code fragments are no longer relevant. Dynamo predicts phase changes by monitoring the fragment creation rate. A phase change is signaled if a sudden increase in the creation rate is detected.

Finally, the cache manager has to implement a replacement policy. A replacement policy is particularly important if individual fragments are deallocated. However, even if an entire cache partition is flushed, a decision has to be made as to which partition to free. The cache manager can borrow simple common replacement policies from memory paging systems, such as first-in, first-out (FIFO) or least recently used (LRU). Alternatively, more advanced garbage collection strategies, such as generational garbage collection strategies, can be adopted to manage the dynamic compilation cache.

Beside space allocation and deallocation, an important code cache service is the fast lookup of fragments that are currently resident in the code cache. Fragment lookups are needed throughout the dynamic compilation system and even during the execution of cached code fragments when it is necessary to look up an indirect branch target. Thus, fast implementation of fragment lookups, for example, via hash tables, is crucial.

19.3.4.2 Multiple Threads

The presence of multithreading can greatly complicate the cache manager. Most of the complication from multithreading can simply be avoided by using thread-private caches. With thread-private caches, each thread uses its own compiled code cache and no dynamically compiled code is shared among threads. However, the lack of code sharing with thread-private caches has several disadvantages. The total code cache size requirements are increased by the need to replicate thread-shared code in each private cache. Beside additional space requirements, the lack of code sharing can also cause redundant work to be carried out when the same thread-shared code is repeatedly compiled.

To implement shared code caches, every code cache access that deletes or adds fragment code must be synchronized. Operating systems usually provide support for thread synchronization. To what extent threads actually share code and, correspondingly, to what extent shared code caches are beneficial are highly dependent on the application behavior.

Another requirement for handling multiple threads is the provision of thread-private state. Storage for thread-private state is needed for various tasks in the dynamic compiler. For example, during fragment selection a buffer is needed to hold the currently collected fragment code. This buffer must be thread private to avoid corrupting the fragment because multiple threads may be simultaneously in the process of creating fragments.

19.3.5 Handling Exceptions

The occurrence of exceptions while executing in the compiled code cache creates a difficult issue for a dynamic compiler. This is true for both user-level exceptions, such as defined in the Java language, and system-level exceptions, such as memory faults. An exception has to be serviced as if the original program is executing natively. To ensure proper exception handling, the dynamic compiler has to intercept all exceptions delivered to the program. Otherwise, the appropriate exception handler may be directly invoked and the dynamic compiler may lose control over the program. Losing control implies that the program has escaped and can run natively for the remainder of the execution.

The original program may have installed an exception handler that examines or even modifies the execution state passed to it. In binary dynamic compilation, the execution state includes the contents of machine registers and the program counter. In JIT compilation, the execution state depends on the underlying virtual machine. For example, in Java, the execution state includes the contents of the Java runtime stack.

If an exception is raised when control is inside the compiled code cache, the execution state may not correspond to any valid state in the original program. The exception handler may fail or operate inadequately when an execution state has been passed to it that was in some way modified through dynamic compilation. The situation is further complicated if the dynamic compiler has performed optimizations on the dynamically compiled code.

Exceptions can be classified as asynchronous or synchronous. Synchronous exceptions are associated with a specific faulting instruction and must be handled immediately before execution can proceed. Examples of synchronous exceptions are memory or hardware faults. Asynchronous exceptions do not require immediate handling, and their processing can be delayed. Examples of asynchronous exceptions include external interrupts (e.g., keyboard interrupts) and timer expiration.

A dynamic compiler can deal with asynchronous exceptions by delaying their handling until a safe execution point is reached. A safe point describes a state at which the precise execution state of the original program is known. In the absence of dynamic code optimization, a safe point is usually reached when control is inside the dynamic compilation engine. When control exits the code cache, the original execution state is saved by the context switch routine prior to reentering the dynamic compilation engine. Thus, the saved context state can be restored before executing the exception handler.

If control resides inside the code cache at the time of the exception, the dynamic compiler can delay handling the exception until the next code cache exit. Because the handling of the exception must not be delayed indefinitely, the dynamic compiler may have to force a code cache exit. To force a cache exit, the fragment that has control at the time of the exception is identified, and all its exit branches are unlinked. Unlinking the exit branches prevents control from spinning within the code cache for an arbitrarily long period of time before the dynamic compiler can process the pending exception.

19.3.5.1 Deoptimization

Unfortunately, postponing the handling of an exception until a safe point is reached is not an option for synchronous exceptions. Synchronous exceptions must be handled immediately, even if control is at a point in the compiled code cache. The original execution state must be recovered as if the original program had executed unmodified. Thus, at the very least, the program counter address, currently a cache address, has to be set to its corresponding address in the original code image.

The situation is more complicated if the dynamic compiler has applied optimizations that change the execution state. This includes optimizations that eliminate code, remap registers or reorder instructions. In Java JIT compilation, this also includes the promotion of Java stack locations to machine registers. To reestablish the original execution state, the fragment code has to be deoptimized. This problem of deoptimization is similar to one that arises with debugging optimized code, where the original unoptimized user state has to be presented to the programmer when a break point is reached.

Deoptimization techniques for runtime compilation have previously been discussed for JIT compilation [26] and binary translation [22]. Each optimization requires its own deoptimization strategy, and not all optimizations are deoptimizable. For example, the reordering of two memory load operations cannot be undone once the reordered earlier load has executed and raised an exception. To deoptimize a transformation, such as dead code elimination, several approaches can be followed. The dynamic compiler can store sufficient information at every optimization point in the dynamically compiled code. When an exception arises, the stored information is consulted to determine the compensation code that is needed to undo the optimization and reproduce the original execution state. For dead code elimination, the compensation code may be as simple as executing the eliminated instruction. Although this approach enables fast state recovery at exception time, it can require substantial storage for deoptimization information.

An alternative approach, which is better suited if exceptions are rare events, is to retrieve the necessary deoptimization information by recompiling the fragment at exception time. During the initial dynamic compilation of a fragment, no deoptimization information is stored. This information is recorded only during a recompilation that takes place in response to an exception.

It may not always be feasible to determine and store appropriate deoptimization information, for example, for optimizations that exploit specific register values. To be exception-safe and to faithfully reproduce original program behavior, a dynamic compiler may have to suppress optimizations that cannot be deoptimized were an exception to arise.

19.3.6 Challenges

The previous sections have discussed some of the challenges in designing a dynamic optimization system. A number of other difficult issues still must be dealt with in specific scenarios.

19.3.6.1 Self-Modifying and Self-Referential Code

One such issue is the presence of self-modifying or self-referential code. For example, self-referential code may be inserted for a program to compute a check sum on its binary image. To ensure that self-referential behavior is preserved, the loaded program image should remain untouched, which is the case if the dynamic compiler follows the design illustrated in Figure 19.3.

Self-modifying code is more difficult to handle properly. The major difficulty lies in the detection of code modification. Once code modification has been detected, the proper reaction is to invalidate all fragments currently resident in the cache that contain copies of the modified code. Architectural support can make the detection of self-modifying code easy. If the underlying machine architecture provides page–write protection, the pages that hold the loaded program image can simply be write protected. A page protection violation can then indicate the occurrence of code modification and can trigger the corresponding fragment invalidations in the compiled code cache. Without such architectural support, every store to memory must be intercepted to test for self-modifying stores.

19.3.6.2 Transparency

A truly transparent dynamic compilation system can handle any loaded executable. Thus, to qualify as transparent a dynamic compiler must not assume special preparation of the binary, such as explicit relinking or recompilation with dynamic compilation code. To operate fully transparently, a dynamic compiler should be able to handle even legacy code. In a more restrictive setting, a dynamic compiler may be allowed to make certain assumptions about the loaded code. For example, an assumption may be made that the loaded program was generated by a compiler that obeys certain software conventions. Another assumption could be that it is equipped with symbol table information or stack unwinding information, each of which may provide additional insights into the code that can be valuable during optimization.

19.3.6.3 Reliability

Reliability and robustness present another set of challenges. If the dynamic compiler acts as an optional transparent runtime stage, robust operation is of even higher importance than in static compilation stages. Ideally, the dynamic compilation system should reach hardware levels of robustness, though it is not clear today how this can be achieved with a piece of software.

19.3.6.4 Real-Time Constraints

Handling real-time constraints in a dynamic compiler has not been sufficiently studied. The execution speed of a program that runs under the control of a dynamic compiler may experience large variations. Initially, when the code cache is nearly empty, dynamic compilation overhead is high and execution progress is correspondingly slow. Over time, as a program working set materializes in the code cache, the dynamic compilation overhead diminishes and execution speed picks up. In general, performance progress is highly unpredictable because it depends on the code reuse rate of the program. Thus, it is not clear how any kind of real-time guarantees can be provided if the program is dynamically compiled.

19.4 Dynamic Binary Translation

The previous sections have described dynamic compilation in the context of code transformation for performance optimization. Another motivation for employing a dynamic compiler is software migration. In this case, the loaded image is native to a guest architecture that is different from the host architecture, which runs the dynamic compiler. The binary translation model of dynamic compilation is illustrated in Figure 19.7. Caching instruction set simulators [8] and dynamic binary translation systems [19, 35, 39] are examples of systems that use dynamic compilation to translate nonnative guest code to a native host architecture.

An interesting aspect of dynamic binary translation is achieving separation of the running software from the underlying hardware. In principle, a dynamic compiler can provide a software implementation of an arbitrary guest architecture. With the dynamic compilation layer acting as a bridge, both software and hardware may evolve independently. Architectural advances can be hidden and remain transparent to the user. This potential of dynamic binary translation has recently been commercially exploited by Transmeta's code morphing software [14] and Transitive's emulation software layer [37].

The high-level design of a dynamic compiler, if used for binary translation, remains the same as illustrated in Figure 19.3, with the addition of a translation module. This additional module translates

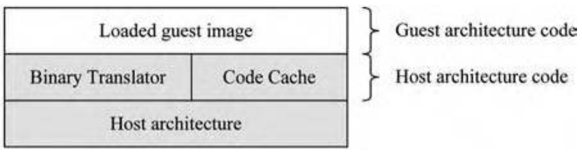


FIGURE 19.7 Binary translation.

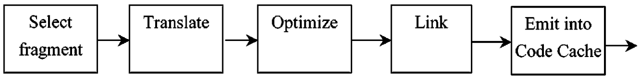


FIGURE 19.8 Dynamic translation pipeline.

fragments selected from guest architecture code into fragments for the host architecture, as illustrated in Figure 19.8.

To produce a translation from one native code format to another, the dynamic compiler may choose to first translate the guest architecture code into an intermediate format and then generate the final host architecture instructions. Going through an intermediate format is especially helpful if the differences in host and guest architecture are large. To facilitate the translation of instructions, it is useful to establish a fixed mapping between guest and host architecture resources, such as machine registers [19].

Although the functionality of the major components in the dynamic compilation stage, such as fragment selection and code cache management, is similar to the case of native dynamic optimization, a number of important challenges are unique to binary translation.

If the binary translation system translates code not only across different architectures but also across different operating systems, then it is called full system translation. The Daisy binary translation system that translates from code for the PowerPC under IBM's UNIX system, AIX, to a customized very long instruction word (VLIW) architecture is an example of full system translation [19]. Full system translation may be further complicated by the presence of a virtual address space in the guest system. The entire virtual memory address translation mechanism has to be faithfully emulated during the translation, which includes the handling of such events as page faults. Furthermore, low-level boot code sequences must also be translated. Building a dynamic compiler for full system translation requires in-depth knowledge of both the guest and host architectures and operating systems.

19.5 Just-In-Time Compilation

JIT compilation refers to the runtime compilation of intermediate virtual machine code. Thus, unlike binary dynamic compilation, the process does not start out with already compiled executable code. JIT compilation was introduced for Smalltalk-80 [18], but has recently been widely popularized with the introduction of the Java programming language and its intermediate bytecode format [23].

The virtual machine environment for a loaded intermediate program is illustrated in Figure 19.9. As in binary dynamic compilation, the virtual machine includes a compilation module and a compiled code cache. Another core component of the virtual machine is the runtime system that provides various system services that are needed for the execution of the code. The loaded intermediate code image is inherently tied to, and does not execute outside, the virtual machine. Virtual machines are an attractive model to implement a "write-once-run-anywhere" programming paradigm. The program is statically compiled to the virtual machine language. In principle, the same statically compiled program may run on any hardware environment, as long as the environment provides an appropriate

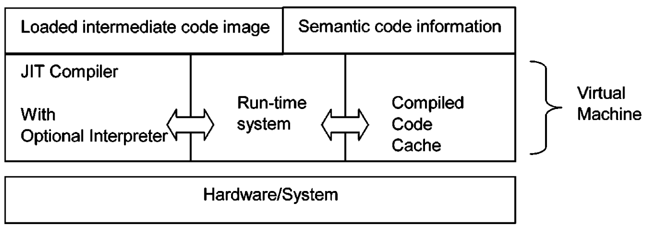


FIGURE 19.9

virtual machine. During execution in the virtual machine, the program may be further (JIT) compiled to the particular underlying machine architecture. A virtual machine with a JIT compiler may or may not include a virtual machine language interpreter.

JIT compilation and binary dynamic compilation share a number of important characteristics. In both cases, the management of the compiled code cache is crucial. Just like a binary dynamic compiler, the JIT compiler may employ profiling to stage the compilation and optimization effort into several modes, from a quick base compilation mode with no optimization to an aggressively optimized mode.

Some important differences between JIT and binary dynamic compilation are due to the different levels of abstraction in their input. To facilitate execution in the virtual machine, the intermediate code is typically equipped with semantic information, such as symbol tables or constant pools. A JIT compiler can take advantage of the available semantic information. Thus, JIT compilation more closely resembles the process of static compilation than does binary recompilation.

The virtual machine code that the JIT compiler operates on is typically location independent, and information about program components, such as procedures or methods, is available. In contrast, binary dynamic compilers operate on fully linked binary code and usually face a code recovery problem. To recognize control flow, code layout decisions that were made when producing the binary have to be reverse engineered, and full code recovery is in general not possible. Because of the code recovery problem, binary dynamic compilers are more limited in their choice of compilation unit. They typically choose simple code units, such as straight-line code blocks, traces or tree-shaped regions. JIT compilers, on the other hand, can recognize higher level code constructs and global control flow. They typically choose whole methods or procedures as the compilation unit, just as a static compiler would do. However, recently it has been recognized that there are other advantages to considering compilation units at a different granularity than whole procedures, such as reduced compiled code sizes [2].

The availability of semantic information in a JIT compiler also allows for a larger optimization repertoire. Except for overhead concerns, a JIT compiler is just as capable of optimizing the code as a static compiler. JIT compilers can even go beyond the capabilities of a static compiler by taking advantage of dynamic information about the code. In contrast, a binary dynamic optimizer is more constrained by the low-level representation and the lack of a global view of the program. The aliasing problem is worse in binary dynamic compilation because the higher level type information that may help to disambiguate memory references is not available. Furthermore, the lack of a global view of the program forces the binary dynamic compiler to make worst-case assumptions at entry and exit points of the currently processed code fragment, which may preclude otherwise safe optimizations.

The differences in JIT compilation and binary dynamic compilation are summarized in Table 19.1. A JIT compiler is clearly more able to produce highly optimized code than a binary compiler. However, consider a scenario where the objective is not code quality but compilation speed. Under these conditions, it is no longer clear that the JIT compiler has an advantage. A number of compilation and code generation decisions, such as register allocation and instruction selection, have already been made in the binary code and can often be reused during dynamic compilation. For

TABLE 19.1 Differences in JIT Compilation and Binary Dynamic Compilation

JIT Compilation	Dynamic Binary Compilation
Semantic information available	Lack of semantic information
Full code recovery	Limited code recovery; limited choice in compilation unit
Full optimization repertoire	Limited optimization potential

example, binary translators typically construct a fixed mapping between guest and host system machine registers. Consider the situation where the guest architecture has fewer registers, say 32, than the host architecture, say 64, so that the 32 guest registers can be mapped to the first 32 host register. When translating an instruction *opcode, op*1*, op*2, the translator can use the fixed mapping to directly translate the operands from guest to host machine registers. In this fashion, the translator can produce code with globally allocated registers without any analysis, simply by reusing register allocation decisions from the guest code.

In contrast, a JIT compiler that operates on intermediated code has to perform a potentially costly global analysis to achieve the same level of register allocation. Thus, what appears to be a limitation may prove to have its virtues depending on the compilation scenario.

19.6 Nontransparent Approach: Runtime Specialization

A common characteristic among the dynamic compilation systems discussed so far is transparency. The dynamic compiler operates in complete independence from static compilation stages and does not make assumptions about, or require changes to, the static compiler.

A different, nontransparent approach to dynamic compilation has been followed by staged runtime specialization techniques [9, 31, 33]. The objective of these techniques is to prepare for dynamic compilation as much as possible at static compilation time. One type of optimization that has been supported in this fashion is value-specific code specialization. Code specialization is an optimization that produces an optimized version by customizing the code to specific values of selected specialization variables.

Consider the code example shown in Figure 19.10. Figure 19.5(i) shows a dot product function that is called from within a loop in the main program, such that two parameters are fixed ($n = 3$ and $row = [5, 0, 3]$) and only a third parameter (col) may still vary. A more efficient implementation can be achieved by specializing the dot function for the two fixed parameters. The resulting function *spec_doc*, which retains only the one varying parameter, is shown in Figure 19.10(ii).

In principle, functions that are specialized at runtime, such as *spec_dot,* could be produced in a JIT compiler. However, code specialization requires extensive analysis and is too costly to be performed fully at runtime. If the functions and the parameters for specialization are fixed at compile time, the static compiler can prepare the runtime specialization and perform all the required code analyses. Based on the analysis results, the compiler constructs code templates for the specialized procedure. The code templates for *spec_dot* are shown in Figure 19.11(ii) in C notation. The templates may be parameterized with respect to missing runtime values. Parameterized templates contain holes that are filled in at runtime with the respective values. For example, template T2 in Figure 19.11(ii) contains two holes for the runtime parameters $row[0] \ldots$ row[2] (hole h1) and the values $0, \ldots, (n - 1)$ (hole h2).

By moving most of the work to static compile time, the runtime overhead is reduced to initialization and linking of the prepared code templates. In the example from Figure 19.10, the program is statically

```
dot(int  n,  int  row[],  int
col[])
{
  int i, sum;
  sum = 0;
 for (i=0; i<n; i++)
     sum += row[i]* col[i];
  return sum;
}

main() {
  read(&n, row);
  . . .
  while (. . .) {
    /* n=3, row={5,0,3} */
    dot(n, row, col);
    . . .
  }
}
            (i)
```

```
spec_dot(int col[]) {
   int sum = 0;
   sum += 5*  col[0];
   sum += 3*  col[2];
   return sum;
}


main{} {
   read(&n, row);
   make_spec_dot(n,
row);
   . . .
   while (. . .) {
      spec_dot(col);
      . . .
   }
}
              (ii)
```

FIGURE 19.10 Code specialization example. A dot–product function before (i) and after specialization (ii).

compiled so that in place of the call to routine *dot,* a call to a specialized dynamic code generation agent is inserted. The specialized code generation agent for the example from Figure 19.10, *make_spec_dot*, is shown in Figure 19.11(i). When invoked at runtime, the specialized dynamic compiler looks up the appropriate code templates for *spec_dot,* fills in the holes for parameters *n* and *row* with their runtime values, and patches the original main routine to link in the new specialized code.

The required compiler support renders these runtime specialization techniques less flexible than transparent dynamic compilation systems. The kind, scope and timing of dynamic code generation are fixed at compile time and hardwired into the code. Furthermore, runtime code specialization techniques usually require programmer assistance to choose the specialization regions and variables (e.g., via code annotations or compiler directives). Because overspecialization can easily result in code explosion and performance degradation, the selection of beneficial specialization candidates is likely to follow an interactive approach, where the programmer explores various specialization opportunities. Recently, a system has been developed toward automating the placement of compiler directives for dynamic code specialization [32].

The preceding techniques for runtime specialization are classified as declarative. Based on the programmer declaration, templates are produced automatically by the static compiler. An alternative approach is imperative code specialization. In an imperative approach, the programmer explicitly encodes the runtime templates. 'C is an extension of the C languages that allows the programmer to specify dynamic code templates [33]. The static compiler compiles these programmer specifications into code templates that are initiated at runtime in a similar way to the declarative approach. Imperative runtime specialization is more general because it can support a broader range of runtime code generation techniques. However, it also requires deeper programmer involvement and is more error prone, due to the difficulty of specifying the dynamic code templates.

```
make_spec_dot(int n, int row[]) {
  buf = alloc();     /* allocate buffer space for spec_dot */

  copy_temp(buf,T1);       /* copy template T1 into buffer */
  for (i=0; i<n; i++) {
     copy_temp(buf, T2);             /* copy template T2 */
     fill_hole(buf, h1, row[i]);    /* fill hole h1 in T2 */
     fill_hole(buf, h2, i);         /* file hole h2 in T2 */
  }
  copy_temp(buf, T3);               /* copy template T3 */
  return buf;
}

                              (i)


Template T1:         spec_dot(int col[]) {  int sum = 0;

Template T2:         sum += {hole h1} * col[{hole h2}];

Template T3:         return sum; }

                              (ii)
```

FIGURE 19.11 Runtime code generation function (i) and code templates (ii) for specializing function dot from Figure 19.10.

19.7 Summary

Dynamic compilation is a growing research field fueled by the desire to go beyond the traditional compilation model that views a compiled binary as a static immutable object. The ability to manipulate and transform code at runtime provides the necessary instruments to implement novel execution services. This chapter discussed the mechanisms of dynamic compilation systems in the context of two applications: dynamic performance optimization and transparent software migration. However, the capabilities of dynamic compilation systems can go further and enable such services as dynamic decompression and decryption or the implementation of security policies and safety checks.

Dynamic compilation should not be viewed as a technique that competes with static compilation. Instead, dynamic compilation complements static compilation, and together they make it possible to move toward a truly write-once-run-anywhere paradigm of software implementation.

Although dynamic compilation research has advanced substantially in recent years, numerous challenges remain. Little progress has been made in providing effective development and debugging support for dynamic compilation systems. Developing and debugging a dynamic compilation system is particularly difficult because the source of program bugs may be inside transient dynamically generated code. Break points cannot be placed in code that has not yet materialized and symbolic debugging of dynamically generated code is not an option. The lack of effective debugging support is one of the reasons that make the engineering of dynamic compilation systems such a difficult task. Another area that needs further attention is code validation. Techniques are needed to assess the correctness of dynamically generated code. Unless dynamic compilation systems can guarantee high levels of robustness, they are not likely to achieve widespread adoption.

This chapter surveys and discusses the major approaches to dynamic compilation with a focus on transparent binary dynamic compilation. For more information on the dynamic compilation systems that have been discussed, we encourage the reader to explore the sources cited in the Reference Section.

References

[1] L. Anderson, M. Berc, J. Dean, M. Ghemawat, S. Henzinger, S. Leung, L. Sites, M. Vandervoorde C. Waldspurger and W. Weihl, Continuous Profiling: Where Have All the Cycles Gone, in Proceedings of the 16th ACM Symposium of Operating Systems Principles, 14, 1997.

[2] D. Bruening and E. Duesterwald, Exploring Optimal Compilation Unit Shapes for an Embedded Just-in-Time Compiler, in Proceedings of the 3rd Workshop on Feedback-Directed and Dynamic Optimization, 2000.

[3] D. Bruening, E. Duesterwald and S. Amarasinghe, Design and Implementation of a Dynamic Optimization Framework for Windows, in Proceedings of the 4th Workshop on Feedback-Directed and Dynamic Optimization, December 2001.

[4] V. Bala, E. Duesterwald and S. Banerjia, Transparent Dynamic Optimization: The Design and Implementation of Dynamo, Hewlett-Packard Laboratories Technical report HPL-1999–78, June 1999.

[5] V. Bala, E. Duesterwald and S. Banerjia, Dynamo: A Transparent Runtime Optimization System, in Proceedings of the SIGPLAN ’00 Conference on Programming Language Design and Implementation, June 2000, pp. 1–12.

[6] T. Ball and J. Larus, Efficient Path Profiling, in Proceedings of the 29th Annual International Symposium on Microarchitecture (MICRO-29), Paris, 1996, pp. 46–57.

[7] W. Chen, S. Lerner, R. Chaiken and D. Gillies, Mojo: A Dynamic Optimization System, in Proceedings of the 3rd Workshop on Feedback-Directed and Dynamic Optimization, December 2000.

[8] R.F. Cmelik and D. Keppel, Shade: A Fast Instruction Set Simulator for Execution Profiling, Technical report UWCSE-93-06-06, Department of Computer Science and Engineering, University of Washington, Seattle, WA, 1993.

[9] C. Consel and F. Noel, A General Approach for Run-Time Specialization and Its Application to C, in Proceedings of the 23rd Annual Symposium on Principles of Programming Languages, 1996, pp. 145–156.

[10] T. Conte, B. Patel, K. Menezes and J. Cox, Hardware-based profiling: an effective technique for profile-driven optimization, *Int. J. Parallel Programming*, 24, 187–206, April 1996.

[11] C. Chambers and D. Ungar, Customization: Optimizing Compiler Technology for SELF, a Dynamically-Typed Object-Oriented Programming Language, in Proceedings of the SIGPLAN ’89 Conference on Programming Language Design and Implementation, 1989, pp. 146–160.

[12] Y.C. Chung and Y. Byung-Sun, The Latte Java Virtual Machine, Mass Laboratory, Seoul National University, Korea, latte.snu.ac.kr/manual/html_mono/latte.html.

[13] R. Cohn and G. Lowney, Hot Cold Optimization of Large Windows/NT Applications, in Proceedings of the 29th Annual International Symposium on Microarchitecture, 1996, pp. 80–89.

[14] D. Ditzel, Transmeta’s Crusoe: Cool chips for mobile computing, in *Hot Chips 12*, Stanford University, August 2000.

[15] D. Deaver, R. Gorton and N. Rubin, Wiggins/Redstone: An On-Line Program Specializer, in Proceedings of Hot Chips 11, Palo Alto, CA, August 1999.

[16] E. Duesterwald, R. Gupta and M.L. Soffa, Demand-Driven Computation of Interprocedural Data Flow, in Proceedings of the 22nd ACM Symposium on Principles on Programming Languages, 1995, pp. 37–48.

[17] E. Duesterwald and V. Bala, Software Profiling for Hot Path Prediction: Less Is More, in Proceedings of 9th International Conference on Architectural Support for Programming Languages and Operating Systems, 2000, pp. 202–211.

[18] L.P. Deutsch and A.M. Schiffman, Efficient Implementation of the Smalltalk-80 System, in Conference Record of the 11th Annual ACM Symposium on Principles of Programming Languages, 1994, pp. 297–302.

[19] K. Ebcioglu and E. Altman, DAISY: Dynamic Compilation for 100% Architectural Compatibility, in Proceedings of the 24th Annual International Symposium on Computer Architecture, 1997, pp. 26–37.

[20] D.R. Engler, VCODE: A Retargetable, Extensible, Very Fast Dynamic Code Generation System, in Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation (PLDI '96), May 1996, pp. 160–170.

[21] D.H. Friendly, S.J. Patel and Y.N. Patt, Putting the Fill Unit to Work: Dynamic Optimizations for Trace Cache Microprocessors, in Proceedings of the 31st Annual International Symposium on Microarchitecture (MICRO-31), 1998, pp. 173–181.

[22] M. Gschwind and E. Altman, Optimization and Precise Exceptions in Dynamic Compilation, in Proceedings of Workshop on Binary Translation, 2000.

[23] J. Gosling, B. Joy and G. Steele, The Java Language Specification, Addison-Wesley, Reading, MA, 1996.

[24] B. Grant, M. Philipose, M. Mock, C. Chambers and S. Eggers, An Evaluation of Staged Run-Time Optimizations in DyC, in Proceeding*s* of the SIGPLAN '99 Conference on Programming Language Design and Implementation, 1999, pp. 293–303.

[25] U. Hoelzle, C. Chambers and D. Ungar, Optimizing Dynamically-Typed Object-Oriented Languages with Polymorphic Inline Caches, in Proceedings ECCOP 4th European Conference on Object-Oriented Programming, July 1991, pp. 21–38.

[26] U. Hoelzle, C. Chambers and D. Ungar, Debugging Optimized Code with Dynamic Deoptimization, in Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation, June 1992, pp. 32–43.

[27] R.J. Hookway and M.A. Herdeg, FX!32: combining emulation and binary translation, *Digital Tech. J.*, 0(1), pp. 3–12, 1997.

[28] Open Runtime Platform, Intel Microprocessor Research Lab, www.intel.com/research/mrl/orp/.

[29] The IBM Jalapeno Project, IBM Research, www.research.ibm.com/jalapeno/.

[30] C. Krintz and B. Calder, Using Annotations to Reduce Dynamic Optimization Time, in Proceedings of the SIGPLAN '01 Conference on Programming Language Design and Implementation, 2001, pp. 156–167.

[31] M. Leone and P. Lee, Optimizing ML with Run-Time Code Generation, in Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation, 1996, pp. 137–148.

[32] M. Mock, M. Berryman, C. Chambers and S. Eggers, Calpa: A Tool for Automatic Dynamic Compilation, in Proceedings of the 2nd Workshop on Feedback-Directed and Dynamic Optimization, 1999.

[33] M. Poletta, D.R. Engler and M.F. Kaashoek, TCC: A System for Fast Flexible, and High-Level Dynamic Code Generation, in Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation, 1997, pp. 109–121.

[34] S. Richardson, Exploiting Trivial and Redundant Computation, in Proceedings of the 11th Symposium on Computer Aruthmetic, 1993.

[35] K. Scott and J. Davidson, Strata: A Software Dynamic Translation Infrastructure, in Proceedings of the 2001 Workshop on Binary Translation, Barcelona, Spain, 2001.

[36] A. Srivastava, A. Edwards and H. Vo, Vulcan: Binary Translation in a Distributed Environment, Technical report MSR-TR-2001-50, Microsoft Research, 2001.

[37] Transitive Technologies, www.transitives.com/.
[38] O. Taub, S. Schechter and M.D. Smith, Ephemeral Instrumentation for Lightweight Program Profiling, Technical Report, Harvard University, 2000.
[39] D. Ung and C. Cifuentes, Machine-adaptable dynamic binary translation, in Proceedings of the ACM Sigplan Workshop on Dynamic and Adaptive Compilation, *ACM Sigplan Notices*, 35(7), 41–51, 2000.
[40] X. Zhang, Z. Wang, N. Gloy, J. Chen and M. Smith, System Support for Automatic Profiling and Optimization, in Proceedings of the 16th ACM Symposium on Operating Systems Principles, 1997, pp. 15–26.

20

Compiling Safe Mobile Code

Ravindra B. Keskar
Sasken Communication Technologies Limited

R. Venugopal
Hewlett-Packard India Software Operations

20.1 Introduction

This chapter deals with checking the safety of executing code obtained from an untrusted source. Such a scenario is fairly common with the increasing popularity of the Internet. In addition, with the arrival of the programming language Java it is likely to become even more common. Applets and plug-ins are routinely downloaded and executed. The characteristic feature of this kind of activity, as far as this chapter is concerned, is that the code comes very often from a third party who may be an untrusted source. We call the person who has produced the code the *code producer*[1] and the person who uses the program the *code consumer*. The problem that is the topic of this chapter is as follows:

[1]It is possible that the code producer is another program.

0-8493-1240-7/03/$0.00+$1.50

how does the code consumer ensure that the code producer's code either maliciously or inadvertently does not corrupt the former's system resources? Note that we allow a weaker interpretation of the word *corrupt*: it is possible that the foreign code may just acquire resources without releasing it instead of proactively causing harm to the consumer's system. The process of ensuring the safety of the code consumer's system is called *safety checking*.

Safety checking of code has always been a significant problem. Code consumers in the past have used code obtained from a third party. In recent years, it is just that the problem has become more significant due to the increase in scale in the usage of code written by third parties.

There is yet another reason why the problem has become more important now. Software projects are increasingly executed by using a component-based design methodology. Commercial products for facilitating this have become available (e.g., COM, CORBA, JavaBeans, etc.). In such a situation, it becomes imperative to verify the safety of the components that are used to build the application. Component-based design improves flexibility and reuse in software projects, but the risk of harm to computer resources has correspondingly increased.

Another reason exists as to why it is imperative to have a good technology for safety checking of code. System software design has evolved to allow user-defined extensions to the functionality and services provided. For example, many operating systems provide extensible kernel where the user is allowed to write customized services and have them executed in the kernel address space. The BSD packet filter [23] is an early example of this trend. The user can write packet filters that filter out network packets based on this user's specification. Other examples are application-specific virtual memory management [8] and active messages [23]. Active networks [22] allow user-defined networking code such as routing algorithms to be injected into intermediate nodes of the network to optimize on parameters such as network congestion and load. The POSTGRES database manager has an extensible type system. Languages such as metalanguage (ML) allow the user to extend its type system.

At this point, let us say a few words that are aimed at restricting the scope of the chapter. Cardelli [2] draws a distinction between mobile code and mobile computation. When one refers to mobile code, one is referring to only the code as the mobile entity. This is exactly the situation with the Java programming language. When a Java source program is compiled to bytecode, transferred over the network and interpreted by a Java virtual machine (JVM) at another network node, we may claim that the bytecode is mobile. In mobile computation, not only the code but its associated state or context is mobile. In this case, the situation is that the execution of a piece of code moves from one node to another (i.e., the code as well as the state moves from one place to another). A piece of computation that is mobile is also called a *mobile ambient*. In this chapter, we do not consider safety checking of mobile ambients at all. This is an emerging area of study and [3, 4] are some of the references that deal with this problem. We restrict ourselves to safety checking of mobile code.

To a large extent, we discuss only those techniques that are general in nature. (However, in discussing these techniques we present examples from the corresponding references that deal with specific languages and processor architectures; this is just to make the techniques clearer to the reader.) For instance, we do not discuss Java bytecode verification because this would require us to get into the details of the Java bytecode specification. We also concentrate on techniques that are, in some way, related to compiler technology. The techniques that we discuss use principles of programming languages (types and type checking), logic, semantics of assembly language instructions, etc., all of which are relevant to compiler technology in one form or the other. For instance, we do not discuss at all techniques such as digital signatures that may be used in authenticating the foreign code.

We also make no claim on the completeness of this survey. The goal of this chapter is to give a flavor of the problem tackled and some of the solutions that are possible. The field is still developing and it is too early to claim the superiority of one solution over the other. It is hoped that after reading this chapter, the reader will realize that a lot of issues are involved in safety checking that may be resolved by borrowing techniques from compilers, programming languages and formal methods.

20.2 A Rough Classification

We now present a rough classification of the techniques of mobile code checking that are relevant for this chapter. A similar classification appears in [25].

Safety checking techniques are either static, dynamic or hybrid. Static methods ensure the safety of foreign code through some form of static analysis. These techniques have the advantage that the number of runtime checks for safety may be reduced in the code. The problems associated with static methods are scalability, conservativeness of the analysis applied and preexecution overhead involved. The static methods that we consider in this chapter are proof carrying code (PCC) [16], typed assembly languages [13], safety checking of machine code [25] and certifying compilation [9, 18].

Dynamic methods work by inserting runtime checks into the code that check for the safety of the program during execution. These methods scale to large programs, but may slow down the execution of the program. Also, some mechanism must be provided for the program to recover from an unsafe state. Software-based fault isolation [24] is the technique that we present as an example of a dynamic safety checking method.

Hybrid methods employ a combination of both static checking and runtime checking. This allows fewer runtime checks than a purely dynamic safety checking method. Many languages use hybrid techniques to ensure the safety of code written in these languages. Examples are Java and Modula 3.

Let us mention another way of looking at safety-checking techniques. Some techniques ensure safety checking by restricting the kind of programs that can be written in the language. For example, in the BSD packet filter approach a special-purpose language exists in which to write the packet filter specification such that only memory associated with a packet or legitimately allowed scratch memory is used. Techniques that work by restricting the languages to a safe subset include the typed assembly language (TAL) approach [13] and certifying compilation [9, 18]. Other techniques do not need the language to be restricted. These techniques check for the validity of safety predicates on the assembly language without requiring the language to be typed. An example of this approach is PCC [16]. Type-state checking of machine code [25, 26] may also be put in this category; although, in this case, it is required that the user supply some type and state information of host data and constraints.

The rest of this chapter is structured as follows. Section 20.3 discusses a dynamic scheme for safety checking, namely, software-based fault isolation. Section 20.4 describes the TAL approach to safety checking. In Section 20.5 we describe the PCC approach. Safety checking can also be done directly on the assembly code irrespective of the source language or the compiler and this technique is described in Section 20.6. Section 20.7 describes one approach to the design of a certifying compiler. In Section 20.8 we describe a certifying compiler that works on dynamically generated code. Section 20.9 concludes this brief survey.

20.3 Software-Based Fault Isolation

One approach to ensure that an untrusted code module does not tamper with the address space of another module is to isolate individual modules within disjoint address spaces. A particular module may access the code or data contained only within its address space. This technique is called *sandboxing*. A distrusted code module interacts with other modules through the remote procedure call (RPC) interface [1]. RPC provides a normal procedure call interface for cooperating modules to interact among one another. Code safety is ensured by using hardware means (perhaps hardware page tables) to ensure that one module does not corrupt the address space of another module.

For a particular RPC call from one module to another, expensive context switching is involved. The steps involved are trapping into the operating system (OS) kernel, copying the RPC call arguments from the caller to the callee, saving and restoring of registers, switching hardware address spaces and then switching back to the user level. When a large number of RPC calls occur (in other words, when the cooperating modules form a tightly coupled system), this method of ensuring safety may prove expensive. This is because the overhead is incurred at runtime. This is a drawback associated with any dynamic scheme for code safety.

Software-based fault isolation is a dynamic scheme for code safety but attempts to reduce some of the overhead involved in hardware-based context switching. In this approach, the cooperating modules all operate within the same address space, which, however, is partitioned into what are called *fault domains*. Each module is associated with a distinct fault domain. The code that it accesses and the data space that it reads or writes are restricted to lie in the same fault domain. Thus, software fault isolation implements the sandboxing approach within a single address space. As a result, the expense associated with hardware-based context-switching schemes is avoided. Instructions are inserted just before potentially unsafe instructions in untrusted modules that check whether the target address lies within the fault domain of the untrusted module. This has the effect of slowing down the execution of distrusted modules to a certain extent, but the execution time of trusted modules is not affected.

The virtual address space of an application is divided into segments. A segment is identified by a unique pattern in the upper bits of the address used in the program. A fault domain for a module consists of two segments — one, for the code and one for the data that it uses. Those addresses that can be statically verified by the compiler to lie within the fault domain are deemed safe. The other addresses are potentially unsafe.

Two subproblems need to be solved for achieving efficient software-based fault isolation. The first is to ensure that each module is sandboxed within its fault domain. This implies ensuring that all addresses used within the module are restricted to the two segments that constitute the fault domain for that module.

This, in turn, is achieved by segment matching. Segment matching is the process of checking, for each potentially unsafe target address, whether it is, in fact, restricted to the fault domain. If the target address refers to a location outside the fault domain, then a trap to an error routine occurs. This also allows one to identify the offending piece of code.

If the runtime overhead incurred in identifying the offending instruction is unacceptable, then address sandboxing can be used to ensure that the target address used by a potentially unsafe instruction is always within the fault domain. This is done by inserting code that sets the upper bits of the address to the associated segment identifier.

The second subproblem is to develop a scheme for low-latency communication across the fault domains. Wahbe et al. [24] give the assembly language pseudo code for segment matching. Four instructions are required for doing this. Table 20.1 reproduces the pseudo code from [24].

TABLE 20.1 Assembly Pseudo Code for Segment Matching

```
dedicated – register ⇐ targetaddress
/∗ Move target address into dedicated register ∗/
scratch – register ⇐ (dedicated – register ≫ shift – register)
/∗ Right – shift address to get segment identifier. shift – register is a
 dedicated register ∗/
compare scratch – register and segment – register
/∗ segment – register is a dedicated register ∗/
trap if not equal
```

TABLE 20.2 Assembly Pseudo Code for Address Sandboxing

```
dedicated – register ⇐ target – register & and – mask – register
/∗ Clearing the bits which will store the segment identifier ∗/
dedicated – register ⇐ (dedicated – register | segment – register)
/∗ Storing the bits representing the segment identifier ∗/
```

Table 20.2 gives the pseudo code from [24] for doing the address sandboxing.

Wahbe et al. [24] describe two ways in which these assembly language instructions may be generated. The first way is the one adopted by the implementation described in the same paper. In this method, the compiler generates these instructions. The second method is binary patching, which is a method to rewrite the original binary into a form that implements segment matching or address sandboxing as the case may be.

Wahbe et al. [24] also describe the scheme for generating cross-fault domain calls. According to this scheme, a fault domain can safely call a trusted stub routine outside its domain, which, in turn, safely calls a routine in the destination domain. The scheme includes mechanisms for efficient passing of arguments from the called domain to the callee domain. The stub is also responsible for saving the machine state and registers during the procedure call. The details may be obtained from [24].

20.4 Typed Assembly Language

20.4.1 Introduction

One way to ensure the safety of a mobile code is that the user should not express, in a source program, anything that is unsafe for the machine on which the code runs. This can be achieved by making the source language safe (i.e., by the source language not allowing anything unsafe to be expressed in it). Thus, if nothing bad can be expressed in the source language and if the compiler ensures that it does not add anything bad while compiling the source code to the assembly code, we can be sure that the target assembly code is safe. This approach requires the following:

1. A source language that does not allow the user to (intentionally or inadvertently) express anything harming the safety of the machine on which the code will run
2. A compiler that translates a "safe" source code to a "safe" assembly code

The main goal of any traditional type-checking system is to ensure that all operations in a program are type safe. Extending this notion further, we can view the problem of safety checking of a mobile code as a type-checking problem. Of course, in this case, we have to model the external environment (static as well as runtime environment in which the code will run) as an entity having type. Also, we need to define runtime activities (like stack allocation and heap allocation) as operations that operate on the environment entities (like program stack and heap). Having done this, we should be able to determine and define what is safe and what is unsafe with respect to each operation. If we can model the external environment entities as program entities having types, then the problem of checking whether a mobile code conforms to the safety conditions or not reduces to a traditional type-checking problem.

The TAL framework that is used for safety checking of machine code is based on the preceding philosophy. TAL provides an automatic way to verify that the source program is safe. This is done by ensuring that the basic (high-level) types are preserved at the assembly language level. The TAL system gives types to the target assembly language and to all the intermediate languages used by the compiler while generating the target code. At every level, the types are preserved and the type safety is ensured. In short, the TAL system provides a framework for converting well-typed source programs to well-typed assembly programs. The type checking is done statically (i.e., it is off-line).

Most of the traditional compilers work on untyped intermediate languages. The type structure of program entities present at the higher level is lost at the lower levels when the code goes through multiple translations done by the compiler. The absence of types not only makes type checking difficult at the lower levels, but also restricts the range of optimizations that can be applied at the lower levels. For example, the presence of the type structure facilitates optimizations (for higher order languages) such as closure conversion, unboxing and region inference at lower levels. More importantly, as the typing structure is preserved at the lower levels, the program can be type checked even at the assembly level. This can be useful in checking whether the compilation process itself is safe, in the sense that whether it produces type-safe assembly language. This is very useful to cross-check the correctness of newly introduced program transformation and optimization techniques.

One main issue in a TAL-like framework is to develop a type structure for the assembly language. The typing structure should be able to define higher level abstract types. At the same time, this additional structure on the assembly language should not be a hindrance to low-level optimizations. To make the typed assembly language more useful, it is required that we should be able to compile any source language program to the typed assembly language program. Thus, the typing structure on the assembly language should be general enough to represent important abstractions in various high-level languages; at the same time, the target code should not be inefficient.

20.4.2 Overview of the Typed Assembly Language Framework

We describe here a TAL framework developed by Morrisett et al. [13, 14]. It takes advantage of typing information present at the assembly language level. TAL is a statically typed assembly language. Low-level optimizations such as global register allocation, copy propagation, constant folding and dead code elimination are supported by the TAL framework. The TAL system can be used to check untrusted code. Specifically, irrespective of the compiler that produced it, the TAL framework can check for the type safety of the program. The type-checking system is a conventional typing system. It also achieves flow sensitivity by giving types to the registers in a code block. Also, the TAL system supports higher level programming features such as tuples, polymorphism, existentials and restricted form of function pointers. In essence, the TAL framework gives an automatic way to convert a well-typed source term to a well-typed target term and, in doing so, opens up a lot of avenues for more efficient low-level optimizations and more efficient safety checking. It provides an automatic way to produce proof-carrying code that we describe later in this chapter.

The TAL system provides a set of abstractions such as integers, pointers to tuples, code labels, modules [11] and runtime stacks [12]. Not all the operations are permitted on these abstractions. A program is said to be *stuck* if it tries to do an operation that is not permitted by the system. The TAL-type system has to ensure that all well-typed programs do not get stuck. Integers are considered to be different from pointers by the abstraction. Arithmetic operations are allowed only on integers whereas only pointers can be dereferenced. This strong typing property helps in checking many safety conditions.

This section is organized as follows. First, we describe in brief some of the important issues to be tackled while putting a type structure on an assembly language. This is based on [14] where the source language is a type-safe C-like language (called Popcorn) and the target assembly language is TALx86, an assembly language based on Intel IA32 architecture. Popcorn can be compiled to TALx86 [14]. Then we illustrate the typical stages of a compiler that generates a typed intermediate language program at every stage as given in [13]. The assembly language used for this illustration is known as TAL (the original typed assembly developed by Morrisett et al.), which is developed for a reduced instruction set computer (RISC)-like machine. We describe the design of a compiler that converts programs in a higher level language (a variant of polymorphic λ calculus) to TAL. It also demonstrates the expressiveness of TAL. Finally, we see how TAL-like languages are helpful even in some other methods of safety checking.

20.4.3 TALx86: Example of a Typed Assembly Language

The source language we consider here is a C-like language called Popcorn whereas the target language is known as TALx86. TALx86 is an assembly language based on IA32 architecture. The type system for TALx86 provides support for stack allocation, higher order and recursive-type constructors and polymorphism. It also provides support for separate-type checking and linking. Popcorn is a C-like higher level language. Though the Popcorn language is based on C, it provides support for higher level constructs such as polymorphism, abstract types and tagged unions and exceptions. At the same time, it does not support pointer arithmetic and pointer casts. TALx86 is a generic language in the sense that programs written in languages other than Popcorn can also be compiled into it.

20.4.3.1 TALx86 System

First, we give an example of a small Popcorn program and its conversion to a well-typed TALx86 program [14]. The program calculates the sum of the first *n* natural numbers:

```
int i = n + 1;
int s = 0;
while(− − i > 0)
s+ = i;
```

The preceding code fragment after translation into TALx86 looks like this:

```
 mov eax, ecx ;  i = n
 inc eax ; ++i
 mov ebx,0 ; s = 0
 jmp test
body:  {eax: B4, ebx: B4}
 add ebx, eax ; s+ = i
test:  {eax: B4, ebx: B4}
 dec eax ; --i
 cmp eax,0 ; i > 0
 jg  body
```

The argument *n* is stored initially in the register **ecx**. The code labels are annotated with type conditions. The annotation on the code label **body** require **eax** and **ebx** to have 4-byte integers (B4). These are preconditions that are checked when the control is transferred to the block. The register type can be polymorphic and it can be represented using abstract types. The type-checking system verifies that all instructions in a code block obey the typing conditions. In this way, the TAL system achieves flow sensitivity at the block level.

Similarly, data items can also be annotated to specify the type of the data item. Other features of TALx86 are the use of type coercions on instructions and the use of macro instructions to encapsulate a code sequence.

Some other important issues handled by TALx86 are:

- *Stack modeling.* To model the runtime environment, the type system should provide a facility to handle function calling using stacks. TALx86 has a stack abstraction for control flow stacks.

The stack can be given polymorphic types, which helps in abstracting a part of the stack. It also supports a form of polymorphic recursion. Exceptions can be handled by using a limited form of pointers into the middle of the stack.

- *Heap memory allocation.* The target assembly language should be able to handle runtime memory allocation if it is provided in the source language. Popcorn provides a facility for runtime memory allocation. TALx86 handles the corresponding heap memory allocation using a *malloc* construct to allocate memory and then uses a sequence of instructions to initialize it. Because the type-checking system does not typically have a provision to track aliasing, it cannot know the aliasing information and can give some approximate results. Because of limited flow sensitivity of the type-checking system and lack of aliasing information, some optimizations cannot be performed and the resulting code is not fully optimized.

Arrays are also supported in TALx86. The size of the array can be easily tracked. Array bound checks are done during subscript or update operation. Pointers into the middle of the array are not supported. TALx86 also support lists that have sum and recursive type.

Other enhancements to TALx86 are floating point and object abstraction support. Also, data flow analysis can be used to get more optimized code. The standard data flow analysis and shape analysis can help in discovering facts such as aliasing memory, removing some redundant checks and producing more correct results.

20.4.4 Illustrating the Compiler Construction

In this section, we describe construction and stages of a compiler that ensure a well-typed source term is converted to a well-typed target term. Also, all the intermediate terms generated are well typed. This section is based on [11, 13] and should give the reader an idea of the issues involved in the construction of such a compiler.

The source language for the compiler, λ^F, is a variant of the polymorphic λ-calculus (also known as system F) and the target language is TAL. The compiler is structured as four translations between five-typed calculi. Each translation accepts a well-typed program of its input calculus and produces a well-typed program of its output calculus. It does not depend on the fact that the input program is an output of the preceding translation. In this sense, each calculus acts as a first-class programming language. The compilation of λ^F to TAL is represented as follows [13]:

$$\lambda^F \longrightarrow \lambda^K \longrightarrow \lambda^C \longrightarrow \lambda^A \longrightarrow \text{TAL}$$

λ^K, λ^C and λ^A are intermediate well-typed languages generated during the compilation process. The translations used earlier are as follows:

1. *Continuation passing style (CPS) conversion* ($\lambda^F \longrightarrow \lambda^K$). This is the first translation stage in the compiler. Continuation is a method used in higher order languages to express the semantics of control operations. The CPS conversion is considered to be an elegant and useful compilation technique for compiling higher order functional languages [7]. The CPS conversion stage here fixes the order of evaluation and then names the intermediate computations. This translation stage also performs some optimizations on the CPS converted term.
2. *Closure conversion* ($\lambda^K \longrightarrow \lambda^C$). A *closure* is a data structure representing a function as a piece of code for the function and data containing free variables in the original function. Closure conversion stage does function rewriting to form closures and achieves separation between code and data.
3. *Allocation* ($\lambda^C \longrightarrow \lambda^A$). This translation makes heap memory allocation explicit. It makes the language very close to the TAL language.
4. Code generation ($\lambda^A \longrightarrow$ TAL). Code generation is the last compilation stage to generate the TAL code.

20.4.4.1 CPS Conversion

The input language for the compilation process is λ^F, which is a variant of system F. It is a call by value variant also implementing products and recursion on terms. The syntax of λ^F is as follows:

$$\begin{array}{lll} \textit{types} & \tau ::= & \alpha \mid \textit{int} \mid \tau_1 \to \tau_2 \mid \forall\alpha.\tau \mid \langle \tau_1, \ldots, \tau_n \rangle \\ \textit{terms} & e ::= & x \mid i \mid \textit{fix}\ \ x(x_1 : \tau_1) : \tau_2.e \mid e_1 e_2 \mid \Lambda\alpha.e \mid e[\tau] \\ & & \mid \langle e_1, \ldots, e_n \rangle \mid \pi_i(e) \mid e_1\ p\ e_2 \mid if0(e_1, e_2, e_3) \\ \textit{prims} & p ::= & + \mid - \mid \times \end{array}$$

The preceding syntax definition has the following characteristics:

- The only base type is integer. The operations on the base type are given by primitives p.
- The term $\textit{fix}\ \ x(x_1 : \tau_1) : \tau_2.e$ denotes the definition of a function x with its argument as x_1 of type τ_1 with the body being e having type τ_2.
- $\langle \tau_1, \ldots, \tau_n \rangle$ represents a tuple.
- For $\Lambda\alpha.e$, α is bound in e and for $\forall\alpha.\tau$, it is bound in τ.
- Projection is denoted as π_e. It represents the ith element of the tuple e.
- Term $\textit{if}\,0(e_1, e_2, e_3)$ evaluates to e_2 if e_1 is 0 and to e_3 otherwise.

Call-by-value operational semantics are used to interpret the language λ^F. It gives a set of rules. The judgments concluded are of the form $\Delta; \Gamma \vdash_F e : \tau$. Here, e is of type τ; Γ is a context assigning types to free variables of e and Δ is a context that contains free type variables of Γ, e and τ.

As an example (taken from [13]), a term that computes the factorial of 6 is given as:

$$(\textit{fix}\ \ f(n : int) : int.\ \ \textit{if}\,0(n, 1, n \times f(n-1)))\ 6$$

The first stage converts a term in λ^F to a CPS-style term. It identifies all intermediate evaluations. All control transfers like function invocation and return are achieved through a function call. Instead of returning a value, function calls invoke continuations that model the control operations. Hence, there is no need for the control stack. This phase produces a term in λ^K. The syntax of λ^K is given as:

$$\begin{array}{lll} \textit{types} & \tau ::= & \alpha \mid \textit{int} \mid \forall[\overrightarrow{\alpha}].(\overrightarrow{\tau}) \to \ \textit{void} \mid \langle \overrightarrow{\tau} \rangle \\ \textit{terms} & e ::= & v[\overrightarrow{\tau}](\overrightarrow{v}) \mid \textit{if}\,0(v, e_1, e_2) \mid \textit{halt}[\tau]v \\ & & \mid \textit{let}\ x = v\ \textit{in}\ e \\ & & \mid \textit{let}\ x = \pi_i(v)\ \textit{in}\ e \\ & & \mid \textit{let}\ x = v_1\ p\ v_2\ \textit{in}\ e \\ \textit{values} & v ::= & x \mid i \mid \langle \overrightarrow{v} \rangle \mid \textit{fix}\ x[\overrightarrow{\alpha}](x_1 : \tau_1, \ldots, x_k : \tau_k).e \end{array}$$

In the preceding definition of λ^K:

- The code in λ^K consists of a series of *let* bindings followed by the term except for the term *if*0, which is an expression having tree form.
- The only abstraction mechanism is *fix*.
- Type $\overrightarrow{\tau}$ represents a tuple $\langle \tau_1, \ldots, \tau_n \rangle$ and $\overrightarrow{v}$ represents a vector of values.
- The functions do not return a value. The function calls invoke continuations that model semantics of control operations. In other words, the function calls are jumps.
- The term $\textit{halt}[\tau]v$ is used to terminate the execution.

In [11], the authors use the CPS-type conversion based on [7]. It transforms a well-typed term in λ^F to a well-typed term in λ^K. The first phase of CPS conversion also optimizes the resulting code by removing tail recursion. The factorial example considered earlier gets translated to the following λ^K term [13]:

$$
\begin{aligned}
&(\mathit{fix}\ \ f\ \ [\,]\ \ (n : \mathit{int}, k : (\mathit{int}) \rightarrow\ \text{void})\\
&\quad \mathit{if}\ 0\ (n, k[\,](1),\\
&\quad \mathit{let}\ \ x = n - 1\ \ \mathit{in}\\
&\quad f[\,](x, \mathit{fix}\_[\,]\ \ (y :\ \mathit{int})\\
&\quad \mathit{let}\ \ z = n \times y\\
&\quad \mathit{in}\ \ k[\,](z))))\\
&[\,]\ \ (6,\ \mathit{fix}\ \ \_[\,]\ \ (n : \mathit{int}).\mathit{halt}[\mathit{int}]n)
\end{aligned}
$$

20.4.4.2 Closure Conversion

Closure conversion is a compilation technique used for compiling higher order languages to achieve a separation between code and data. A function with free variables is converted to a closure, which represents a piece of code for the function and a representation of its environment. Free variables in the body of a function are replaced by references to the environment and thus a function with free variables is replaced by a code abstracted on the environment parameter. The actual binding to the environment parameter is provided only when the function is applied to its arguments; until then, the application is delayed. Function calls are replaced by closures (invocation of the code part as well as values for the free variables).

The closure conversion stage here converts a well-formed λ^K term to a well-formed λ^C term. It is required that, after closure conversion, two functions with same types but different free variables should have closures with the same type. For this, during polymorphic instantiation, a copy of the code is made with type variables given their types. Before execution, types on the terms are erased and the copies can be represented by the same term. This is known as *type erasure* mechanism, which does not have any runtime cost; at the same time it prohibits some optimizations and it has side effects. The syntax of λ^C is the same as that of λ^K except for the fact that partial instantiation of types is considered as a value.

Closure conversion separates program code and data by making the closures explicit. Closure conversion consists of two steps:

1. *Closure conversion proper.* This step rewrites all function terms to their appropriate closures. As mentioned earlier, this is achieved through the type erasure mechanism. The closure is represented as a pair, one element of which denotes the code instantiated with the type environment and the other with the value environment. The type and value arguments of the original abstraction and value environment of the closure forms the input for the instantiated code.
2. *Hoisting.* Most of the work is done in the previous step. In this step, the closures are hoisted to the top level. After hoisting, programs get converted to terms similar to those in λ^C. In this target calculus, code is referred to by labels and code blocks are denoted by *letrec* prefix that binds labels to blocks. In this calculus, all *fix* expressions are replaced by new variables (labels) that are bound to the code blocks in the heap.

20.4.4.3 Allocation

The intermediate language λ^C contains a constructor for tuples. The space allocation and initialization of tuples is made explicit in this phase. The allocation for an (n-element) tuple is done by creating space for n elements. This step is followed by n initialization steps, one for each element of the tuple. This translation from λ^C to the target calculus λ^A adds initialization flags for each field of the tuple.

20.4.4.4 Code Generation

This stage converts terms in λ^A calculus to TAL syntax. Most of the TAL features are present in λ^A. TAL uses register names whereas λ^A uses α-varying variables. The registers are spilled into tuples and values from the tuple are loaded whenever required. The code-calling convention is made explicit in the code generation phase. Each code block is described by an entry condition that describes the types of values in the registers to be used in the code block. In this way, flow sensitivity is achieved in the system. Also, TAL makes the data layout of memory explicit by heap allocating tuples and code blocks.

20.4.4.5 TAL Syntax

The TAL syntax in [13] is given later. A TAL program consists of a heap H, a register file R and a sequence of instructions I. The heap is a mapping from labels (which are word values) to code blocks and tuples. The heap values do not need to be word values. The notation $H\{l \mapsto h\}$ indicates that the binding of l is replaced by binding h in the heap. The word values are labels, integers, existential packages and junk values (represented as $?\tau$). The instruction set has *jmp* and *halt* instructions along with other instructions like arithmetic instructions (*add*, *sub*, etc), load and store instructions and *unpack* instruction for evaluating packages.

types	τ	$::=$	$\alpha \mid int \mid \forall[\vec{\alpha}].\Gamma \mid \langle \tau_1^{\varphi_1}, \ldots, \tau_n^{\varphi_n} \rangle \mid \exists\alpha.\tau$
initialization flags	φ	$::=$	$0 \mid 1$
heap types	Ψ	$::=$	$\{l_1 : \tau_1, \ldots, l_n : \tau_n\}$
register file types	Γ	$::=$	$\{r_1 : \tau_1, \ldots, r_n : \tau_n\}$
type contexts	Δ	$::=$	$\vec{\alpha}$
registers	r	$\in$	$\{\mathtt{r1}, \mathtt{r2}, \mathtt{r3}, \ldots\}$
word values	w	$::=$	$l \mid i \mid ?\tau \mid w[\tau] \mid pack[\tau, w]\ \ as\ \tau'$
small values	v	$::=$	$r \mid w \mid v[\tau] \mid pack[\tau, v]\ \ as\ \tau'$
heap values	h	$::=$	$\langle w_1, \ldots, w_n \rangle \mid \mathtt{code}[\vec{\alpha}]\Gamma.S$
heaps	H	$::=$	$\{l_1 \mapsto h_1, \ldots, l_n \mapsto h_n\}$
register files	R	$::=$	$\{r_1 \mapsto w_1, \ldots, r_n \mapsto w_n\}$
instructions	ι	$::=$	$\mathtt{add}\ r_d, r_s, v \mid \mathtt{bnz}\ r, v \mid \mathtt{ld}\ r_d, r_s[i]$
			$\mid \mathtt{malloc}\ r_d[\vec{\tau}] \mid \mathtt{mov}\ r_d, v \mid \mathtt{mul}\ r_d, r_s, v$
			$\mid \mathtt{sto}\ r_d[i], r_s \mid \mathtt{sub}\ r_d, r_s, v \mid \mathtt{unpack}\ [\alpha, r_d], v$
instruction sequences	S	$::=$	$\iota; S \mid \mathtt{jmp}\ v \mid \mathtt{halt}[\tau]$
programs	P	$::=$	(H, R, S)

20.4.4.6 TAL Semantics

The detailed TAL operational semantics are given in [11]. The term represented as $(H, R\{r1 \mapsto w\}, halt[\tau])$ denotes a machine state with the result of the evaluation in $r1$. This is a term for which the evaluation terminates. For all other (terminal) configurations, the computation is said to be stuck.

Two instructions of TAL that are not supported in most of the machine languages are *unpack* and *malloc*. If v is of the form $pack\ \ [\tau', v']\ \ as\ \ \tau$, then the instruction *unpack* $[\alpha, r_d]$ substitutes τ' for α in the remaining code sequence. The *malloc* instruction *malloc* $r_d[\tau_1, \ldots, \tau_n]$ allocates memory for a tuple. The register r_d is bound to the new tuple. The instruction *malloc* can be considered as an instruction that can be expanded into an (abstract) sequence of instructions allocating the required space.

The static semantics of TAL [11] ensures the type safety of TAL programs by proving that well-formed TAL programs always reduce to the terminal configuration $(H, R\{r1 \mapsto w\}, \mathit{halt}[\tau])$.

The compiler thus produces a well-typed TAL term starting from a well-typed λ^F term.

20.4.4.7 Stack-Based Typed Assembly Language

As seen earlier, continuation passing style is used in compilation of higher order languages to eliminate the control stack. Stack-based typed assembly language (STAL) is an extension of TAL [12], which provides support for stack types and constructs. It can support Java, Pascal and ML. It provides typing support for procedure calling. Also, optimizations like tail recursion optimization can be handled by STAL. The work in [12] also explores the similarities in continuation passing style and stack-based evaluation. The authors conclude that both styles are essentially similar with minor differences and can be transformed from one form to the other.

20.4.5 Type-Safe Linking

To ensure the type safety of the code, it is desirable that a safety check is done during linking also. The work presented in [5] describes a set of inference rules to guarantee the safety of linking. To do this, an object file calculus known as MTAL has been developed by Glew and Morrisett [5]. It supports abstract types and higher level (abstract) type constructors. It models real-life linkers by making low-level tasks explicit. The notion of link compatibility is built in MTAL. Dynamic linking support is not provided by MTAL, but it can be extended to do that. This requires a balance of (type) information sharing between program control and OS control. Extensible systems can be type checked during linking for safety.

20.4.6 Use of Typed Assembly Language Framework

TAL framework is also useful in other safety checking methods such as PCC. PCC system uses first-order logic to encode the operational semantics of the type systems. The TAL system provides a higher level abstraction mechanism but is less expressible than the PCC system. TAL abstractions can be useful to provide compact encoding for a PCC-like system. The TAL framework is also used in certifying compilation and runtime code specialization. The TAL framework provides a formal framework to reason about the object code generation in runtime specialization [9]. It has also been found that the use of TAL assembly language reduces the work to be done by the type-state system described later in the chapter.

20.5 Proof Carrying Code

PCC [16] is another static method for ensuring mobile code safety. In this method, the code consumer publishes a safety policy that is assumed to assure the safety of the consumer's system. The code producer considers this safety policy, and supplies with the code that is produced a proof or certificate that the code respects the safety policy. The code that arrives at the consumer's site comes with the safety policy bundled along with it. The consumer then checks the supplied proof and ensures that the proof is valid with respect to the mobile code as well as the published policy.

The technique is well grounded in formal methodology and draws heavily on language theory (types for representation of the proof and type checking for checking the validity of the proof) and logic (the specification mechanism to specify the safety policy and the proof).

This method of ensuring code safety is attractive for a number of reasons:

1. It does not need the intervention of any third party to ensure trust between the code producer and the customer.
2. The method is fairly lightweight — the proof production may be done off-line. The validity of the proof is checked on-line by an inexpensive type-checking method.

3. The method is intrinsically tamperproof. If the proof is mutated during the transfer process from the producer to the consumer, then it would no longer be valid for the code along with which it is transmitted; if the code is tampered with, then again, the proof would not be valid for the code; and if both proof and code are modified in such a way that the proof is still valid for the code and the safety policy, then no harm would be caused to the consumer's machine by executing the code.
4. The method is based on sound formal principles including types, type checking and logic.

This method has been demonstrated in at least two applications. In [17] it is shown how PCC can be used to check the safety of network packet filters written by the user to filter out unwanted packets from arriving at a network node. PCC has also been used to check the safety of extensions to the runtime system of a functional programming language like ML [16]. More details on these applications of PCC are supplied later in the following sections.

20.5.1 Overview of Proof Carrying Code

Figure 20.1 shows a diagram that gives an overview of the PCC method. This diagram is taken from [16].

The code consumer publishes a safety policy that is essentially a collection of rules that any code running on the producer's site should adhere to for ensuring the safety of the consumer's machine resources. The safety policy has two components: safety rules and an interface. The safety rules are rules that the foreign code should follow. This is similar to type rules provided with the type system of a language. The interface defines the calling conventions to be followed between the code consumer

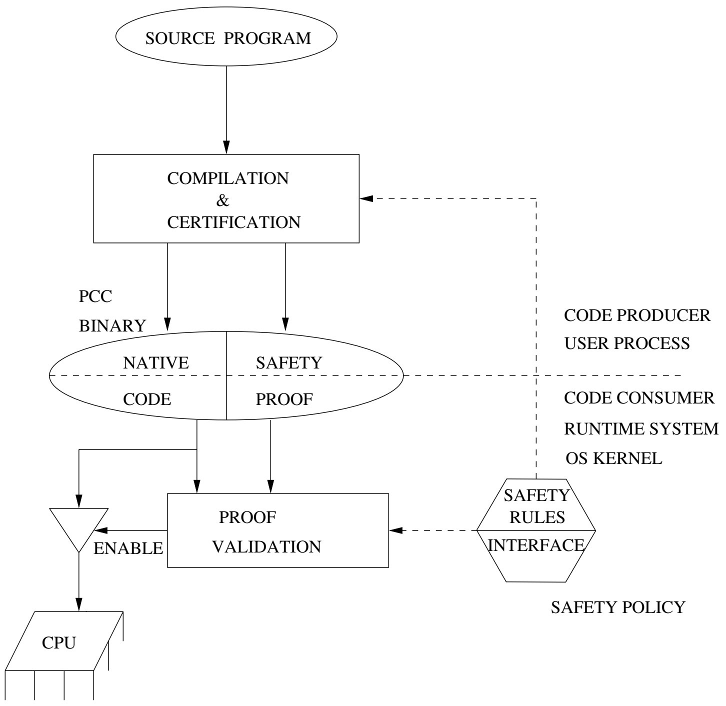


FIGURE 20.1 Overview of proof-carrying code.

and the foreign code. These are the conventions that have to be followed by the foreign code when it invokes any function on the consumer's site. These conventions also define how the code consumer can invoke the foreign code that is supplied by the producer. The interface is analogous to function signatures or prototype declarations in a language like C.

There are three stages in the PCC life cycle. The first stage is called *certification*. In this stage, the code producer studies the consumer's safety policy and produces a proof that the code supplied satisfies the safety policy. This stage is similar to program verification and can be facilitated if an (operational) semantics of the machine language exists in which the code downloaded by the consumer can be written. This stage can be done off-line and is thus not in the critical path of the PCC life cycle. At the end of this stage the PCC binary is available with the code producer, which is essentially the code produced by the producer together with the proof of safety.

The code consumer examines this PCC binary in the next stage, which is called *validation*. In this stage the code consumer validates that the safety proof supplied along with the PCC binary is actually a valid proof. The method used here is similar to type checking in languages and is relatively less time consuming.

In the final stage of the PCC life cycle, the code consumer executes the foreign code whose safety proof has been validated in the validation phase. It is possible that the foreign code may be executed a number of times, thereby amortizing the cost of validation over the many runs of the foreign code.

In the next few sections we shall explain PCC using the same example that is used in [17]. This is the example in which PCC is used to check the safety of user written network packet filters.

Usually packet filters are written using the Berkeley packet filter approach in which the filter is written in a specialized language [10]. This language is a restricted accumulator-based language in which backward jumps in the code are prevented. The interpreter explaining the filter written in this language ensures that only valid instructions as defined in the language are used. In this way it is ensured that the filter application uses only the packet memory and the scratch memory of the system.

In the PCC approach to writing packet filters as described in [17], an unsafe assembly language, the DEC Alpha assembly language, is used to code the packet filter. The PCC approach is then used to ensure the safety of this application with respect to a published safety policy at the code consumer's site.

20.5.2 DEC Alpha Abstract Machine Specification for Verification Condition Generation

We begin this subsection by describing an abstract specification of the instructions of the DEC Alpha assembly language. In the example described in [17], the subset of instructions of the processor considered is shown in Table 20.3. In this table n refers to a constant and r_i refers to an Alpha register. Because Alpha is a 64-b machine, the operations are on 64-b operands. The instructions considered are ADDQ, SUBQ, AND, OR, SLL, SRL, BEQ, BNE, BGE, BLT, LDQ, STQ and RET.

The semantics of this subset are given by the operational semantics of the instructions as shown in Table 20.4. In this table, the notation Π represents the program that consists of the instructions in the subset. The semantics are defined by a transition system with each state having two components:

TABLE 20.3 Subset of DEC Alpha Instructions Considered in the Example

op	$::=$	$n \mid r_i \; i \in 0 \ldots 10$
al	$::=$	$ADDQ \mid SUBQ \mid AND \mid OR \mid SLL \mid SRL$
br	$::=$	$BEQ \mid BNE \mid BGE \mid BLT$
$instr$	$::=$	$LDQ\; r_d, n(r_s) \mid STQ\; r_s, n(r_d) \mid al\; r_s, op, r_d \mid br\; r_s, n \mid RET$

TABLE 20.4 Semantics of the Subset of DEC Alpha Instructions Considered

$$(\rho, pc) \rightarrow \begin{cases} (\rho[r_d \leftarrow r_s \oplus op], pc+1), \ \textit{if} \ \prod_{pc} = ADDQ\ r_s, op, r_d \\ (\rho[r_d \leftarrow sel(r_m, r_s \oplus n], pc+1), \\ \quad \textit{if} \ \prod_{pc} = LDQ\ r_d, n(r_s) \ and \ \underline{rd(r_s \oplus n)} \\ (\rho[r_m \leftarrow upd(r_m, r_d \oplus n, r_s)], pc+1), \\ \quad \textit{if} \ \prod_{pc} = STQ\ r_s, n(r_d) \ and \ \underline{wr(r_d \oplus n)} \\ (\rho, pc+n+1), \ \textit{if} \ \prod_{pc} = BEQ\ r_s, n \ and \ r_s = 0 \\ (\rho, pc+1), \ \textit{if} \ \prod_{pc} = BEQ\ r_s, n \ and \ r_s \neq 0 \end{cases}$$

ρ is the state of the machine and pc is the value of the program counter. The notation $\rho[r_i]$ refers to the value of register r_i in the state of the machine ρ. The notation $\rho[r_d \leftarrow r_d \oplus 1]$ refers to the state obtained by replacing the contents of destination register r_d with the result of the computation $r_d \oplus 1$. The notation $\oplus$ stands for two's-complement addition on 64-b operands. This operation is defined as $e_1 \oplus e2 = (e_1 + e_2)\ mod\ 2^{64}$. Memory is represented by a register r_m. To select the value at a particular memory location, a, the notation used is $sel(r_m, a)$. To write into a memory location, the notation used is $upd(r_m, a, r_s)$, which means that memory location at address a is updated using the contents of register r_s. Memory operations also work on 64-b operands and the address is aligned to 8-byte boundaries.

In Table 20.4, the underlined terms represent extensions to the DEC Alpha semantics to incorporate safety checks. The predicate $rd(a)$, when true, indicates that it is safe to read from memory location a. This means that the address is aligned to an 8-byte boundary. Similarly, the predicate $wr(a)$, when true, indicates that it is safe to both read and write to memory location a.

The semantics define that when either $rd(a)$ or $wr(a)$ is false, the machine execution halts. In this semantics, the goal of safety checking is to ensure that the machine does not halt due to the failure of these safety checks.

Before we describe the certification for the packet filter example, let us illustrate the process using a small application from [17]. This user application accesses a table in the OS kernel that contains entries having data on user processes. The kernel provides an interface to the user program to access this kernel. Essentially, the interface provides, in a designated register r_0, the base address of the table that the user application can access. Further, the kernel guarantees that the table base address contained in r_0 is a valid address.

Each entry in the table consists of two fields. The first field is a tag that defines permissions for the user application. The second field contains the data that the application may want to access. The tag field defines the access permissions for the application to access the corresponding data field that it tags.

The safety policy specifies the following clauses:

1. The application should not access any table entries other than those in the table pointed to be r_0.
2. The application cannot modify the permission specified by the tag for a given field. In other words, tags are read-only.
3. A particular data item may be written into only if its corresponding tag value is not zero. Otherwise, it can be only read.
4. The code should not modify reserved and callee-save registers.

The safety policy described in the previous paragraph can be encoded in a predicate *Pre*, which may be defined in predicate logic shown as follows:

$$Pre \equiv r_0\ mod\ 2^{64} = r_0 \wedge rd(r_0) \wedge rd(r_0 \oplus 8) \wedge sel(r_m, r_0) \neq 0 \Rightarrow wr(r_0 \oplus 8) \tag{20.1}$$

Pre represents the precondition that has to be satisfied for the application to execute on the consumer's machine. In general, a postcondition may also be satisfied that represents the invariant that has to be satisfied when the code finishes executing. In the current example, the postcondition is trivially *True*.

The certification phase of the PCC methodology now consists of the following steps:

1. Given the program whose safety has to be checked, use the abstract machine specification and *Pre* to generate a predicate formula. The predicate thus computed is called a safety predicate. This step is called *verification condition generation (VC generation).*
2. Generate a proof of the validity of the formula computed in the previous step.

The verification condition generation proceeds on a Floyd-style verification approach. For each instruction in the program a verification condition (VC) or invariant is generated. For an instruction at the current program counter (*pc*) value, its verification condition is denoted by VC_{pc}. VC generation proceeds backward in the program (i.e., VC_{pc} is computed from the *VC* of the next instruction VC_{pc+1}). The VC at the beginning of the program is denoted by VC_0. It may be noted, at least for the example under consideration, that all branches are restricted to be in the forward direction only. Table 20.5 describes the rules used in verification condition generation for the subset of instructions that we consider in the DEC Alpha machine. These rules are obtained from the abstract machine specification given in Table 20.4. As before, the notation $P[r_d \leftarrow r_s \oplus op]$ stands for the predicate obtained from P by substituting $r_s \oplus op$ for r_d.

Once the VCs for the instructions in the program have been computed and, in particular, VC_0 has been computed, the safety predicate that has to be proved to ensure the safety of the program consistent with the published safety policy is given by:

$$SP\ (\Pi, Pre, Post) = \forall r_0, \ldots, \forall r_k \forall r_m.\ Pre \ \Rightarrow \ VC_0$$

where k is the number of machine registers that are considered for the example.

A valid proof for the preceding formula implies that program Π starts from an initial state in which *Pre* is satisfied and, if it terminates, it terminates in a state in which *Post* is satisfied. As already mentioned, successful termination of the program implies that it passes the safety checks prescribed.

The application that we consider is a DEC Alpha program that accesses a data field in the process table, increments the value of the field and writes back the updated value if the tag corresponding to the fields permits write back. As is the case with this complete discussion, this example also is taken from [17]. The program to accomplish the task is shown in Table 20.6.

TABLE 20.5 Rules to Compute the Verification Conditions for the DEC Alpha Subset

$$VC_{pc} = \begin{cases} VC_{pc+1}[r_d \leftarrow r_s \oplus op],\ \textit{if} \prod_{pc} = ADDQ\ r_s, op, r_d \\ rd(r_s \oplus n) \wedge VC_{pc+1}[r_d \leftarrow sel(r_m, r_s \oplus n], \\ \quad \textit{if} \prod_{pc} = LDQ\ r_d, n(r_s) \\ (wr(r_d \oplus n) \wedge VC_{pc+1}[r_m \leftarrow upd(r_m, r_d \oplus n, r_s)], \\ \quad \textit{if} \prod_{pc} = STQ\ r_s, n(r_d) \\ r_s = 0 \Rightarrow VC_{pc+n+1} \wedge (r_s \neq 0 \Rightarrow VC_{pc+1}),\ \textit{if} \prod_{pc} = BEQ\ r_s, n \\ Post,\ \textit{if} \prod_{pc} = RET \end{cases}$$

TABLE 20.6 DEC Alpha Assembly Language Program Used in the Example

1	*ADDQ* $r_0,\ 8,\ r_1$	*% Address of data in* r_0
2	*LDQ* $r_0,\ 8(r_0)$	*% Data in* r_0
3	*LDQ* $r_2,\ -8(r_1)$	*% Tag in* r_2
4	*ADDQ* $r_0,\ 1,\ r_0$	*% Increment Data in* r_0
5	*BEQ* $r_2,\ l_1$	*% Skip if tag* $== 0$
6	*STQ* $r_0,\ 0(r_1)$	*% Write back data*
l_1	*RET*	

The verification conditions and hence VC_0 for this program may be computed using the verification condition rules given in Table 20.5. *Pre* is given by Equation (20.1) and *Post* is assumed to be *True*. The safety predicate for this assembly language program as given in [17] is:

$$SP_r = \forall r_0.\forall r_m.Pre \;\Rightarrow\; rd(r_0 \oplus 8) \wedge\; rd(r_0 \oplus 8 \ominus 8) \wedge\; sel(r_m, r_0 \oplus 8 \ominus 8) = 0 \;\Rightarrow\; true \\ \wedge\; sel(r_m, r_0 \oplus 8 \ominus 8)\ neq\ 0 \;\Rightarrow\; wr\ (r_0 \oplus 8) \tag{20.2}$$

The proof for this safety predicate is supplied with the PCC binary for the assembly language program given in Table 20.6. The safety predicate expresses the safety condition that for all values of register r_o and states of memory r_m satisfying the precondition *Pre*, the memory locations $r_0 \oplus 8$ and $(r_0 \oplus 8 \ominus 8)$ must be readable; and, if the tag value at address $r_0 \oplus 8 \ominus 8$ is readable, then its corresponding data must be writable.

The safety predicate is proved by using rules of first-order predicate logic extended with the two-complement integer arithmetic. The proof for the safety predicate for the current example and which is given in Equation (20.2) may be expressed as shown in Table 20.7. In this table, the *extract(Pre)* operation extracts a conjunct from *Pre* and returns it as the term within the angular brackets $\langle \ldots \rangle$. The u is a hypothesis introduced in the proof of the predicate $sel\ (r_m, r_0 \oplus 8 \ominus 8) \neq 0$. Otherwise, the proof is fairly easy to understand.

In [17] it is mentioned that this proof was generated by a theorem prover written by the authors that sometimes required manual intervention, especially to reason about the two-complement integer arithmetic. It is, however, not difficult to envisage a theorem prover that can automatically generate such a proof.

Having studied how the safety predicate is computed and the proof is produced, let us return to the network packet filter application. The safety policy criteria may be enumerated as follows:

1. Memory reads are restricted to the packet and scratch memory.
2. Memory writes are limited to the scratch memory.
3. All branches are forward.
4. Reserved and callee-saves registers are not modified.

There safety criteria are derived from those safety restrictions that are imposed by the BSD packet filter approach.

In [17], the packet filter approach is written assuming that the return value is in r_0, the aligned address of the packet is given in register r_1, the length of the packet is given in r_2 and the address of a

TABLE 20.7 Steps in the Proof of the Safety Predicate for the DEC Alpha Assembly Language Program

Step	*Conjunction Extraction/Assumption and Inference*
1	$\langle rd(r_0)\rangle = extract(Pre)$
2	$\langle r_0\ mod\ 2^{64} = r_0\rangle = extract(Pre)$
3	$\langle sel(r_m, r_0) \neq 0 \Rightarrow wr(r_0 \oplus 8)\rangle = extract\ (Pre)$
4	$\langle r_0\ mod\ 2^{64} = r_0\rangle = extract\ (Pre)$
5	$r_0\ mod\ 2^{64} = r_0\ (step\ 2)$
	Inference : $r_0 = r_0 \oplus 8 \ominus 8$
6	$rd(r_0)\ (step\ 1) \wedge r_0 = r_0 \oplus 8 \ominus 8\ (step\ 5)$
	Inference : $rd(r_0 \oplus 8 \ominus 8)$
7	u
	Inference : $sel(r_m, r_0 \oplus 8 \ominus 8) \neq 0$
8	$r_0\ mod\ 2^{64} = r_0\ (step\ 4)$
	Inference : $r_0 = r_0 \oplus 8 \ominus 8$
9	u
	Inference : $sel(r_m, r_0 \oplus 8 \ominus 8) \neq 0$
10	$sel(r_m, r_0 \oplus 8 \ominus 8) \neq 0\ (step\ 9) \wedge r_0 = r_0 \oplus 8 \ominus 8\ (step\ 8)$
	Inference : $sel(r_m, r_0) \neq 0$
11	$sel(r_m, r_0) \neq 0 \Rightarrow wr(r_0 \oplus 8)\ (step\ 3) \wedge sel(r_m, r_0) \neq 0\ (step\ 10)$
	Inference : $wr(r_0 \oplus 8)$
12	$wr(r_0 \oplus 8)\ (step\ 11)$
	Inference : $sel(r_m, r_0 \oplus 8 \ominus 8) \neq 0 \Rightarrow wr(r_0 \oplus 8)$
13	$rd(r_0 \oplus 8 \ominus 8)\ (step\ 6) \wedge sel(r_m, r_0 \oplus 8 \ominus 8) \neq 0 \Rightarrow wr(r_0 \oplus 8)\ (step\ 12)$
	Inference : $rd(r_0 \oplus 8 \ominus 8) \wedge (sel(r_m, r_0 \oplus 8 \ominus 8) \neq 0 \Rightarrow wr(r_0 \oplus 8)) \wedge \ldots$
14	$rd(r_0 \oplus 8 \ominus 8) \wedge (sel(r_m, r_0 \oplus 8 \ominus 8) \neq 0 \Rightarrow wr(r_0 \oplus 8)) \wedge \ldots\ (step\ 13)$
	Inference: $Pre \Rightarrow rd(r_0 \oplus 8 \ominus 8) \wedge (sel(r_m, r_0 \oplus 8 \ominus 8) \neq 0 \Rightarrow wr(r_0 \oplus 8)) \wedge \ldots$
15	$Pre \Rightarrow rd(r_0 \oplus 8 \ominus 8) \wedge (sel(r_m, r_0 \oplus 8 \ominus 8) \neq 0 \Rightarrow wr(r_0 \oplus 8)) \wedge \ldots (step\ 14)$
	Inference: $\forall r_0.\forall Pre \Rightarrow rd(r_0 \oplus 8 \ominus 8) \wedge (sel(r_m, r_0 \oplus 8 \ominus 8) \neq 0 \Rightarrow wr(r_0 \oplus 8)) \wedge \ldots$

16-byte aligned scratch memory is given in r_3. The packet length is assumed to be at least 64 bytes. Then the safety predicate for this set of criteria may be expressed as the precondition:

$$\begin{aligned}Pre = {} & (r_1\ \ mod\ \ 2^{64} = r_1) \wedge (r_2\ mod\ 2^{64} = r_2) \wedge (r_2 < 2^{63}) \wedge (r_2 \geq 64)\\ & \wedge (r_3\ \ mod\ \ 2^{64} = r_3) \wedge (\forall i.(i \geq 0 \wedge i < r_2 \wedge (i\&7) = 0) \Rightarrow rd(r_1 \oplus i))\\ & \wedge (\forall j.(j \geq 0 \wedge j < 16 \wedge (j\&7) = 0) \Rightarrow wr(r_3 \oplus j))\\ & \wedge (\forall i.\forall j.(i \geq 0 \wedge i < r_2 \wedge j \geq 0 \wedge j < 16) \Rightarrow (r1 \oplus i \neq r_3 \oplus j))\end{aligned}$$

The first five conjuncts restrict the range of values the registers can take. The next two conjuncts specify the addresses from which it is safe to read and addresses to which it is safe to write. The last term merely states that the packet memory and the scratch memory should be distinct. The postcondition is taken to be *true*.

20.5.3 Second Example

At this point we include another canonical example that is found in the literature. This example is taken from [16] and deals with the ML type system. ML has a type system for which it is possible to ensure the safety of ML programs. However, in many practical situations, not all the components of a software project need be written in ML. It is possible that some of the components may be written in a language such as C and hence these components fall outside the ambit of the ML type system. Thus, a problem exists for ensuring that these components respect the invariants associated with the ML heap.

The example considered in [16] is that of an ML type that may be an integer or a pair of integers. The foreign code computes the sum of the elements of a list where each element of the list can be an integer or a pair of integers. The code to do this is shown as follows and is taken from [16]:

```
datatype T = Int of int | Pair of int * int


fun sum(l : T list) =
  let
     fun foldr f nil a = a


    | foldr f (h::t) a = foldr f t (f(a, h))
in
   foldr (fn (acc, Int i) => acc + i
          | (acc, Pair (i, j)) => acc + i + j)
        l 0
end
```

This code fragment defines the way to compute the sum for an integer as well as a pair of integers. The sum is accumulated in **acc**. The example describes how to frame a safety policy for extensions to the runtime system of the TIL compiler for Standard ML [21]. It then shows how the hand-optimized version of the DEC Alpha assembly code for the preceding program may be proved to be safe with respect to this safety policy.

Consider the data-type declarations shown in Table 20.8. The heap structure for these data-type declarations is shown in Figure 20.2.

An integer value is represented as a 32-b machine word. A pair of integer values is represented as consecutive locations containing these values. A type that represents the union of these two

TABLE 20.8 Examples of Data Type Declarations

val	$r0$	:	$int = 5$
val	$r1$	:	$int * int = (2, 3)$
val	$r2$	:	$T = Pair\, r1$
val	$r3$	:	$T = Int\, 6$
val	$r4$	:	$T\ list = [r3, r2]$

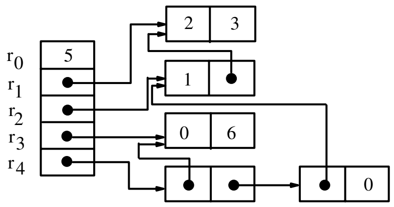


FIGURE 20.2 Data representation in TIL. Each box represents a machine word.

types (i.e., it can be either an integer or a pair of integers) is represented as a pointer to a pair of locations. The first location in the pair contains a value that indicates whether the type represented is an integer or a pair of integers. If the value of this field is 0, then it indicates that this is an integer type and the integer is contained in the next location. If the value of this field is 1, then it indicates that this is a type that represents a pair of integers and the next location contains a pointer to the pair of integers. The representation for a list of values each of which may be an integer or a pair of integers is also shown in Figure 20.2. The empty list is represented by a zero.

The layout of the heap may also be specified by typing judgments each of which is of the form $m \vdash e : \tau$ where m is the state of the memory, e is the expression under consideration and τ is the type that may be assigned to the expression. The expressions and memory states considered are summarized as follows:

$$\begin{aligned} e &::= n \mid r_i \mid sel(m, e) \mid e_1 + e_2 \\ m &::= r_m \mid upd(m, e_1, e_2) \end{aligned} \tag{20.3}$$

Here, as usual, r_i is a DEC Alpha machine register, r_m summarizes the state of the memory, $sel(m, e)$ denotes the contents of location e in memory state m and $upd(m, e_1, e_2)$ denotes the memory state obtained by updating the contents of location e_1 with the contents of location e_2.

The typing rules that define the layout of the heap are shown in Table 20.9. These rules are defined in [16]. These typing rules use the expression *addr*. This means that the expression value is a memory address whose contents can be read.

The DEC Alpha assembly language program whose safety has to be verified is shown in Table 20.10. This program accumulates the sum of integer values or pairs of integer values depending on the type. It is assumed that r_0 contains the argument of type *T list* the sum of whose elements is computed by the program. It is also assumed that the final sum is stored in register r_0. Registers r_1, r_2 and r_3 are stored as temporaries.

In the assembly code shown in Table 20.10, *INV* refers to an invariant that must be maintained at that program point. If *Inv* denotes the indices of the invariants, then Inv_0 represents the precondition that the program has to satisfy before it starts executing. In this case, the precondition is $r_m \vdash r_0 : T\ list$. The postcondition is $r_m \vdash r_0 : int$. An invariant also occurs at line number 2. This invariant is $r_m \vdash r_0 : T\ list \wedge r_m \vdash r_1 : int$. In general, loop invariants can be obtained by considering the registers that are live at that point.

TABLE 20.9 Typing Rules Defining the Heap Layout

$\dfrac{m \vdash e : \tau_1 * \tau_2}{m \vdash e : addr \wedge m \vdash e + 4 : addr \wedge m \vdash sel(m, e) : \tau_1 \wedge m \vdash sel(m, e + 4) : \tau_2}$
$\dfrac{m \vdash e : \tau_1 + \tau_2}{\begin{array}{l} m \vdash e : addr \wedge m \vdash e + 4 : addr \wedge sel(me) = 0 \supset m \vdash sel(m, e + 4) : \tau_1 \\ \quad \wedge\ sel(m, e) \neq 0 \supset m \vdash sel(m, e + 4) : \tau_2 \end{array}}$
$\dfrac{m \vdash e : \tau\ list\ e \neq 0}{m \vdash e : addr \wedge m \vdash e + 4 : addr \wedge m \vdash sel(m, e) : \tau \wedge m \vdash sel(m, e + 4) : \tau\ list}$
$\dfrac{m \vdash e_1 : int\ m \vdash e_2 : int}{m \vdash e_1 + e_2 : int}$
$\dfrac{}{m \vdash 0 : int}$

TABLE 20.10 DEC Alpha Assembly Program to Compute the Sum of Elements in List

0	*sum*:	*INV*	$r_m \vdash r_0 : T\ list$	*%r_0 is l*
1		*MOV*	$r_1,\ 0$	*%r_1 is acc*
2	L_2	*INV*	$r_m \vdash r_0 :\ T\ list \wedge r_m \vdash r_1 :\ int$	*%Intialize acc* *%Loop invariant*
3		*BEQ*	$r_0,\ L_{14}$	*%Is list empty?*
4		*LD*	$r_2,\ 0(r_0)$	*%Load head*
5		*LD*	$r_0,\ 4(r_0)$	*%Load tail*
6		*LD*	$r_3,\ 0(r_2)$	*%Load constructor*
7		*LD*	$r_2,\ 4(r_2)$	*%Load data*
8		*BEQ*	$r_3,\ l_{12}$	*%Is an integer?*
9		*LD*	$r_3,\ 0(r_2)$	*%Load i*
10		*LD*	$r_2,\ 4(r_2)$	*%Load j*
11		*ADD*	$r_2,\ r_3,\ r_2$	*%Add i and j*
12	L_{12}	*ADD*	$r_1,\ r_2,\ r_1$	*%Do the addition*
13		*BR*	L_2	*%Loop*
14	L_{14}	*MOV*	$r_0,\ r_1$	*%Copy result in r_0*
15		*RET*		*%Result is in r_0*

TABLE 20.11 Rules to Compute the Verification Conditions for ML Type Example

$$VC_i = \begin{cases} VC_{i+1}[r_d \leftarrow r_s \oplus op],\ \text{if } \prod_i = ADDQ\, r_s, op, r_d \\ r_m \vdash r_s + n :\ addr \wedge\ VC_{i+1}[r_d \leftarrow\ sel\,(r_m, r_s \oplus n)], \\ \quad \text{if } \prod_i =\ LD\, r_d, n(r_s) \\ r_s = 0 \supset\ VC_{i+n+1} \wedge (r_s \neq\ 0 \supset\ VC_{i+1}),\ \text{if } \prod_i =\ BEQ\, r_s, n \\ Post,\ \text{if } \prod_i =\ RET \\ I,\ \ \text{if } \prod_i =\ INV\, I \end{cases}$$

As described in the earlier discussion on the network packet filter example, the verification condition generator generates the verification condition at each program point. Table 20.11 shows the rules for computing the verification conditions for our current example. These rules are also from [16]. The restriction is that for the computation of the verification conditions for a program with loops there has to be an invariant in every loop.

Now, the safety predicate may be expressed as:

$$VC\,(\Pi,\ Inv,\ Post) = \forall r_i.\ \bigwedge_{i \in Inv} Inv_i \supset VC_{i+1}$$

Then for the current example, the safety predicate takes the form: $r_m \vdash r_0 :\ Foo\ list \supset (r_m \vdash r_0 :\ Foo\ list \wedge r_m \vdash 0 :\ int)$.

We have now reached the end our discussion of the certification stage. The next stage in the PCC life cycle is proof validation.

20.5.4 Proof Validation

At the consumer's site, the safety predicate is computed using the VC rules. Then the supplied proof is examined to see if it is valid. The PCC system uses the Edinburgh logical framework (LF) [6] to represent the predicates and the proofs. LF is an ML for the expression of high-level specifications in

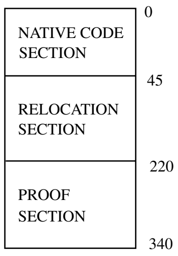


FIGURE 20.3 The layout of the PCC binary for the resource access example. The offsets are in bytes.

logic. The advantage of the LF representation is that checking the validity of the proof amounts to type checking its representation in the LF framework. In other words, the validity of the proof is implied by the well-typedness of the proof representation. The type-checking algorithm is fairly simple and its implementation is usually small. Thus, the code consumer only needs to trust the correctness of this small implementation to be assured of the safety of the foreign code. This also implies that proof validation is fairly lightweight in terms of the execution time that it takes; and because this step is in the critical path of the download and execute process, this is an added advantage. Some details on the LF representation and a theorem expressing the adequacy of the LF framework to represent proofs is described in [16].

20.5.5 Proof Carrying Code Binary

The PCC binary typically has three sections as illustrated in Figure 20.3.

One section is the native code section that contains the native code that may be mapped to memory and executed. Then there is a symbol table that contains the information to reconstruct the LF representation of the safety proof by the code consumer. Also the binary encoding of the LF representation of the safety proof also exists. More work needs to be done on slim representations for PCC binaries.

20.6 Safety Checking of Machine Code Using Type-State Checking

We now give a brief description of yet another static method for safety checking. This section is based on the work done by Xu, Miller and Reps [25, 26] who have developed a safety-checking technique known as *type-state checking* that relies neither on restrictions on the source language nor on the correctness of the compiler for safety checking. Essentially, this technique allows a programmer to write the code in any language of choice and to use any compiler of choice. The safety conditions are enforced on the machine code. As long as the machine code is not doing anything "bad" on a system on which it is running, it can be assumed to be safe. This approach has uncovered a lot of possibilities because safety checking is decoupled from the expressive power of the source language. Also, the need to depend on the correctness of a compiler is alleviated. Safety properties are specified with respect to the host (on which the potential unsafe code can run) and hence they can be extensible. Most of the static safety-checking techniques use traditional type checking to infer the safe or unsafe nature of the code. Traditional type checking is a flow-insensitive technique. Type-state checking, on the other hand, is a flow-sensitive technique. While analyzing an operation for safety, it not only checks types of operands but also their states (hence, the name *type state*). This is important because conditions under which an operation is safe or unsafe may depend not only on the types of operands but also on their states. For example, dereferencing an initialized pointer is allowed, but not an uninitialized pointer. Initialization or uninitialization is the state of an operand, and this state information might help in getting more accurate safety-checking results. This implies that the safety-checking algorithm should also take into consideration data flow properties associated with each operand. The type-safe-checking algorithm uses standard data flow techniques (more precisely, abstract interpretation techniques), to derive the machine state at each program point. As the safety checking is done using flow-sensitive type-state information, it provides a finer granularity to the safety-checking algorithm.

20.6.1 Overview of Type State Checking of Machine Code

In essence, the type-state checking of machine code involves the following steps:

1. Extracting the source level type (i.e., type state) information from the machine code based on annotations about initial inputs to the untrusted code
2. Determining whether the untrusted code is safe by applying techniques (or a combination of techniques) some of which were initially developed for program verification

To extract the source level type-state information, first the global data are annotated in the trusted host. Starting with the initial memory state at the entry point of the untrusted code, type-state analysis abstractly interprets the untrusted machine code to produce a safe and approximate machine state at each program point. The memory states at each program point are described in terms of type-states and linear constraints. The linear constraints are linear equalities and inequalities that are combined with $\wedge$, $\vee$, $\neg$ and the quantifiers $\exists$ and $\forall$. By using the type-state information obtained in the previous step, the control flow graph of the untrusted code is traversed to annotate each instruction with local and global safety conditions and assertions. The conditions that can be checked by using type-state information alone are called local safety conditions whereas global safety conditions are conditions such as array bound exception, address alignment checks and null pointer dereferencing. These conditions are represented as linear constraints. For array bound checks, a range analysis technique is used that determines the safe estimates of the range of values that each variable (register) can take at each program point. For conditions that cannot be proved using range analysis, powerful but costly program verification techniques are used.

To do the type-state checking of machine code, the following issues need to be resolved:

- Design of a language for specifying policies
- Inference of a type state at each program point
- Overload resolution of certain machine language instructions
- Synthesis of loop invariants

The safety of an untrusted piece of code needs to be defined in terms of what is acceptable or unacceptable for the host, on which the untrusted code can run. Some of the generic conditions such as type violations, array out-of-bound exceptions, address alignment violations, uses of uninitialized values and null pointer dereferences need to be satisfied irrespective of the host preferences. These conditions provide memory protection at a finer granularity. We call these conditions *default safety conditions*.

Apart from these default safety conditions, the host specifies access policies on host, which define least privileges provided by the host to the untrusted code. It allows the host to specify data that can be accessed and the host functions that can be called by the untrusted code. This reduces potential damages on the host that could happen when untrusted code runs on the host. An access policy can specify which of the pointer types are followable and thus can control the breach of security by the untrusted code on the host side. Similarly, accessible memory locations and their contents are given types, which specify ways to access them.

The memory state at each program point needs to be described. At the same time, the safety checking analysis work on a finite-sized domain is needed. The abstract storage model used in type-state checking includes an abstract store and linear constraints. An abstract store is a map from abstract locations to type states. An abstract location summarizes a set of physical locations so that the analysis has a finite domain to work on. It has a name, a size, an alignment and optional attributes to indicate whether the location is readable or writable. A type state describes the type, state and access permissions of contents of abstract locations. Linear constraints are linear equalities and inequalities combined using logical operations and quantifiers. They are used to represent safety requirements such as array bound checks, address alignment checks and null pointer checks.

The inputs to the type-state-checking analysis are an untrusted code, a host-type-state specification, an invocation specification and a safety policy. A host-type-state specification describes the type and the state of host data before the invocation of the untrusted code. It also specifies safety pre- and postconditions for calling host functions. An invocation specification specifies initial values passed to the untrusted code when it is invoked by the host. The host-specified access policies define minimum privileges given to the untrusted code by the host. All inputs except for the untrusted code are provided by the host.

The type-safe-checking analysis can de divided into the following five phases:

1. *Preparation.* The input for the preparation phase is host-type-state specification, access policy and invocation specification. The outputs of this phase are in the form of annotations and a control flow graph. The input is translated to initial annotations that give the (initial) abstract store at the entry of the untrusted code. These initial annotations are specified in terms of linear constraints and type states of the inputs.
2. *Type-state propagation.* The inputs for this phase are the interprocedural control flow graph and the initial annotations constructed in the previous phase. This phase abstractly interprets the untrusted code. Each instruction is annotated using the abstract storage model with abstract representation of memory contents where an annotation for each instruction represents the abstract memory state before the instruction execution.
3. *Annotation.* The annotation phase takes the type-state information constructed in the type-state propagation phase as an input. Then each instruction is annotated by traversing the untrusted code. The annotations represent the safety preconditions and assertions. The safety conditions are of two types, local safety conditions and global safety conditions.
4. *Local verification.* The local verification phase checks the local safety preconditions. These conditions can be checked by using type-state information alone. A linear scan over the instructions in the untrusted code is made for this.
5. *Global verification.* The global verification phase checks global safety preconditions. The global safety conditions include array bound checks, address alignment checks and null pointer dereference checks. These conditions can not be checked just by using the type-state information and further (global) analysis is required. To do this analysis, program verification techniques are used. Because program verification techniques are more costly, a technique such as symbolic range analysis is used to speed up the analysis. It may be mentioned that though program verification techniques are used here, the goal is much more modest than proving total or partial correctness of the program.

The preceding five phases of the type-state checking are illustrated in Figure 20.4 as given in [25]. In the following sections, we study the type-state system and the phases of the type-state-checking algorithm in detail.

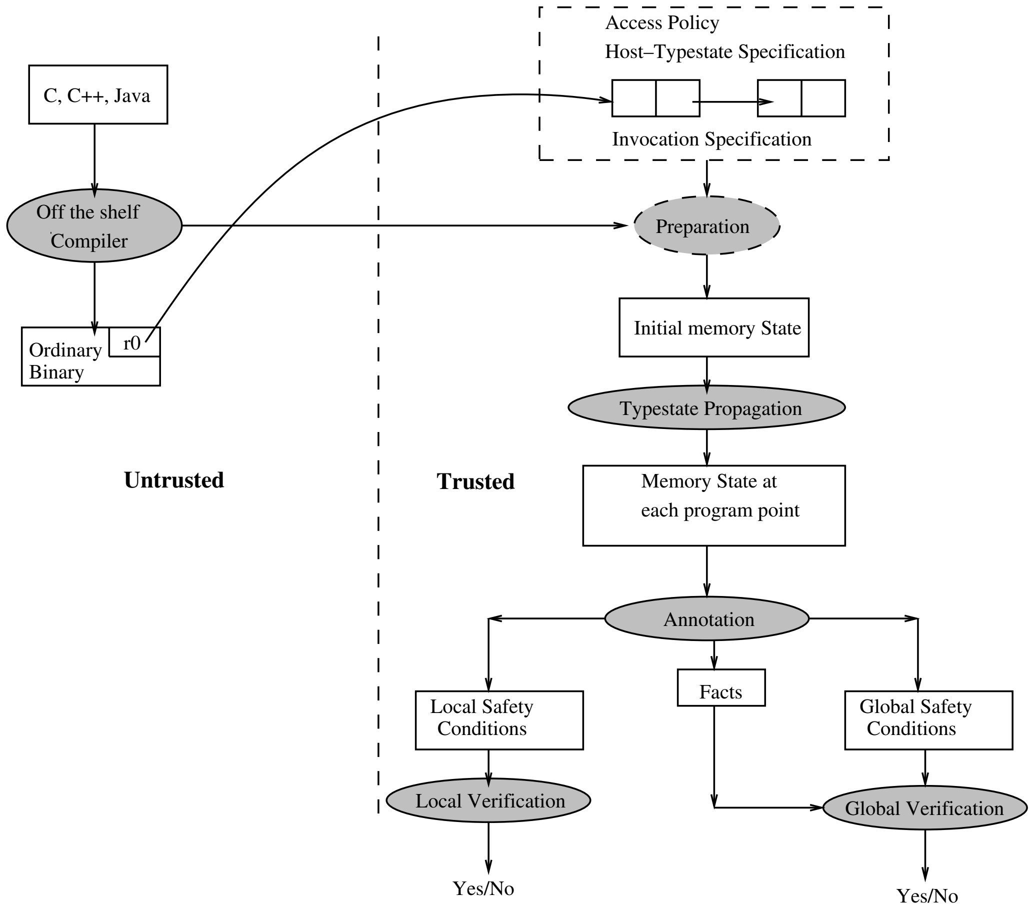


FIGURE 20.4 Illustrating the phases of type-state safety checking analysis.

20.6.2 Type-State System

Because the type-state-checking system is a static safety-checking methodology, it needs a finite domain to work on. The type-state-checking system is based on an abstract storage model. The abstract storage model consists of an abstract store and linear constraints. An abstract store maps abstract locations to type states. Each abstract location represents a set of physical locations. This makes abstract store a finite domain. An abstract location has the following attributes: name, size, alignment and optional attributes r and w to denote whether the abstract location is readable or writable by the untrusted code.

Let *AbsLoc* denote the set of all abstract locations and *Size*(l), *Align*(l) to denote the size and alignment of an abstract location l. Each type state is defined a triple $\langle type, state, access\rangle$. The type, state and access permission components of the type-state system are described next.

20.6.2.1 Type

As mentioned earlier, the type-state propagation phase of the type-state-checking algorithm uses abstract interpretation techniques to find the memory store at every program point. This is facilitated by defining a meet operation on the type states so that they form a meet semilattice. The type-state propagation phase can be viewed as some kind of symbolic execution that is flow sensitive. As in a machine level program, a register or a memory location can store values of different types at different program points; the type-state-checking algorithm has to infer an appropriate type state at each program point. The type-state-checking algorithm does so by finding the greatest fixed point of the type-state propagation equation at every program point. The type-state system uses a notion of subtyping to achieve this. Each use of register or a memory location at a given program point is resolved to a supertype of accepted values. The language of type expressions used by Xu is given in Figure 20.5 [25].

The language of type expressions as in Figure 20.5 has bit level representation of integer types. A signed integer type int($g : s : v$) has $g + s + v$ number of bits. The highest g bits are ignored, the middle s bits are sign bits and lowest v bits represent the value. This bit level representation that separates the sign bits from the value bits helps in maintaining the precision of integer operands.

A pointer into the middle of the array of type t and size n is denoted by $t(n]$. This helps in distinguishing that whether an array pointer is pointing to the start of the array or in the middle of the array. As these two types are separated, pointer arithmetic of advancing the array pointer to point to another element of the array is possible and thus the analysis is more accurate.

The top and bottom types are represented as $\top(n)$ and $\bot(n)$, respectively. They are parameterized by the size parameter n.

The subtyping rules are given next (this subtyping is known as physical subtyping and it depends on the layout of aggregate fields in the memory):

1. A type is a subtype of itself.
2. If type t has n bits, then the top type of n bits (i.e., $\top(n)$) is a subtype of t. This is because, in type-state checking, ordering is reversed (i.e., in type lattice $t_1 \leq t_2$ if and only if t_2 is a subtype of t_1). Because of the same reason, the type t is a subtype of $\bot(n)$ where t has n bits.

t :: ground	*Ground types*
| t [n]	*Pointer to the base of an array of type t of size n*
| t (n]	*Pointer into the middle of an array of type t of size n*
| t `ptr`	*Pointer to t*
| s\{ m_1 , ..., m_k \}	*`struct`*
| u\{ m_1, ..., m_k \}	*`union`*
| (t_1, ..., t_k) $\longrightarrow$ t	*Function*
| $\top$ (n)	*Top type of n bits*
| $\bot$ (n)	*Bottom type of n bits (Type "any" of n bits)*
m :: (t, l, i)	*Member labeled l of type t at offset i*
ground :: `int(`g:s:v`)` | `uint(`g:s:v`)` | `...`	

t stands for type and m stands for a structure or a union member.

FIGURE 20.5 Language for type expressions.

3. An integer of type $int(g : s : v)$ is a subtype of $int(g' : s' : v')$ if $g + s + v = g' + s' + v'$, $g \leq g'$ and $v \leq v'$. In other words, an integer type t is subtype of integer type t' if the range of (integer values) of t is subrange of t' and the sign bits of t' at the most are equal to that of t.
4. A structure is a subtype of type t' if the first member of the structure is of type t and t is a subtype of type t'. This means a structure can be passed at a place where supertype of the first member of structure is expected.
5. A structure s is a subtype of structure s' if s' is a prefix of s and each member of s is a subtype of the corresponding member of s'.
6. If t is a subtype of t', pointer to t is a subtype of t'.
7. A pointer to the base of an array is a subtype of a pointer into the middle of the array. Also, a pointer to an array whose each element is of type t is a subtype of pointer to the base of an array of size 0 and element type t.

For an assignment to be valid, it is necessary that the type on the left-hand side of the expression is a supertype (according to the preceding rules) of the type on the right-hand side and the left-hand side location has enough space.

Because the type-state-checking analysis is a flow-sensitive analysis, it tracks aliasing information among the abstract locations. This avoids illegal field access. As the aliasing information is captured, preceding rule 6 concerning subtyping of pointers is safe.

20.6.2.2 State

The conditions under which an operation is safe or unsafe is not just the function of types of its operands but also of their states. For a scalar of type t, its state can be uninitialized (denoted by $[u_t]$) or initialized (denoted by $[i_t]$). An uninitialized pointer is represented as $[u_p]$ whereas state of an initialized pointer is a nonempty set of states where one of the elements can be null. The state of an aggregate type is given by the states of its fields. A portion of state lattice as given in [25] is shown in Figure 20.6. In this lattice, the partial order for scalars is $[u_t] \leq [i_t]$, for pointers it is set inclusion and for aggregates, the partial order depends on the pairwise partial order for every field.

20.6.2.3 Access Permissions

An access permission is either a subset of f, x, o (for an individual field) or a tuple of access permissions (for an aggregate). The properties f, x, o are the properties of value stored in the abstract location. Here, f means followable, which determines whether the pointer can be dereferenced; x means executable, which denotes if the function pointed to (by a pointer to the function) can be called by the untrusted code; and o means operable, which includes operations not covered by f and x such as copy and examine. The meet operation on the access permission sets is set intersection. For aggregates, the meet of access permissions is defined as the meet of the individual elements of the the access permission.

20.6.3 Overload Resolution of Machine Instructions

In machine language programs, many instructions such as *add* and *load* are overloaded instructions. For example, *add* instruction can be used for addition of two scalars or for array address computation. However, at a given program point, each overloaded instruction usage intends only one kind of use of the instruction. Resolution of overloaded instructions at every program point to only one usage type is termed as *single-usage restriction*. At every program point, type-state propagation phase (which is a flow-sensitive phase) of type-state-checking algorithm determines the type state by taking the greatest fixed point of all type states possible at that program point.

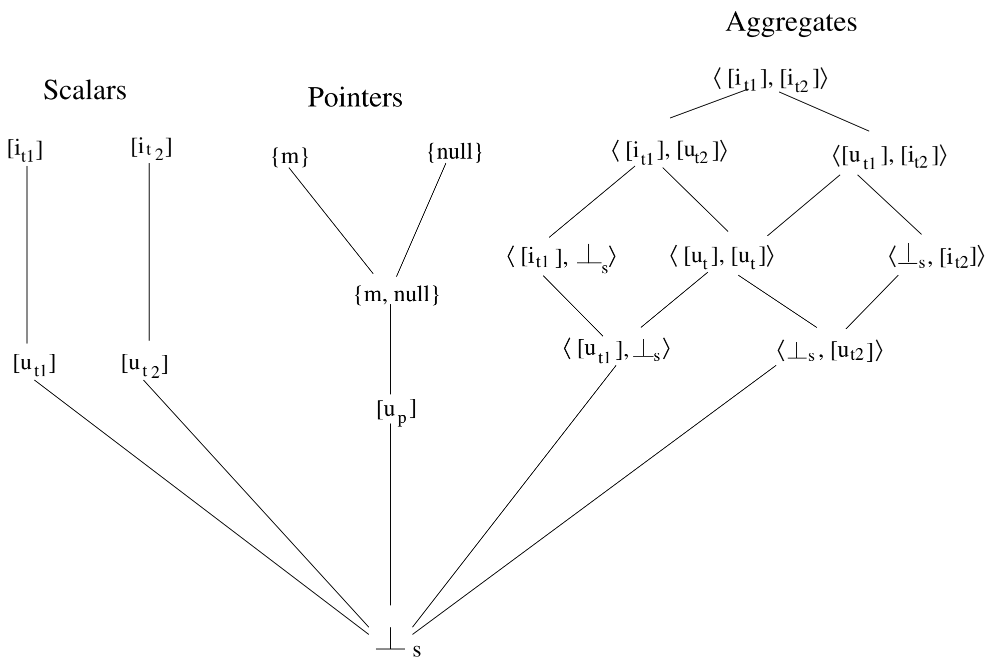

Scalars
Pointers
Aggregates
$[i_{t1}]$
$[i_{t2}]$
$[u_{t1}]$
$[u_{t2}]$
$\{m\}$
$\{null\}$
$\{m, null\}$
$[u_p]$
$\langle [i_{t1}], [i_{t2}] \rangle$
$\langle [i_{t1}], [u_{t2}] \rangle$
$\langle [u_{t1}], [i_{t2}] \rangle$
$\langle [i_{t1}], \perp_s \rangle$
$\langle [u_t], [u_t] \rangle$
$\langle \perp_s, [i_{t2}] \rangle$
$\langle [u_{t1}], \perp_s \rangle$
$\langle \perp_s, [u_{t2}] \rangle$
$\perp_s$

FIGURE 20.6 A portion of the state lattice.

20.6.4 Type-State-Checking Algorithm

As described earlier, type-state-checking algorithm consists of five phases:

1. Preparation
2. Type-state propagation
3. Annotation
4. Local verification
5. Global verification

The input to the type-state-checking algorithm is untrusted code, a host-type-state specification, an invocation specification and a safety policy. An example of an untrusted (SPARC assembly language) machine code from [25] is given in Figure 20.7. This code is machine code for a program that sums elements in the array. Host-type-state specification describes the type state of the host at the entrance of the untrusted code. It also describes pre- and postconditions under which it is safe to call host functions. Invocation specification describes the initial inputs to the untrusted code. The safety policy describes (abstract) memory locations, their contents and ways to access them. For example, host-type-state specification, invocation specification and safety policy of a host on which the untrusted code in Figure 20.7 can run is given in Figure 20.8 [25]. The host-type-state specification describes that e is an abstract location used to summarize all the elements in the array **arr**. It says that each element in the array can be "read" and is "operable". The array **arr** is of size n and pointer to the array is "followable" (i.e., it can be dereferenced). The safety policy is described in terms of a triple $\langle$*Region* : *Category* : *Access*$\rangle$. The safety policy classifies the memory into different regions. Each region can be as large as entire memory block (or address space). The category field is a set of types or aggregate fields. The access policy is defined in terms of access permissions r, w, x, f and o, which mean "readable," "writable," "executable," "followable" and "operable," respectively. As the access policy defines access permissions for a memory location, it has r and w permissions, which

```
                    UNTRUSTED CODE

 1:  mov %o0, %o2              // move %o0 into %o2
 2:  clr %o0                   // %o0 = 0
 3:  cmp %o0, %o1              // compare %o0 and %o1
 4:  bge 12
 5:  clr %g3                   // %g3 = 0
 6:  sll %g3, 2, %g2           // %g2 = 4 * g3
 7:  ld  [%o2+%g2], %g2        // %g2 = [%o2+%g2]
 8:  inc %g3                   // %g3 = %g3 + 1
 9:  add %o0, %g2, %o0         // %o0 = %o0+%g2
10:  cmp %g3, %o1
11:  bl 6                      // if (%g3 < %o1) goto 6
12:  retl
13:  nop
```

FIGURE 20.7 An example of untrusted code.

HOST TYPESTATE	ACCESS POLICY	INVOCATION
e: ⟨int, initialized, ro⟩ ap: ⟨int[n], {e}, rfo⟩ $\{n \geq 1\}$	V = {e, ap} [V : int : ro] [V : int[n] : rfo]	%o0 ←— ap %o1 ←— n
ap *is an integer array of size n, where n* ≥ 1. e *is an abstract location that summarizes all elements of* ap.	ap *and* e *are in the V region. All integers in the V region are readable and operable. All base addresses to an integer array of size* n *in the V region are readable, operable and followable.*	ap *and the size of* ap *will be passed through the registers* %o0 *and* %o1, *respectively*

FIGURE 20.8 Host information for the code in Figure 20.7.

were absent for values as given in Section 20.6.2.3. Access permissions given in Figure 20.8 tell us that each element of the array (of type integer) in the region V is readable and operable whereas the array pointer in the region V is readable, operable and followable. According to the invocation specification in Figure 20.8, the base to the array of integers is copied to the register $\%o0$ and the size of the array is passed in the register $\%o1$.

The safety policy here describes the rights given to the untrusted code to access the host data. For example, in the preceding example, the pointer to the base of an array is “readable” and “followable.”

INITIAL TYPESTATE	INITIAL CONSTRAINTS
`e`:⟨`int`, initialized, ro⟩ `%o0`:⟨`int[`n`]`, {`e`}, rwfo⟩ `%o1`:⟨`int`, initialized, rwo⟩	$n \geq 1$ and n = `%o1`

FIGURE 20.9 Initial annotations.

By explicitly specifying the safety policy in terms of access permissions, a finer level of control over the host data (by the untrusted code) can be achieved.

The five phases of the type-state-checking algorithm are preparation, type-state propagation, annotation, local verification and global verification.

20.6.4.1 Preparation

The preparation phase is the first phase of the type-state-checking algorithm. The preparation phase takes as input an untrusted code, a host-type-state specification, an invocation specification and the safety policy. From the host-type-state specification, the invocation specification and the safety policy, it finds out initial annotations. The initial annotations consist of type states and linear constraints and they specify the abstract store at the entry of the untrusted code. For example, initial annotations for host-type-state specification, invocation specification and safety policy described in Figure 20.8 are given in Figure 20.9 [25]. It describes that the array address is copied to the register $\%o0$, whereas the size of the array is passed through the register $\%o1$. It also denotes that each element of the array is initialized and it cannot be overwritten (no w permission). The w permissions to $\%o0$ and $\%o1$ refers to the permission of the registers themselves and not of their contents. The preparation phase also constructs the control flow graph for the untrusted code. The nodes in the control flow graph represent instructions and the edges represent the control flow between the instructions.

20.6.4.2 Type-State Propagation

The type-state propagation state accepts the control flow graph and initial annotations constructed in the preparation phase as its input. It then abstractly interprets the untrusted code to find out a safe approximation of abstract memory store at every program point. This analysis is a flow-sensitive analysis and is similar to the standard data flow techniques (which are also instances of abstract interpretation [19].) The initial abstract store (which is a total map from abstract locations to type states) is assigned to the start node and all other nodes in the control flow graph are assigned to the $\top$ element of the abstract store (the top element of an abstract store is a map that maps all abstract locations to their type-state respective top element). The abstract store at the entry of each node is calculated by taking meet of the abstract stores at the exit of its predecessors. The meet of two abstract stores (say mapping an abstract location a to type states t_1 and t_2, respectively) is an abstract store (a map) that maps to the meet of their respective elements (i.e., a map that maps a to the meet of t_1 and t_2).

The actual algorithm that is used by Xu [25] is a work list-based algorithm. It puts the start node into the work list and propagates the type-state information to its successors. One node is taken out from the work list at a time. The start node is given the abstract store at the entry of the untrusted code, whereas for all other nodes it is the meeting of abstract stores at the exit of their predecessors. The abstract store at the entry of every node is interpreted with respect to the semantics of the instruction.

ABSTRACT STORE
`e:` ⟨`int,` initialized, ro⟩
`%o1:` ⟨`int,` initialized, rwo⟩
`%o2:` ⟨`int[`*n*`],{e},` rwfo⟩
`%g2:` ⟨`int,` initialized, rwo⟩
`%g3:` ⟨`int,`initialized, rwo⟩

FIGURE 20.10 The memory state at line 7 of the program in Figure 20.7.

Every machine language instruction is statically given a semantics in terms of the effect it has on the abstract store. By using this semantics, the abstract store at the exit of the node is computed, which in turn may affect its successors. All affected successors are put on the work list. This process is repeated until a fixed point is reached (i.e., abstract store at entry and exit of every node assume a fixed-point value that do not change and essentially the work list becomes empty). The reaching of fixed point is guaranteed if the transition function (the function that represents the effect of every instruction on the abstract store) is a monotonic and a distributive function and the abstract store values form a finite lattice [19] (which is the case here).

Hence, a fixed point is reached and every program point (entry and exit of every node in the control flow graph) has an abstract store, which is a safe approximation of the actual abstract store values that can be assumed during execution of the program. In machine language programs, as overload resolution of instructions such as *add* or *load* is to be done, abstract store information is not propagated through the instructions of a loop until it assumes a non-$\top$ value.

For our example program in Figure 20.7, the memory state at the entry of line 7 in the example program is shown in Figure 20.10 [25]. Here, it gives us information that $\%o2$ is assigned to the base address of an array and $\%g2$ is an integer that means it is an index to the array:

- *Interprocedural analysis.* One way to handle function calls in the type-state propagation stage is to propagate the information into the body of the function. In general, problem with interprocedural analysis is dealing with parameter passing mechanisms, especially the aliasing resulting out of call by reference. An approximate but safer way is to treat the function as a black box (or a single instruction) and validate safety conditions at the entry and exit of the function. Because the conditions have to be general to cater to all call points for a function, they are weaker and the resulting type-state information can be approximate. This is known as summarization of function calls.

  Summarizing the function calls is the chosen way to handle interprocedural analysis in the type-state implementation of Xu, Miller and Reps [26]. The safety pre- and postconditions are defined at the entry and exit of each function. These conditions are represented in terms of abstract locations, type states and linear constraints. At the call site, the actual parameters are checked against the conditions they have to satisfy. The safety postconditions help to determine the state at the exit of the function. Aliasing information if provided at the exit point makes the analysis more accurate. Automatic generation of aliasing information is a difficult task and user inputs are helpful to construct the aliasing information. If the aliasing information is not present, then the state deduced at the end of the function call is less accurate but safe.
- *Detection of higher level data structures.* Because the type-state-checking algorithm operates on the machine language program, higher level type information (whether a variable represents

an array. etc.) is not available. Automatic deduction of such higher level types is difficult, but not impossible. Inference rules specific to a higher level data structure can be determined. Because each instruction is used only in one context at a given program point and the abstract interpretation that is a flow-sensitive technique can find multiple types of operands of the same instruction at a given program point, higher level type information can be deduced. For example, for a given instruction at a given program point, if the abstract interpretation discovers that a register points to two different locations in the stack, we can deduce that the register points to an array and the size of each element of the array is the greatest common divisor of the differences between location values. The type-state propagation algorithm makes it sure that the deduced types are compatible and safe.

20.6.4.3 Annotation

The annotation phase takes the type-state information generated in the type-state propagation phase as an input and annotates the machine language programs with safety conditions and assertions. The safety conditions are generated from the safety policy for the host and the default safety conditions for instructions. Depending on whether the safety conditions can be checked using the type-state information alone, the safety conditions are termed as local and global safety conditions, respectively. Assertions are the facts derived from the type state propagation values. For example, checking the type state of an operand satisfying a given safety condition is a local safety condition, whereas checking whether the array index reference is within the array bounds is a global safety condition (because array index range calculation requires global information). Examples of assertion are checking for nonnull values and address alignments.

20.6.4.4 Local Verification

Local verification stage verifies the local safety preconditions that are annotated to instructions in the previous phase. These are the conditions that can be checked using the type-state information alone. The type-state propagation phase infers the type state of each register and memory location at every program point. It can also infer the type state of values stored in the register and memory location. This information can be used to generate local safety conditions that can be checked locally. For example, if the type-state propagation finds that a register stores an array element and if every element of the array is readable, then the type state for the register should show that its content is readable.

20.6.4.5 Global Verification

Global verification is used to check safety properties that cannot be checked locally using the type-state information alone. For example, checking array bounds conditions and null pointer dereferencing require global information to validate the safety at a given program point. These properties can be verified using program verification techniques that are powerful but expensive. Alternatively, approximate but efficient abstract interpretation techniques can be used. Xu [25] first used an abstract interpretation technique known as *range analysis*, which is an approximate but a safe analysis to find out array-out-of-bound errors. The safety conditions and assertions that cannot be checked by range analysis are checked using program verification techniques. Because many of the safety conditions are already checked using range analysis, expensive program verification techniques are used only for the remaining conditions and the overall analysis is less costly.

- *Range analysis.* The range analysis finds out the (approximate and safe) range of values a memory location or a register may take. Because from the previous phases, we already know which of the registers are containing array indices, range analysis technique can be used to find whether an array out-of-bound exception exists. This is an abstract interpretation

technique where range each memory location can take or range each register can take forms a meet semilattice and standard fixed point algorithms can be used to achieve the fixed point. Abstract interpretation techniques such as widening and narrowing can be used to speed up (but approximate) the analysis.
- *Program verification.* Program verification is used only for the global safety conditions that cannot be validated using the range analysis. Automatic program verification is costly for the fact that the goal of verifying safety conditions is much more modest than checking the correctness of the program. Hence, this technique should be used rarely and on demand.

To check the global safety conditions, program verification techniques verify the assertions at a program point and prove that they hold whenever control passes through them. For an assertion, this is done by generating verifying conditions at every program point and invoking theorem provers to prove them. In case of programs with loops, one needs to synthesize loop invariants; otherwise, the analysis may go into infinite loop. A loop invariant is an assertion that is true at the entry of a loop whenever the control passes to the loop. By synthesizing the loop invariants, safety conditions can be checked by invoking theorem provers. Synthesis of loop invariants can be done using a manageable program verification technique known as *induction iteration* method, which checks the safety conditions one at a time in a demand-driven fashion. Xu [25] has suggested techniques to improve on the induction iteration method to check global safety conditions for nested and consecutive loops that are written in terms of type states and linear constraints.

20.7 Design and Implementation of a Certifying Compiler

In this section and the next, we describe how the safety-checking process may be included as part of the compilation process. We describe the design of compilers that produce a certificate of safety as an output along with the machine code. Such compilers are called *certifying compilers*.

Necula and Lee [18] describe the design and implementation of a certifying compiler that uses concepts drawn heavily from PCC techniques. The certifying compiler compiles a type-safe subset of C to the DEC Alpha assembly language. It also takes the safety policy as input and produces a proof for the safety of the generated assembly code if in fact the code is safe. Otherwise, it produces a counterexample that points to a possible violation of the type system requirements. The proof is then checked by a type-checking program as described in the earlier discussion on PCC. Thus, the approach is to check the correctness of the compiler output instead of the more ambitious attempt to verify the compiler itself. This method can be applied to optimizing compilers also. A significant fact is that the design of the certifying compiler is quite similar to the usual design of a noncertifying compiler.

In the context of the previous discussion on PCC, the certifying compiler may also be used as a front end to a PCC system. In other words, it may be used as the component that automatically produces the proof of safety.

Figure 20.11 shows an overview of the design of the certifying compiler. It consists of the compiler component and the certifier component. The compiler produces assembly code annotated with invariants that must hold to establish the safety of the code along with annotations. The annotations help the certifier to understand enough of the code to verify the type safety. This is required because the compiler performs a large number of optimizations that may break the correspondence between the assembly language program elements and the type information. For example, global register allocation may break the correspondence between registers and the type of the data that they may store. As a second example, the compiler removes the array bound checks that are present in the source code. In the absence of these checks it becomes very difficult to prove the memory safety of the assembly language program.

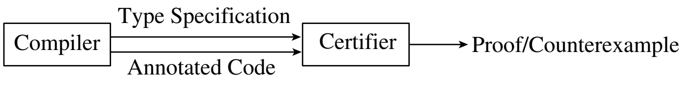


FIGURE 20.11 Overview of the certifying compiler.

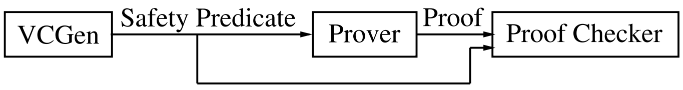


FIGURE 20.12 The structure of the certifier.

The certifier produces the safety predicate and uses a theorem prover to construct the proof the safety predicate if indeed the assembly language code is safe. Otherwise, it produces a counterexample. A proof checker verifies the validity of the proof produced by the certifying compiler. The block diagram of the certifier component of the certifying compiler is shown on Figure 20.12.

The VCGen block is the component that generates the verification conditions as already described. Its design uses the rules for generating verification conditions based on the precondition, postcondition and semantics of the DEC Alpha machine instructions. The prover block as mentioned earlier is a theorem-proving system written by the authors. As usual, the proof is encoded in the LF framework. Thus, the proof checker is nothing but a type-checking program in the LF framework.

20.8 Cyclone Certifying Compiler

Hornof and Jim [9] describe a certifying compiler that also performs runtime code generation. The source language for this compiler is a type-safe subset of C. For a given function in the source language, the user identifies the arguments that can be evaluated statically. The compiler uses the static analysis of the Tempo system [20] to produce an action annotated program. This is then translated to the Cyclone language that is a dialect of C extended with some constructs that facilitate run-time code generation.[2] Then the Popcorn compiler [13] is modified to output a language called TAL/T, which is a typed assembly language similar to TAL with support for templates that aid in runtime code generation. Because the innovative features of Cyclone and the template support in TAL/T have more to do with run-time code generation, we do not discuss these topics any further here. Code safety is ensured by writing the code in the type safe subset of C, compiling it to Cyclone that is verified to be type safe and then further compiling it to TAL/T, which can be again checked for type safety. The user has the option of programming in the C source language, in Cyclone or directly in TAL/T; and safety can be ensured through verification at any level. Proving safety properties of a TAL/T program is akin to proving the safety predicate for assembly code as done in the PCC approach. With respect to code safety, the contribution of the Cyclone approach is that although PCC showed how safety can be proved for statically generated code, the Cyclone compiler shows how it may be extended to dynamically generated code also.

[2]The Cyclone language later evolved into a language that is both safe and compatible with C.

20.9 Conclusion

This chapter gives a brief survey of a few techniques for checking mobile code safety. In particular, we concentrate on techniques that are at the interface of compilation and mobile code safety checking. For researchers in programming language theory and compilers, this is an exciting new area of research that has scope for the application of techniques in these areas to a crucial problem in mobile systems. Obviously, the topic is currently in the development stage and no one technique can be claimed to provide a complete solution. As a yardstick to measure the relevance of any new technique proposed in this area, let us enumerate what we think are some of the criteria a solution to this problem may satisfy:

1. The technique should scale to real-life, complex programs.
2. The amount of code that needs to be trusted in the solution should be as small as possible.
3. The technique should be lightweight. Many of the applications of mobile code have stringent real-time requirements. In such situations, the overhead incurred in safety checking should be as small as possible. One example of such an application is on-line downloading and executing of a wireless protocol onto a mobile phone such that minimum disruption is caused to a call in progress.
4. The technique should not be so restrictive on the source language as to impair the general applicability of the language.
5. Safety checking should involve as little manual intervention as possible.

In this chapter, we surveyed one dynamic technique (software-based fault isolation) and a few static techniques (TAL, PCC, and type-state checking of machine code) along with a brief description of the design of two certifying compilers ([18] and the Cyclone compiler).

Dynamic methods have the drawback that they may slow down the program that is undergoing safety checking. Techniques such as TAL have the drawback that they restrict themselves to a safe subset of the source language. The main drawback in the PCC approach is the size of the object code that is to be transferred. The type-state approach requires the user to supply the type and state information of the host data. As far as this method is concerned, further work needs to be done that can make this technique viable for real-life programs by reducing the cost of the global verification stage that currently uses program verification techniques. Also, the precision of the analysis can be enhanced by extending the technique to runtime checks.

Among the prominent methods that are not described in this survey is the approach based on information flow control [15]. In this approach, integrity of sensitive data is protected by statically checking the information flow of the program. Myers [15] describes an extension to the Java language called *JFlow* in which information flow annotations can be statically added.

References

[1] A.D. Birrell and B.J. Nelson, Implementing remote procedure calls, *ACM Trans. Comput. Syst.* (*TOCS*), 2(1), 39–59, February 1984.

[2] L. Cardelli, Mobile Computation, in *Mobile Object Systems — Towards the Programmable Internet*, J. Vitek and C. Tschudin, Eds., *Lecture Notes in Computer Science*, Vol. 1222, Springer-Verlag, New York, 1997, pp. 3–6.

[3] L. Cardelli and A.D. Gordon, Types for Mobile Ambients, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1999.

[4] J. Feret, Abstract Interpretation-Based Static Analysis of Mobile Ambients, *Eighth International Static Analysis Symposium (SAS'01), Lecture Notes in Computer Science*, Vol. 2126, Springer-Verlag, New York, 2001.

[5] N. Glew and G. Morrisett, Type Safe Linking and Modular Assembly Language, *26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM Press, New York, pp. 250–261.

[6] R. Harper, F. Honsell and G. Plotkin, A framework for defining logics, *J. ACM* (*JACM*), 40(1), 143–184, January 1993.

[7] R. Harper and M. Lillibridge, Explicit Polymorphism and CPS Conversion, 20th ACM Symposium on Principles of Programming Languages, Charleston, SC, January 1993, pp. 206–219.

[8] K. Harty and D.R. Cheriton, Application-controlled physical memory using external page-cache management, *ACM SIGPLAN Notices*, 27(9), 187–197, September 1992.

[9] L. Hornof and T. Jim, Certifying Compilation and Run-Time Code Generation, 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '99), January 22–23, 1999.

[10] S. McCanne and V. Jacobsen, The BSD Packet Filter: A New Architecture for User-Level Packet Capture, Proceedings of the 1993 Winter USENIX Conference, January 1993.

[11] G. Morrisett, D. Walker, K. Crary and N. Glew, From System F to Typed Assembly Language (extended version), Technical report TR97-1651, Cornell University, Ithaca, NY, November 1997.

[12] G. Morrisett, K. Crary N. Glew and D. Walker, Stack based typed assembly language, *Workshop on Types in Compilation*, *Lecture Notes in Computer Science*, Vol. 1473, Kyoto, Japan, March 1998, pp. 28–52.

[13] G. Morrissett, D. Walker, K. Crary and N. Glew, From System F to Typed Assembly Language, *ACM Trans. Programming Languages Syst.*, 21(3), May 1999.

[14] G. Morrisett, K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich and S. Zdancewic, TALx86: A Realistic Typed Assembly Language, 1999 ACM SIGPLAN Workshop on Compiler Support for System Software, 1999, pp. 25–35.

[15] A.C. Myers, JFlow: Practical Mostly-Static Information Flow Control, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1999.

[16] G.C. Necula, Proof-Carrying Code, Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 1997.

[17] G.C. Necula and P. Lee, Safe Kernel Extensions without Run-Time Checking, ACM SIGOPS Operating Systems Review, Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation, Vol. 30, October 1996.

[18] G.C. Necula and P. Lee, The Design and Implementation of a Certifying Compiler, ACM SIGPLAN Notices, Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation, Vol. 33(5), May 1998.

[19] F. Nielson, H.R. Nielson and C. Hankin, *Principles of Program Analysis*, Springer, 1999.

[20] F. Noel, L. Hornof, C. Consel and J. Lawall, Automatic, Template-Based Runtime Specialization: Implementation and Experimental Study, International Conference on Computer Languages, Chicago, IL, 1998, pp. 132–142.

[21] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper and P. Lee, TIL: A Type-Directed Optimizing Compiler for ML, Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation, 1996, pp. 181–192.

[22] D. Tennenhouse, J. Smith, D. Sincoskie, D. Wetherall and G. Minden, A survey of active network research, *IEEE Commun. Mag.* 35(1), 80–86, 1997.

[23] T. von Eicken, D.E. Culler, S.C. Goldstein and K.E. Schauser: Active messages, *ACM SIGARCH Comput. Architecture News*, 20(2), 256–266, May 1992.

[24] R. Wahbe, S. Lucco, T.E. Anderson and S.L. Graham, Efficient Software-Based Fault Isolation, ACM SIGOPS Operating Systems Review, Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles, Vol. 27(5), December 1993.

[25] Z. Xu, Safety-Checking of Machine Code, Ph.D. dissertation, University of Wisconsin, Madison, December 2000.

[26] Z. Xu, B.P. Miller and T. Reps, Safety Checking of Machine Code SIGPLAN'00, Proceedings of the ACM Conference on Programming Language Design and Implementation, Vancouver, B.C., Canada, June 18–21, 2000, pp. 70–82.

21

Type Systems in Programming Languages

Ramesh Subrahmanyam
Burning Glass Technologies

21.1 Introduction

From a typing point of view, programming languages can be statically typed, dynamically typed or typeless. Barring very low level languages such as assembly languages, no modern programming language is typeless. In statically typed languages, types are associated with (syntactic) entities in a program at compile time. The language C is an example; variables, expressions and functions have types, and programs are checked for type violations at compile time. Dynamically typed languages support such a notion only at runtime. The Lisp language is an example. A Lisp compiler or interpreter associates no types with the expressions in the supplied program; however, the values created at runtime are associated with types, and at runtime care is exercised to ensure that an operation is performed on a value only if the operation is compatible with the type of the value. Thus, a Lisp expression that adds a number to a string can be flagged at runtime as causing a runtime type error. Many statically typed languages — Simula, C++ and Java, for instance — also allow the inspection of the type of runtime values, and branching based on that type, to inject some of the flexibility of dynamic typing into an otherwise statically typed language.

This chapter discusses static typing. We consider statically typed languages and examine type systems for them. A type system specifies a set of types and a collection of typing rules. The typing

0-8493-1240-7/03/$0.00+$1.50

rules can be used to determine whether a program fragment in the language is typable — that is, whether it can be assigned a type — and, if so, to assign one or more types to the fragment. Typing rules are generally syntax directed; in this form, they are easier to state, understand and reason about than type-checking algorithms. They are also much more amenable to formal analysis; there is a wide body of literature on type theory that has had significant impact on the design of flexible type systems.

A type system is a part of the definition of a statically typed language. An implementation of the language includes a type-checking algorithm consistent with the type system. A language may require that legal programs carry explicit type annotations; such a language is said to be *explicitly typed*. For explicitly typed languages corresponding to the various type systems discussed in this chapter, it is straightforward to crank out a type-checking algorithm from the typing rules. Thus, we do not delve into the design of type checkers here. Making explicit the type of every expression is burdensome to the programmer and useless to the reader. In a languages like C, type declaration of key program entities — variables, function parameters, function results, etc. — is adequate; the rest of the types can be deduced very easily. Most languages allow the programmer to supply just about enough type information, shifting the burden of inference of the missing types to the compiler. At the other extreme to explicitly typed languages, purely *implicitly typed* languages allow a programmer to provide no type information whatsoever; for such languages, the compilers type inference algorithm deduces the type information from the program structure. However, in most practical implicitly typed languages, occasionally a programmer may need to supply information that helps the type inference algorithm tide over some ambiguity.

Types in practical type systems generally have intuitive meanings. This allows one to convince oneself that the rules in a type system make sense. In the absence of static or dynamic type checking, the execution of a program can result in a number of errors. One particularly pernicious form of error is where a semantically meaningless computation is performed on a runtime value: for instance, during a computation an integer value may be treated as a pointer and dereferenced. This notion of a computation violating type sense can be made precise; to that end, we must first describe a set of rules — called an *operational semantics* — that specify the evaluation behavior of programs in the language. Programs are evaluated by repeatedly applying rules until a value, of a specified form, is reached. However, there may be computations where the repeated application of the evaluation rules yields a term that is not of a form that values are required to be in, and yet no rule is applicable to it (e.g., during evaluation of a C program we might arrive at a state where the next subexpression is the pointer dereferencing operator applied to an integer-valued variable). Such a situation is called a runtime type error.

A type system classifies programs as well-typed or ill-typed. A good type system must guarantee that any program it certifies as well-typed cannot cause a runtime type error when evaluated. Such a type system is said to be *sound*. However, sound systems can only be conservative. Because of the undecidability[1] of any nontrivial property of program runtime behavior, while rejecting programs that are problematic, a type system may reject perfectly legitimate programs (i.e., programs whose execution does not cause runtime type errors). This is a major reason why investigations are continuing into more flexible, richer-type disciplines — type systems that toss away fewer good programs.

Unfortunately, although soundness is an objective with profound engineering consequences, either by design or by oversight, many statically typed languages have rigid type systems that reject useful programming idioms. In such cases, language designers have admitted features to help the programmer circumvent the type system; C-style casts are an example of this. The consequence is

[1]Rice's theorem in recursion theory formalizes this idea.

the violation of soundness. In other cases, flaws have crept into a type system because the designer overlooked certain subtleties. The original type system for Simula suffered from the following flaw: the type system allowed an array of type A to be a subtype of an array of type B, if A was a subtype of type B (i.e., values of type A may be manipulated in contexts expecting values of type B). Because arrays can be both accessed and written to, this allows an element of type A to be assigned to a cell of an array of type B. Performing an operation on this array cell, which is applicable to B objects but not to A objects, causes a runtime type error.

Besides the safety guarantee that a sound type system provides, static typing offers many benefits. The original motivation for introducing types in FORTRAN was to assist in compilation, specifically, efficient compilation; the introduction of distinct types for floating point numbers and integers helped overcome difficulties in mixed arithmetic at runtime. Knowledge of the type of an expression helps avoid a large amount of runtime overhead involved in performing runtime checks; for instance, knowing that a variable is of a certain record type at compile time enables a compiler to safely avoid generating runtime checks when compiling an expression that accesses a field of the structure. This improves execution efficiency. In many programming languages, the type of a variable allows precise computation of its size and this allows allocating appropriate sized memory regions to hold their values; in the absence of types, data representations must fit a default size and may require indirection, which in turn results in inefficiency.

In typed object-oriented languages, knowing the type of an object on which a method is invoked can sometimes help in determining the address of the corresponding method at compile time, enabling additional optimizations such as inlining and avoiding the runtime overhead of dynamic dispatch. More generally, knowledge of the types of objects at compile time can enable aggressive compiler optimizations. There appears to be a strong relationship between compiler optimization techniques such as flow analysis and type inference. The articles found in [32, 50] describe a correspondence between well-known type systems and various flow analysis techniques. Type systems applied to module mechanisms enable separate compilation; they also clarify dependencies enabling parallel development.

The aim of this chapter is to introduce type-theoretical issues in programming languages. We outline some of the major type disciplines to be found in existing programming language designs and algorithmic questions such as type inference for specific systems. The subject is vast and deep, and the chapter is intended to be accessible to a broad computer science audience. Whereas concepts have been introduced with some rigor and results have been stated where appropriate, detailed proofs are not provided; the interested reader can locate the proofs (as well as more detail) in the cited references. Also, we primarily look at type-theoretical issues in a functional (nonimperative) setting. This allows us to study issues such as higher order functions, polymorphism, subtyping and object orientation without the complexity of mutable state and nonlocal control. Most type-theoretical issues in imperative languages already manifest themselves in their functional subsets.

For much of this chapter, the λ-calculus is our formal vehicle for exploring typing issues. It is a syntactically minimal, yet expressive, language of (higher order) functions that is amenable to formal analysis. Section 21.2 provides an introduction to the λ-calculus. In Section 21.3, we explore a number of type disciplines in the λ-calculus framework, in particular, simple types, sum and product types, recursive types, reference types, parametric polymorphism and subtyping. We study various properties of the type system including principal typing and type inference. In Section 21.4 we examine abstract data types. In Section 21.5, we explore the type-theoretical foundations of module systems. Section 21.6 examines type systems for object-oriented languages.

21.2 Lambda Calculus

The λ-calculus is a formalism that captures the essential computational structure of functional programming languages. λ-Calculi have origins in the work of Church and Curry. They are simple calculi that support the representation of, and reasoning about, functions.

In this section we give a brief introduction to the type-free (or untyped) λ-calculus [6]. The language of untyped λ-terms is generated by the following grammar:[2]

$$t ::= \nu \mid (t \quad t') \mid \lambda x.\, t$$

Here ν is a (countably) infinite set of variable symbols; the construct $(t \quad t')$ is called an *application*; and, $\lambda x.\, t$, where $x \in \nu$, is called a *λ-abstraction*. Intuitively, an application $(t \quad t')$ represents the application of the function denoted by t to the argument denoted by t'. Also, the term $\lambda x.\, t$ denotes a function accepting an argument that is bound to the formal parameter x, and returns the value obtained by evaluating the "function body" t. As is the case with formal parameter declarations in statically scoped languages, all occurrences of the variable x in t are bound. That is to say, its scope is limited to the body t. In particular, the specific variable name x is irrelevant, and replacing all occurrences of the variable x in $\lambda x.\, t$ does not alter its meaning. By using the notation $t[t'/x]$ to mean that all instances of x in t are to be substituted by the term t', we can say that the term $\lambda x.\, t$ and $\lambda y.\, t[y/x]$ are equivalent, provided there is no occurrence of y in t that is not bound. This equivalence of terms arising from the renaming of bound variables is called *α-equivalence*.

Variable occurrences that are not bound by a λ-abstraction are said to be free; for a term t, $\mathbf{fv}(t)$ denotes the set of free[3] variables. A term that contains no free variables is said to be a *closed term*. An *open term* is a term that may have occurrences of free variables. Another notion that is used in this chapter is the notion of a substitution. A term substitution is a map from a finite set of term variables to the set of terms. For a term substitution θ and term t, the term $t\theta$ is obtained by replacing each occurrence of a term variable x, that is, in the domain of θ, in t by the term $\theta(x)$. Unless otherwise mentioned, in this chapter — subscripted or otherwise — the metavariables s and t denote terms; x, y, z denote term variables; ν, τ and σ denote types; and α and β denote type variables.

Example 21.1

Let $x, y, z \in \nu$. The following are valid terms: $\lambda x.\, \lambda y.\, (x \quad (x \quad y))$, $(x \quad \lambda x.\, x)$. In the first term, all occurrences of x are bound by the first λ-abstraction, and all variable occurrences are bound. In the second term, the left occurrence of the variable x is free, the right occurrence of the variable x is bound and the two occurrences should be deemed as completely unrelated; in fact, by α-equivalence, the second term may be rewritten as $(x \quad \lambda y.\, y)$ to emphasize this. Also, note that the variable occurring immediately after a λ is to be regarded as belonging to the λ and should not be counted as an occurrence.

21.2.1 Evaluation of Terms

Because λ-terms represent computational expressions we must state the rules for evaluating them. The intuitive meaning of λ-abstractions stated earlier suggests the following λ-calculus rule, called the *β-reduction* rule:

$$((\lambda x.\, t) \quad t') \longrightarrow t[t'/x] \qquad (\beta)$$

In other words, because $\lambda x.\, t$ represents a function with formal parameter x, its application to actual parameter t' should be evaluated by first replacing occurrences of the formal parameter in the function body t by the argument (or actual parameter), and then evaluating the resulting term.

[2]We treat the symbol as υ as a nonterminal. If $\{x_1, x_2 \ldots\}$ is the set of variable symbols, then the rule for υ is $\upsilon ::= x_1 \mid x_2 \;\ldots.$

[3]This notion makes sense for other kinds of "terms" that also have binding operators; for instance, later we talk about type expressions containing binding operators, and for a type expression τ, $\mathbf{fv}(\tau)$ denotes the free-type variables in τ.

As satisfying as the β-reduction rule is, more there mat be needs to be said before we have a full description of evaluation of λ-terms. On the face of it, there may be multiple loci within a term to which β-reduction may be applied, leading to nondeterministic evaluation. For example, consider the term $((\lambda x.\ y)\ (\lambda x.\ (x\ x)\lambda x.\ (x\ x)))$ that contains two redexes — subterms that match the left-hand side of the β-reduction rule and hence rewritable — and in evaluating the term we may reduce the left redex first and then the other redex, or the other way round. Interestingly, the λ-calculus enjoys the property that scheduling reductions in different orders cannot produce different results — the so-called *Church–Rosser theorem* [6] — which somewhat mitigates the problem. However, choosing one order might produce a result whereas choosing another can lead to an endless sequence of reductions (a nonterminating computation). Moreover, the issue raises its head as we enrich the λ-calculus with nonfunctional features; note that call-by-name and call-by-value evaluations can produce different results in a conventional programming language.

Thus, a description of evaluation of λ-terms, in addition to the rewrite rules, must also specify a strategy for choosing redexes. To this end we define the notion of *evaluation contexts*. We begin by defining *contexts*. A context, intuitively, is a λ-term with (possibly multiple occurrences) "holes" inside which we place other λ-terms. Formally, a context is a term generated by the following grammar:

$$C ::= v \ \mid\ []\mid\ \lambda x.C\mid\ (C\quad C)$$

Another way to look at a context is as follows: consider the expression tree corresponding to a λ-term. Each node in the tree is (1) a leaf node annotated by a variable; (2) a degree 1 node annotated by a λ-abstraction (and whose subtree is the tree corresponding to the term body); or (3) a degree 2 node that represents a function application, and whose left and right subtrees are the trees corresponding to the function and argument terms of an application, respectively. A context is formed by replacing zero or more subterms (subtrees) by a leaf term annotated with the symbol []. Given a context $C[]$ and a term t, the term $C[t]$ is the tree obtained by replacing all leaf nodes annotated with [] by the tree corresponding to the term t.

Evaluation contexts are a subset of the set of all contexts. The set of evaluation contexts must satisfy the following unique decomposition property: for any closed term t, a unique evaluation context $E[]$ and redex t' exist such that $t = E[t']$. An operational semantics specifies a collection of reduction rules and evaluation contexts. To apply one step of evaluation to a term t, we decompose t into the form $E[t']$, which is tantamount to locating the next subterm to be evaluated, and perform a reduction on the redex t' to say the term t''. The result of the evaluation step is the term $E[t'']$. We repeat this process by applying evaluation steps to the terms so generated until the term obtained cannot be decomposed anymore into an evaluation context–redex pair. This final term is the result of the evaluation. In the current case the only redexes are terms of the form $(\lambda x.t\quad t')$. However, as we look at various extension of λ-calculi we will encounter more redex forms.

21.2.2 Two Operational Semantics

We now define two commonly studied operational semantics for λ-calculi. Both operational semantics define the same set of reduction rules (consisting of the sole β-reduction rule) and differ only in the evaluation contexts they specify.

The call-by-value operational semantics defines the following set of evaluation contexts:

$$E ::= []|(t\quad E)$$

Also the sole reduction rule is a restriction of the β-rule $((\lambda x\cdot\ t)\quad t') \longrightarrow t[t'/x]$; it requires that t' be a λ-abstraction.

The call-by-name operational semantics defines the following set of evaluation contexts:

$$E ::= [\,]\mid\ (E\quad t)$$

The sole reduction rule is the β-rule. The reader should verify that for the two semantics the collection of evaluation contexts indeed satisfies the unique decomposition property.

Example 21.2
Consider the term $((\lambda x.\ y)\quad ((\lambda x.\ (x\quad x))\quad \lambda x.\ (x\quad x)))$. Under call-by-value semantics we obtain the following nonterminating evaluation sequence:

$$\begin{aligned}&((\lambda x.\ \lambda y.\ y)\quad ((\lambda x.\ (x\quad x))\quad \lambda x.\ (x\quad x)))\\ &\qquad \longrightarrow ((\lambda x.\ \lambda y.\ y)\quad ((\lambda x.\ (x\quad x))\quad \lambda x.\ (x\quad x))) \longrightarrow \dots\end{aligned}$$

Under call-by-name semantics we obtain:

$$((\lambda x.\ \lambda y.\ y)\quad ((\lambda x.\ (x\quad x))\quad \lambda x.\ (x\quad x))) \longrightarrow^{*} \lambda y.\ y$$

21.3 Typed Lambda Calculi

Typed λ-calculi are λ-calculi associated with a type system. Many aspects of functional programming languages, such as typing, and higher order functions are best studied in the stripped-down formalism of typed λ-calculi. Nonfunctional aspects — such as the treatment of state, and local and nonlocal control operators — have been studied in extensions of the λ-calculus where new constructs are added to model these features.

Our goal in this section is to examine a series of increasingly sophisticated typing mechanisms. We start with simple types, and progressively introduce various type constructions, polymorphism, subtypes and existential types. All these type disciplines are to be found — to varying extents — in programming languages today, though only some of these type disciplines are to be found in widely used languages. The ML, language for example, supports a limited form of parametric polymorphism, and its module language extension is based on existential types and higher order polymorphic types. Object-oriented languages support subtyping and, to varying degrees, some form of subtype polymorphism.

The specification of a typed λ-calculus starts with a description of the set of all legal type expressions. Next, the language of well-formed raw terms is specified using a grammar. Such raw terms are λ-terms embellished with type annotations. Finally, a set of typing rules is presented; these rules determine which raw terms make type sense (such terms are said to be well-typed).

Not all raw terms make type sense. For instance, consider the term $(x\quad x)$, where x is assumed to have type **int**, and **int** represents the set of integers. The term applies an integer (to itself), evidently a meaningless computation. Typing rules play the role of weeding out terms whose evaluation can cause type violations, as well as describing the types of acceptable terms.

The goal of a type system is to allow the conclusion of valid typing judgments. A typing judgment states that a term has a certain type. Because terms may have free variables, and the type of the term depends on the types of these variables, a typing judgment must explicitly specify the type environment — a mapping from free variables to their types — and thus has the form $\Gamma \vdash t : \tau$. Here t is a raw term and τ is a type. Γ is a type environment, that is, it is a partial map from ν to the set of all types, with finite domain. It specifies the types of a finite set of variables.

Notation — Often we write a type environment as simply a set of its variable-type pairs, as in $\{x_1 : \tau_1, \ldots, x_n : \tau_n\}$. The notation $\Gamma[x : \tau]$ denotes a type environment obtained by extending the type environment Γ with one more variable x and giving it the type τ.

21.3.1 Simple Types

In this section we present the types, raw terms and typing rules for the simply typed λ-calculus. This system — denoted $\lambda^{\rightarrow}$ — forms the basis for a variety of extensions studied in later sections of this chapter. The set of simple types is generated by the following grammar:

$$\tau := \mathcal{A} \ | \tau \rightarrow \tau$$

Here $\mathcal{A}$ is a set of atomic (or basic) types — types such as integers and Booleans, for example. For types τ_1, τ_2, the type $\tau_1 \rightarrow \tau_2$ should be interpreted as the type of functions that accept arguments of type τ_1 and return results of type τ_2.

The reader may wonder whether multiargument functions find a place in such a formulation. One way to model such function types is using product types that we describe later. However, one can model multiargument functions using simple types themselves — using a device called *curried function types*. For instance, consider the type **int** → (**int**→**int**), where we have assumed an atomic type **int** (denoting the set of integers). This represents the type of functions that accept an argument of type integer, and return a function of type **int**→**int**, which itself accepts an integer argument. Thus, a member of the type accepts two integers, acts on the first and, to the function returned, supplies the second as an argument eventually outputting an integer; that is, it can be viewed as accepting two integer arguments and returning an integer result.

Next, we describe the set of raw terms. These are simply terms where the λ-abstraction is annotated with the type of the abstracted variable:

$$t := v \ | \ (t \quad t)| \ \lambda x : \tau. \ \ t$$

The typing rules for the simply typed λ-calculus appear in Table 21.1. Note that these rules are templates; a specific instance of the rule is obtained by taking specific terms, types and type environments. For instance, the following is an instance of the type abstraction rule:

$$\frac{\{x : \mathbf{int}, f : \mathbf{int} \rightarrow \mathbf{int}\} \vdash (f \quad x) : \mathbf{int}}{\{f : \mathbf{int} \rightarrow \mathbf{int}\} \vdash \lambda x : \mathbf{int}. \ \ (f \quad x) : \mathbf{int} \rightarrow \mathbf{int}}$$

To establish a typing judgment one starts with the axioms — the rules without premises. In the present case, we have one such, namely, the rule labeled (**var**). Then, one applies the preceding rules repeatedly. This idea is formalized using the notion of type derivations. A type derivation is a tree in which each node is annotated with a typing judgment; further, each leaf node is annotated with an axiom, and for each nonleaf node labeled J there is a rule instance whose premises are the judgments annotating the children and the conclusion is J.

TABLE 21.1 Typing Rules for Simply Typed λ-Calculus

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\,(\mathbf{var}) \qquad \frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x : \tau_1. \ \ t : \tau_1 \ \rightarrow \tau_2}\,(\mathbf{abs})$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash (t_1 \quad t_2) : \tau_2}\,(\mathbf{app})$$

Example 21.3

Let ι be some type (say **int**). Then the following is a type derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\{x : \iota, f : \iota \to \iota \to \iota\} \vdash x : \iota \quad \{x : \iota, f : \iota \to \iota \to \iota\} \vdash f : \iota \to \iota \to \iota}{\{x : \iota, f : \iota \to \iota \to \iota\} \vdash (f \quad x) : \iota \to \iota} \quad \{x : \iota, f : \iota \to \iota \to \iota\} \vdash x : \iota}{\{x : \iota, f : \iota \to \iota \to \iota\} \vdash ((f \quad x) \quad x) : \iota}}{\{f : \iota \to \iota \to \iota\} \vdash \lambda x : \iota. \quad ((f \quad x) \quad x) : \iota \to \iota}}{\{\} \vdash \lambda f : \iota \to \iota \to \iota. \quad \lambda x : \iota. \quad ((f \quad x) \quad x) : (\iota \to \iota \to \iota) \to (\iota \to \iota)}$$

The derivation allows us to conclude that the closed term $\lambda f : \iota \to \iota \to \iota. \quad \lambda x : \iota. \quad ((f \quad x) \quad x)$ has the type $(\iota \to \iota \to \iota) \to (\iota \to \iota)$.

21.3.2 Sum and Product Types

21.3.2.1 Products

A rather straightforward extension of $\lambda^{\to}$ is with product types. For any two types τ_1 and τ_2, the product type $\tau_1 \times \tau_2$ represents the type of pairs, whose first and second components are of type τ_1 and τ_2, respectively. We, thus, have a new type constructor $\times$, as well as new term constructs. Specifically, the extended set of types is now given by the grammar:

$$\tau := \ldots \mid \tau \times \tau$$

The raw terms, now, have constructs to construct pairs and to take them apart:

$$t := \ldots \mid \mathbf{pair}(t, t) \mid \mathbf{fst}(t) \mid \mathbf{snd}(t)$$

We also have the following typing rules in addition to the ones before:

$$\frac{\Gamma \vdash t_1 : \tau_1 \quad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash \mathbf{pair}(t_1, t_2) : \tau_1 \times \tau_2}(\times - \mathbf{pair}) \quad \frac{\Gamma \vdash t : \tau_1 \times \tau_2}{\Gamma \vdash \mathbf{fst}(t) : \tau_1}(\times - \mathbf{fst}) \quad \frac{\Gamma \vdash t : \tau_1 \times \tau_2}{\Gamma \vdash \mathbf{snd}(t) : \tau_2}(\times - \mathbf{snd})$$

Product types in this form are supported in many typed functional languages, including ML and Haskell. However, in traditional imperative languages, these show up in a slightly different form as record types, for instance, struct types in C. The difference is that records are multiary (arbitrary arity) products, in which the components are marked with syntactic labels. Labeled products have the potential to support an interesting form of polymorphism [60], though this is not to be found in the widely known imperative languages.

21.3.2.2 Sums

If product types are inspired by the set-theoretical intuition of Cartesian product of sets — regarding types as sets — then sum types are inspired by the set-theoretical notion of disjoint union of sets. Given two types τ_1 and τ_2, the type $\tau_1 + \tau_2$ should be regarded as a type inhabited by pairs of the following form: the first component is a marker that takes on the values “left” or “right”; the second component is either an element of τ_1 or τ_2. If the first component is left (right, respectively), then the second component must be an element of type τ_1 (τ_2, respectively). In conventional languages, such as Pascal, this concept appears in the form of discriminated union types. An element of such a type contains a discriminator field that identifies which of one of several types appears in the rest of the union.

As before, the introduction of sum types results in the following extension to the grammar of types:

$$\tau := \ldots \mid \tau + \tau$$

The raw terms now have constructs to construct left elements and right elements, as well as a case-analysis construct that, given a $\tau_1 + \tau_2$ element, analyzes whether it is a left element or a right element; extracts the τ_1 or τ_2 element contained within; and depending on the type of the extracted element, passes that element to either a function accepting a τ_1 or to a function that accepts a τ_2 element:

$$t := ...|\ \mathbf{inl}^{\tau_1,\tau_2}(t)|\ \mathbf{inr}^{\tau_1,\tau_2}(t)|\ \mathbf{case}(s, t, u)$$

We also have the following typing rules in addition to the ones before:

$$\frac{\Gamma \vdash u : \tau_1 + \tau_2 \quad \Gamma \vdash t_1 : \tau_1 \to \tau \quad \Gamma \vdash t_2 : \tau_2 \to \tau}{\Gamma \vdash \mathbf{case}(u, t_1, t_2) : \tau}(+ - \mathbf{case})$$

$$\frac{\Gamma \vdash t : \tau_1}{\Gamma \vdash \mathbf{inl}^{\tau_1,\tau_2}(t) : \tau_1 + \tau_2}(+ - \mathbf{inl}) \quad \frac{\Gamma \vdash t : \tau_2}{\Gamma \vdash \mathbf{inr}^{\tau_1,\tau_2}(t) : \tau_1 + \tau_2}(+ - \mathbf{inr})$$

In ML, the language construct **datatype** is used to construct sum types. To take an example, a financial application may want to deal with equities: an equity may either be a stock (with a string-valued ticker symbol), or an option (with a string-valued ticker symbol, an exercise price and an expiry date). In essence, what we have is a sum of two types: the product type **int** $\times$ **string**, and **string** $\times$ **real** $\times$ **string**. In ML, the product type constructor is written $*$ and the sum type constructor is written |; also the discriminants (called constructors in ML) can be given meaningful names. This is illustrated in the following ML type declaration.

datatype equity = Stock of string| Option of string*real*string;

Values of the sum type can be constructed by "applying" the constructors to objects of the appropriate type; thus, **Options('SUNW', 15.00, 'December 2001')** is a value of this type. The ML pattern matching mechanism [31] can be used to simulate the behavior of the case construct.

21.3.3 Type Safety

So far we have discussed a type system that admits functional, product and sum types. We now formalize the notion of type safety outlined in the introduction to this chapter. A functional language defines a set of values V (which are the results of program execution), and an evaluation map $\longrightarrow^*$. The evaluation map determines the results of evaluating any raw term; the result of such an evaluation is a value, undefined or an error. A value is obtained when the evaluation is successful. An error results when evaluation is blocked at some point because no operational rule is applicable, and yet the term reached is not a value. The result can be undefined if evaluation occurs endlessly, and fails to terminate. A type system for the language is said to be *type safe* if no well-typed raw term evaluates to error. Type safety is a sanity check for a type system. The goal of type safety is sufficiently nontrivial when it needs to be balanced against the need for language expressiveness; this is illustrated by the fact that many of the widely used languages, in the interest of expressiveness, have sacrificed type safety. For example, in C, type casts introduce type unsafety.

We have formulated type safety and operational semantics in this chapter, for the most part, only for functional languages (though we do revisit this idea for a language with assignment statements in Section 21.3.8). As we said earlier, much of the complexity of typing already appears in functional subsets of imperative languages; hence, the focus is on functional languages, both in the literature and in this chapter.

To make the issue of type safety concrete, we introduce a simple language based on the simply typed λ-calculus called PCF programming language for computable functions [27] and state the type-safety theorem for it. PCF has two base types — a type of integers and a type of Booleans.

TABLE 21.2 Typing Rules for PCF

$\frac{}{\Gamma \vdash \mathbf{true} : \mathbf{bool}}$	$\frac{}{\Gamma \vdash \mathbf{false} : \mathbf{bool}}$	$\frac{}{\Gamma \vdash 0 : \mathbf{int}}$
$\frac{\Gamma \vdash t : \mathbf{int}}{\Gamma \vdash \mathbf{succ}(t) : \mathbf{int}}$	$\frac{\Gamma \vdash t : \mathbf{int}}{\Gamma \vdash \mathbf{pred}(t) : \mathbf{int}}$	$\frac{\Gamma, x : \tau \vdash t : \tau}{\Gamma \vdash \mathbf{rec}\, x : \tau.\, t : \tau}$
$\frac{\Gamma \vdash t : \mathbf{int}}{\Gamma\ \mathbf{zero?}(t) : \mathbf{bool}}$	$\frac{\Gamma \vdash t_1 : \mathbf{bool} \quad \Gamma \vdash t_2 : \tau \quad \Gamma \vdash t_3 : \tau}{\Gamma \vdash \mathbf{cond}(t_1, t_2, t_3) : \tau}$	

It has a constant 0, constructs for incrementing and decrementing numbers, a zero-test predicate, a conditional and a construct (**rec**) for making recursive definitions. More formally, the set of types is specified by:

$$\tau ::= \mathbf{int} \mid\ \mathbf{bool} \mid\ \tau \to \tau$$

Let ν be a set of term variables. The raw terms of PCF are specified by the following grammar:

$$\begin{aligned} t \ := \ & 0 \mid\ \mathbf{true} \mid\ \mathbf{false} \mid\ \mathbf{tyerr} \mid\ \mathbf{succ}(t) \mid\ \mathbf{pred}(t) \mid\ \mathbf{zero?}(t) \mid\ \mathbf{cond}(t, t, t) \mid \\ & \nu \ \mid\ (t \quad t) \mid\ \lambda x : \tau.\ \ t \mid\ \mathbf{rec}\, x : \tau.\, t \end{aligned}$$

The typing rules comprise the rules stated for the $\lambda^{\to}$, extended with those given in Table 21.2. As far as an operational semantics is concerned, we can assume call-by-name or call-by-value evaluation. The set of evaluation contexts is defined by the following grammar:

$$E ::= [] \mid\ (E \quad M) \mid\ \mathbf{succ}(E) \mid\ \mathbf{pred}(E) \mid\ \mathbf{zero?}(E) \mid\ \mathbf{cond}(E, t, t)$$

and the evaluation rules are:

$$\begin{array}{lll} \mathbf{cond}(\mathbf{true}, t_1, t_2) \longrightarrow t_1 & \mathbf{cond}(\mathbf{false}, t_1, t_2) \longrightarrow t_2 & \mathbf{pred}(\mathbf{succ}(t)) \longrightarrow t \\ \mathbf{pred}(0) \longrightarrow 0 & \mathbf{zero?}(\mathbf{succ}(t)) \longrightarrow \mathbf{false} & \mathbf{zero?}(0) \longrightarrow \mathbf{true} \\ & \mathbf{rec}\, x : \tau.\ \ t \longrightarrow t[\mathbf{rec}\, x : \tau.\ \ t/x] & \end{array}$$

Further, define the set of type-error terms by the set as follows:

- $\mathbf{cond}(t_1, t_2, t_3)$, where t_1 is not **true** or **false**.
- $\mathbf{succ}(t)$, $\mathbf{pred}(t)$, $\mathbf{zero?}(t)$ where t is not of the form 0 or $\mathbf{succ}(t)$.
- $(t_1 \quad t_2)$, where t_1 is not a λ-abstraction.

A PCF term t evaluates to a type error if there is a sequence $t \equiv t_0 \longrightarrow \ldots t_n$, where $n \geq 0$, $\longrightarrow$ is the one-step evaluation relation determined by the preceding specified notion of evaluation contexts and reduction rules, and the redex in t_n is a type-error term. The type safety of PCF is formalized in the following theorem [27]:

Theorem 21.1. *If t is a raw, closed, well-typed PCF term then t does not evaluate to a type error.*

21.3.4 Type Inference

Although requiring type annotations in programs considerably benefits programming practice, as type systems get richer and richer, annotating programs with types becomes cumbersome. In addition, the programmer may annotate symbols in a program with types that are not as general as they could be, thereby restricting the reusability of programs; this possibility arises when the type system supports some form of polymorphism. For instance, consider a program in C++ with a class hierarchy in which class B is a subclass of class A. A programmer may define a function with the declaration **void f(B)**, when in fact it is both meaningful and type correct to declare it to be of the more general type **void f(A)**.Clearly, the more general type makes the function more broadly applicable.

The process of type inference is intended to address these two issues. For many type disciplines it is possible to construct type reconstruction algorithms that accept as input a program in which some or all the type information is elided, and produce as output a program where the elided information is deduced and restored. In general, however, two obstacles exist: one, the problem of type reconstruction itself may be undecidable; two, there may be more than one — and worse infinitely many — possible type reconstructions, with no finite way to summarize the reconstructed type information. In these cases, we may only be able to perform incomplete type inference: in the first case, the algorithm may fail to reconstruct the type when in fact the program is typable; in the second case, the type information summary may be incomplete because it may not subsume one or more of the possible types. We encounter these difficulties when we consider the second-order polymorphic λ-calculus in Section 21.3.6. In many type systems where one or both of these obstacles present themselves, considerable research has gone into discovering subsystems or variants where these obstacles are absent.

In this section we consider the type reconstruction problem for the simply typed λ-calculus with a single base type ι. For a simply typed term define **erase**(t) to be the type-free term obtained by deleting all types occurring in t (note that the only occurrences of types in simply typed terms are as annotations to the bound variables in λ-abstractions). Our goal is to design an algorithm such that for any type-free term u, either one or the other of the following conditions exist:

1. The algorithm produces a typed term t and a type environment Γ such that $\Gamma \vdash t : \tau$ is provable for some type τ, and **erase**$(t) = u$ (that is the type erasure of t is u).
2. The algorithm returns failure. In this case, no Γ and satisfy the property stated in 1.

In general, even if a pair (Γ, t) satisfies this requirement, there is not a unique one. One reason is that Γ may contain type bindings for superfluous variables. Thus, we require Γ to only define types for variables that occur free in u. The problem persists, however. Consider $u \equiv \lambda x.\ x$; note that, for any type τ, the pair $\Gamma \equiv \{\}$ and $t \equiv \lambda x : \tau.\ x$ satisfy the preceding property. However, this example also demonstrates a scheme to this madness: a pattern to the various type annotations is possible.

To formalize this idea extend the set of types with a countably infinite set of variables, $\mathcal{T}$, so that types are generated by the grammar:[4]

$$\tau := \mathcal{T} \mid \ \iota \mid \ \tau \to \tau$$

Furthermore, it suffices to have the type reconstruction algorithm output the type environment and the type of the (type annotated) term; the type annotation of the term can be easily recovered from the type environment and the term type.

By revisiting our example we see that for the term $\lambda x.\ x$, any typing can be obtained by applying a type substitution to the type $\alpha \to \alpha$. A type substitution is a map from a finite set of type variables to types. To apply a type substitution θ, to a type τ, we replace each occurrence of a type variable α in τ, that is in the domain of θ, by the type $\theta(\alpha)$; the result is written $\tau\theta$. This motivates the following definition that captures the intuitive notion of the most general type for a type-free term.

[4]In the case of full-type reconstruction the base type ι is superfluous, but is relevant when we have term constants, or when we perform type reconstruction for terms where a part of the type information is elided.

Definition 21.1 (principal types). *Let u be a closed type-free term. The type τ is said to be a principal type for u, if:*

- *There is a typed term t such that* $\mathbf{erase}(t) = u$, *and,* $\{\} \vdash t : \tau$
- *For any term t and type τ' such that $\{\} \vdash t : \tau'$ and $u =$ $\mathbf{erase}(t)$, a type substitution θ exists such that $\tau\theta = \tau'$.*

We can extend this notion to open terms as follows: the pair (Γ, τ) is called a principal type environment-type pair for a type-free term u if:

- $\Gamma \vdash t : \tau$.
- For any term t, type environment Γ' and type τ such that $\Gamma' \vdash t : \tau'$, a substitution θ exists such that $\tau' = \tau\theta$, and for any variable x occurring free in u, $\Gamma'(x) = \Gamma(x)\theta$.

Two type environment-type pairs that can be obtained from each other by merely renaming the free-type variables occurring in them are considered equivalent. Given a type-free term that is typable, a unique principal type environment-type pair exists. We present an algorithm that produces the principal type environment-type pair for a type-free term u. The algorithm makes use of the notion of unification and most general unifiers [54]. The following discussion presents the relevant ideas for the specific case of type expressions.

Two types τ_1 and τ_2 are unifiable if a type substitution θ exists such that $\tau_1\theta = \tau_2\theta$; such a substitution is called a unifier of τ_1 and τ_2. In English, viewing variables in types as placeholders, two types are deemed unifiable if those placeholders may be replaced by types in such a way that the two types become identical. A unifier θ is said to be the most general unifier (mgu) if for any unifier θ', a substitution θ'' exists such that for any type variable α occurring in either τ_1 or τ_2, $\theta'(\alpha) = \theta''(\theta(\alpha))$.

Example 21.4

Let ι be a type constant. Consider the substitution $\theta = \{\alpha_1 \mapsto \iota \to \iota, \alpha_2 \mapsto \iota, \alpha_3 \mapsto \iota, \alpha_4 \mapsto \iota\}$. This substitution is defined on the set of variables $\{\alpha_1, \alpha_2, \alpha_3, \alpha_4\}$, and maps the variable to the left of the $\mapsto$ symbol to the type to the right of the symbol. Now consider type expressions $\tau_1 = (\alpha_1 \to \alpha_2)$ and $\tau_2 = (\alpha_3 \to \iota) \to \alpha_3$. Applying the substitution θ to, both, τ_1 and τ_2, results in the same type expression; that is, τ_1 and τ_2 are unifiable. However, this substitution "overcommits" in unifying τ_1 and τ_2. It suffices for a unifying substitution to unify α_1 and $\alpha_3 \to \iota$, and unifying α_3 and α_2. The substitution $\theta_0 = \{\alpha_1 \mapsto \alpha_3 \to \iota, \alpha_2 \mapsto \alpha_3\}$ achieves this. It is a unifier, and as the reader can verify, the most general unifier. Renaming type variables that occur in the codomain of the most general unifier — for instance replacing α_3 by β in θ_0 — also results in a most general unifier.

The notions of unifiability and most general unifier apply not just to a pair of types but to a set of pairs of types. If S is a set of pairs of types, a unifier of the set is a substitution θ satisfying the following property: for any $(\tau_1, \tau_2) \in S$, $\tau_1\theta = \tau_2\theta$. As before, a unifier θ is said to be the mgu for S if for any unifier θ' of S, a substitution θ'' exists such that for any type variable α occurring in some type in S, $\theta'(\alpha) = \theta''(\theta(\alpha))$. The following lemma is a corollary of the general theorem stating existence of most general unifiers for unifiable algebraic terms [54].

Lemma 21.1. *For any two unifiable types there is a most general unifier; the mgu is unique up to renaming of the variables occurring in the codomain. More generally if S is a unifiable set of type pairs, then there is a most general unifier for S.*

For types τ_1 and τ_2, $\mathbf{mgu}(\tau_1, \tau_2)$ denotes the most general unifier (fixing some choice of variables in the codomain). Likewise for a unifiable set S of type pairs, $\mathbf{mgu}(S)$ denotes the most general unifier of the set.

Next we describe an algorithm, $\mathcal{H}$, that, given a type-free term t, returns the principal environment-type pair for it if one exists, and fails if one does not exist. $\mathcal{H}$ can be defined inductively on the structure of a type-free term u. To make the induction work we have to generalize it. We assume that $\mathcal{H}$ is presented a type environment Γ^- and a term u, where Γ^- is an environment containing prior type information. It returns a pair (Γ, τ), where Γ extends Γ^-, and τ is the reconstructed type of u. The three cases follow:

- If $u \equiv x$, then $\Gamma = \Gamma^-[x : \alpha]$, and $\tau = \alpha$; here α is a type variable chosen fresh, that is, one that does not already occur in Γ^-.
- If $u \equiv \lambda x.t'$, then choose a type variable α that does not occur in Γ^-. Invoke $\mathcal{H}$ on $(\Gamma^-[x : \alpha], t')$. Let (Γ^+, τ') be the pair returned. Then Γ is the type environment obtained by removing the type binding for variable x from Γ^+. If τ'' is the type of x in Γ^+, the type to be output, τ, is $\tau'' \to \tau'$.
- If $t = (t_1 \;\; t_2)$ then invoke $\mathcal{H}$ on the pairs (Γ^-, t_1) and (Γ^-, t_2). If either invocation fails $\mathcal{H}$ returns failure. Otherwise, let $(\Gamma_1, \tau_1) = \mathcal{H}(\Gamma^-, t_1)$ and $(\Gamma_2, \tau_2) = \mathcal{H}(\Gamma^-, t_2)$. Consider a variable α that does not occur in either Γ_1 or Γ_2. Consider the set:

  $$S = \{(\tau_2 \to \alpha, \tau_1)\} \cup \{(\Gamma_1(x), \Gamma_2(x))| \;\; x \text{ has a type binding in both } \Gamma_1 \text{ and } \Gamma_2\}$$

  If S is not unifiable, return failure; otherwise, let $\theta = \mathbf{mgu}(S)$. Then set $\tau = \theta(\alpha)$.

As for Γ, it is defined as follows: for any variable, x, with a type binding in both Γ_1 and Γ_2, define $\Gamma(x)$ as $\Gamma_1(x)\theta = \Gamma_2(x)\theta$ (recall that θ unifies $\Gamma_1(x)$ and $\Gamma_2(x)$). For any variable, y, with a type binding only in Γ_1, (Γ_2, respectively), define $\Gamma(y)$ as $\Gamma_1(y)\theta$ ($\Gamma_2(y)\theta$, respectively).

Theorem 21.2. *For any type-free term u:*

- *If $\mathcal{H}$ returns failure, there is no type environment Γ, typed term t and type τ such that $\Gamma \vdash t : \tau$ and* $\mathbf{erase}(t) = u$.
- *If $\mathcal{H}$ returns* (Γ, τ), then (Γ, τ) is the principal type environment-type pair for u.

Algorithm $\mathcal{H}$ demonstrates that the inference of types actually amounts to unifiability of type expressions. More generally, we can regard type inference algorithms as attempting to solve constraints over algebras of types. The constraints may involve equalities (as was the case here), subtyping relationships (more on this in Section 21.3.7), or subset inclusions [4, 20]. A constraint solving framework can be useful even in situations where the system does not enjoy a principal typing property; the type information deduced is, essentially, a simplified constraint set. If type inference is performed for consumption by a programmer, an issue may result with presenting the deduced type information in such cases. However, it makes sense to perform type inference — in a rich type system that may not enjoy the principal type property — primarily for compilation and compiler optimization purposes; in such cases the inability to present type information succinctly is not problematic.

Partial type inference — Although the inference of types is a welcome objective from a practical standpoint, efficient and complete type inference, as well as the existence of some notion of minimal or principal types, is not possible for many rich type disciplines. One approach to overcoming this obstacle is partial type inference. In partial type inference, the programmer provides some type annotation and expects the type inference algorithm to discover the rest of the type information. This opens up the possibility for implementing heuristic techniques to attempt to discover this information. However, in practice, it is helpful to have a partial type inference framework where an abstract description exists — more abstract than the specifics of a particular algorithm — for when type inference can succeed in eliciting the elided information, and when it cannot. In [51], the authors consider a (syntactic variant of) the second-order λ-calculus, and describe two algorithms — one for the reconstruction of the (elided) types at which polymorphic functions are instantiated; the other, for the inference of the types of parameters of anonymous functions that appear as arguments to other functions. They argue empirically that these programming patterns — polymorphic instantiation and anonymous function definitions — occur very frequently in practice, and the inference of the type information described earlier is of considerable value. Their algorithm produces a minimal type where possible, and where no such minimal type exists it reports as such. The techniques they describe are local, in the sense that type information is propagated only between adjacent nodes in the abstract syntax tree. Although the algorithms are not (and cannot be) complete, they are locally complete — that is, the missing type information is inferred if it can be done so locally.

21.3.5 Recursive Types

Recursive definitions of types are fairly common in programming languages. For instance, the following type definition in C defines the type of list elements containing an integer field:

typedef struct List {int fld; List* next; } List

The following is a definition of complete binary trees with integer-valued leaves in ML:

```
datatype inttree = leaf of int| node of inttree * inttree
```

In type theory, recursive types are introduced by way of a type recursion operator **rec**; the term syntax is extended by constructs that allow elements of recursive type to be constructed and constructs that allow recursive type elements to be taken apart. First, the grammar of types:

$$\tau ::= \mathcal{T} \mid \; \tau \to \tau \mid \; \mathbf{rec}\, \alpha .\, \tau$$

The type **rec** $\alpha.\ \tau$, where, generally, α occurs in τ, should be read as the (smallest) type that is the solution of the equation $\alpha = \tau$. Consider the type **rec** $\alpha.$ $\mathbf{int} + \alpha \times \alpha$, for instance. The corresponding type equation is $\alpha = \mathbf{int} + \alpha \times \alpha$. It is not hard to see that the type of finite, complete binary trees with integer-valued leaves is the least solution to this type equation: this type consists of single-node trees labeled with integers (which can be placed in one-to-one correspondence with the integers), and complete binary trees with more than one node; each tree of the latter form can be regarded as a pair of complete binary trees. In other words, equality here is really to be interpreted as isomorphism. Because a formal semantics of types is beyond the scope of this chapter, we do not expand on this theme here.

Terms are defined by:

$$t ::= v \mid \; \lambda x : \tau .\, t \mid (t\ t) \mid \mathbf{fold}(t) \mid \mathbf{unfold}(t)$$

To understand the **fold**(.) and **unfold**(.) operators let us briefly revisit the preceding type of complete binary trees. If **tree** is the collection of complete binary trees with integer leaves, then **tree** is isomorphic to **int** + **tree** × **tree**. **fold**(.) and **unfold**(.) represents the two maps corresponding to this isomorphism; **unfold**(.) acts on an element of **tree** yielding as result the corresponding element in the union type **int** + **tree** × **tree**. It maps each single-node tree, labeled with integer n, to the element in the left component of the union that is labeled n; also, it maps a tree whose root node has subtrees A and B to an element in the right component of the union that is labeled by the pair (A, B). The operator **fold**(.) acts in the opposite direction and is the inverse of **unfold**(.). Given a binary tree we can use **unfold**() to take it apart, act on the element of type **int** + (**tree** × **tree**) — whose top level operator is not type recursion. We can also construct objects of the recursive type **tree** by first constructing an object of type **int** + (**tree** × **tree**) and applying operator **fold**(). Here is an example of a function that accepts a tree as argument and returns the tree obtained by incrementing the integers at its leaves by 1:

$$f(x : \mathbf{tree}) = \mathbf{case}(\mathbf{unfold}(x), \lambda x.\ \mathbf{fold}(x+1), \lambda y.\ \lambda z.\ \mathbf{fold}((f\ \ y), (f\ \ z)))$$

We have defined the function f by recursion (and eschewed the term recursion syntax of PCF in favor of this more readable notation). The argument x is first unfolded, and a case-based computation is performed. One case is for when the unfolded object belongs to the left component of the union, and the other is for when it belongs to the right component of the union.

The operational semantics for the extension by recursive types can be captured by extending the evaluation rules of $\lambda^{\rightarrow}$ by the following rules:

$$\mathbf{fold}(\mathbf{unfold}(M)) \longrightarrow M \qquad\qquad \mathbf{unfold}(\mathbf{fold}(M)) \longrightarrow M$$

The new constructs extend the set of evaluation contexts; for the call-by-name case, the new contexts are given by:

$$E[] ::= []|\ \ (E[]\ \ t)|\ \ \mathbf{unfold}(E[])|\ \ \mathbf{fold}(E[])$$

With these definitions, a type safety theorem can be proved for the system. Similarly, type safety can be proved for a call-by-value variant as well.

Type inference — The type inference problem for a system consisting of the $\rightarrow$ type constructor and recursive types can be formulated, as for other systems, by first defining type erasure of terms and then asking if the type of a type-erased term can be reconstructed. Define type erasure of a term to mean simply the removal of all types from terms; note that, in a raw term, besides λ-abstractions the **fold** and **unfold** constructs carry type information; these constructs are also erased during type erasure. Type inference for this system can be reduced to the problem of solving a constraint set made up of type equations. An algorithm for this decision problem can be constructed using techniques presented in [35]; in fact, [35] addresses the larger problem of type inference for systems with recursive types and subtyping.

Name and structural equivalence — The notion of type equality assumes significance with the addition of recursive types because type recursion introduces the possibility of nontrivial type equality. In a number of languages type equality is determined by identity of type names instead of type structure; in such a language, for example, two structure types with identical set of fields and field types are not considered interchangeable. In Modula-2, 2 types are considered equal only if they are stated to be equal. Algol-68 was the first language to rely on a structural type equality algorithm for recursive types. In ML, a recursive type is equal to its unfolding; barring that name equivalence prevails. However, in [5] the authors argue that name matching is problematic in languages that support data persistence and data migration. When should two types be deemed equal? A system of rules for proving type equations must include reflexivity, transitivity and the unfolding rule — namely, $\mathbf{rec}\ \alpha.\ \tau = \tau[(\mathbf{rec}\ \alpha.\ \tau)/\alpha]$ — and a rule for equality of function types stating that $\tau_1 = \tau_2$ and $\nu_1 = \nu_2$ imply that $\tau_1 \rightarrow \nu_1 = \tau_2 \rightarrow \nu_2$. However, valid equality relationships exist that cannot be validated with merely these rules. Consider the types $\tau \equiv \mathbf{rec}\ \alpha.\ \mathbf{int} \rightarrow \alpha$ and $\nu \equiv \mathbf{rec}\ \alpha.\ \mathbf{int} \rightarrow \mathbf{int} \rightarrow \alpha$. The infinitary expansions of τ and ν are identical. The rules, mentioned earlier, for deducing type equality are not adequate to prove that $\tau = \nu$. What we need is a rule that, in essence, states that for any nontrivial (nonvariable) type expression ν with free variable α, a unique solution exists for the type equation $\alpha = \nu$. Hence, the rule schema:

$$\frac{\nu[\tau/\alpha] = \tau \quad \nu[\tau'/\alpha] = \tau' \quad \nu \text{ nontrivial}}{\tau = \tau'}\ (= \mathbf{unique})$$

21.3.6 Polymorphism

Consider the (type-free) definition of the function **map** (defined using ML notation):

```
fun  map f [] = []|
     map f (x::l) = (f x)::map f l
```

This defines a functional that accepts a function f and a list $[x_1, \dots, x_n]$, and returns the list $[f(x_1), \dots, f(x_n)]$. In the type inference framework studied so far (Section 21.3.4), we can assign it any type of the form $(\alpha \to \beta) \to \alpha$ **list** $\to \beta$ **list**. We might make this definition in the context of a larger program, and proceed to use it at different points in the program. If **map** is used in a context where it acts on an **int list**, the type inference algorithm $\mathcal{H}$ infers that $\alpha \equiv$ **int**. Once α is determined to be **int**, it would be impossible to invoke **map** (in other contexts) on lists of any type other than **int list**. The type systems studied so far do not support a crucial form of reuse, one where a function is defined so as to be instantiable at several different types and reused as such. In contrast, in a language supporting parametric polymorphism — ML and Haskell, for instance — **map** can be assigned the polymorphic type $\forall\alpha, \beta.\ (\alpha \to \beta) \to \alpha$ **list** $\to \beta$ **list**. This allows any occurrence of **map** in a program to have any instance of this type, completely independent of the types of the other occurrences.

Polymorphism means having many forms; in a type-theoretical context, a polymorphic entity is one that has many distinct types. A fairly common form of polymorphism[5] is ad hoc polymorphism. For example, the addition operator in most languages has at least a couple different types. In C, it can be given the type **int** $\times$ **int** $\to$ **int** as well as the type **double** $\times$ **double** $\to$ **double**. We can regard the symbol $+$ as one that can be type instantiated to have the type **int** $\times$ **int** $\to$ **int**, as well as instantiated to have the type **double** $\times$ **double** $\to$ **double**. This overloading mechanism is a syntactic convenience, however; semantically the integer addition operator and the double addition operator are unrelated. However, an element of a polymorphic type, in a system supporting parametric polymorphism, can be instantiated at many different types and these instantiations are semantically related; intuitively, the map function acting on integer lists and float lists "acts in the exact same way."

21.3.6.1 Second-Order λ-Calculus

The formalism, $\lambda^{\forall}$, also called the second-order polymorphic λ-calculus [26, 53], comprises the set of types — called *polymorphic types* — defined by the following grammar:

$$\tau ::= \mathcal{T} \mid \ \tau \to \tau \mid \ \forall\alpha.\ \tau$$

We can add other type constructions — sum and product, for instance — without much difficulty, but we keep the system simple at this stage. Types of the form $\forall\alpha.\ \tau$ are called *universally quantified types*; the type variable α is a bound variable in the type expression $\forall\alpha.\ \tau$.

The ability to have terms of polymorphic types is accompanied by two extensions to the syntax of raw terms: one is the ability to explicitly instantiate a term of universally quantified type at some specified type; the other is the ability to abstract over the type variables occurring in a term so that it may receive a universally quantified type:

$$t ::= \nu \mid \ \lambda x : \tau.\ \ t \mid \ (t \ \ t) \mid \ \Lambda\alpha.\ t \mid \ t[\tau]$$

For instance, the term $\lambda x : \alpha.\ \ x$ has type $\alpha \to \alpha$, and the term obtained by "type abstracting" it, $\Lambda\alpha.\ \lambda x : \alpha.\ \ x$, receives the type $\forall\alpha.\ \alpha \to \alpha$. We can now "type instantiate" this last

[5]We do not discuss ad hoc polymorphism in the rest of this chapter. The addition of ad hoc polymorphism to languages that support parametric polymorphism and type inference is studied in [59].

term — obtaining the term $(\Lambda\alpha.\ \lambda x : \alpha.\ \ x)[\mathbf{int} \to \mathbf{int}]$ — which gives us the identity function of type $(\mathbf{int} \to \mathbf{int}) \to (\mathbf{int} \to \mathbf{int})$.

The typing rules for $\lambda^{\forall}$can be obtained by extending the typing rules of $\lambda^{\to}$ by adding the two rules that follow. It should be noted that in the rules carried over from $\lambda^{\to}$, the types occurring in type environments and as the types of terms are, in general, polymorphic types. The new rules describe how the new language constructs are to be typed:

$$\frac{\Gamma \vdash t : \tau \quad \alpha \notin \mathbf{fv}(\Gamma) \cup \mathbf{fv}(\tau)}{\Gamma \vdash \Lambda\alpha.\, t : \forall\alpha.\, \tau}\ (\forall - \mathbf{abs}) \qquad \frac{\Gamma \vdash t : \forall\alpha.\, \tau}{\Gamma \vdash t[\tau'] : \tau[\tau'/\alpha]}\ (\forall - \mathbf{inst})$$

The notation $\mathbf{fv}(\Gamma)$ refers to the type variables that occur free in the various types assigned to term variables by Γ. In the type abstraction rule, we require that the variable that is abstracted over does not occur in the types of variables that are free in the term.

A key concept here is that of type impredicativity. Note that in explaining universally quantified types, $\forall\alpha.\ \tau$, we referred to the notion of a family of elements indexed by the set of all types. An element of circularity exists here because the set of all types includes $\forall\alpha.\ \tau$. Impredicativity leads to semantic complexity, and the formal analysis of impredicative systems can be nontrivial.

Some key results about this system follow:

- Given a typing judgment, the problem of deciding whether the judgment is derivable is decidable.
- Given a closed type-free term t the problem of type reconstruction for the term is unsolvable [61]; that is, it is undecidable whether a typable raw term exists whose erasure is t.
- The principal typing property does not carry over to the $\lambda^{\forall}$.

In other words, although we have a powerful type discipline, we cannot have the benefits of efficient type inference for such a system.

21.3.6.2 ML Polymorphism

A considerable weakening of second-order polymorphism has been implemented in the ML language. This fragment enjoys the principal typing property, and has a decidable type inference problem [18]. In ML, polymorphism appears in a very restricted form — one that does not allow function types (or, for that matter, any type constructors) with polymorphic argument types; hence, this restricted form of polymorphism is also called *let polymorphism.* We employ a fragment λ^{ML} of ML in this section to explain the main ideas. λ^{ML} defines two kinds of types: monotypes and type schemes. The set of monotypes — denoted by the metavariable τ — is generated by the grammar:

$$\tau ::= \mathcal{T} \mid \tau \to \tau$$

Type schemes — denoted by the metavariable σ and its subscripted variants — are of the form $\forall\alpha_1, \ldots, \alpha_n.\ \tau$, where τ is a monotype. The variable α in the type scheme $\forall\alpha.\ \tau$ ranges over the set of monotypes. By adopting this stratification, impredicativity is kept at bay.

The set of λ^{ML} raw terms is specified by the following grammar:

$$t ::= \nu_T \mid \nu_S \mid \lambda x : \tau.\ \ t \mid\ (t \quad t) \mid\ \mathbf{let}\ X : \sigma =\ t\ \mathbf{in}\ t\ \mathbf{end} \mid\ X[\tau]$$

Here, we have two variable sets — the lower case metavariables x, y, z range over the set ν_T and represent variables that are monotypes. The upper case metavariables X, Y, Z range over the set ν_S and represent variables that are of universally quantified type. The former variables can occur as formal parameters of functions; the latter variables are bound by the **let** construct, which allows the binding of polymorphic entities to variables.

The rules for λ^{ML} comprise the rules for $\lambda^{\rightarrow}$ extended by the following two rules:

$$\frac{X : \forall\alpha.\, \sigma \in \Gamma}{\Gamma \vdash X[\tau] : \sigma[\tau/\alpha]}\,(\textbf{type inst}) \qquad \frac{\Gamma, X : \forall\alpha_1, \ldots, \alpha_n.\, \tau' \vdash t : \tau \quad \Gamma, X : \tau' \vdash t' : \tau' \quad \alpha_i \notin \mathbf{fv}(\Gamma)}{\Gamma \vdash \mathbf{let}\ X : \forall\alpha_1, \ldots, \alpha_n.\, \tau' = t'\ \mathbf{in}\ t\ \mathbf{end} : \tau}\,(\textbf{let})$$

For this system we regain the key properties that were lost in the second-order λ-calculus. Define a type-erased λ^{ML} term to be a raw λ^{ML} term from which all types have been removed. The following key results can be shown for this system [27]:

- The system λ^{ML} enjoys the principal typing property.
- Type reconstruction is decidable. For any type-erased λ^{ML} term t, an algorithm exists to determine whether there is a well-typed λ^{ML} term whose type-erasure is t.

The type inference algorithm is essentially the algorithm $\mathcal{H}$ extended to handle the let construct, and to deal with type environments containing variables whose types are type schemes. The latter poses no serious problems. As for the former, suppose the expression, whose type is to be inferred, is of the form **let** $X = t'$ **in** t **end**. For simplicity, assume that the definition $X = t'$ is not recursive; the recursive case is left as an exercise to the reader. Invoke $\mathcal{H}$ on t' to deduce its type τ; generalize the type, τ, to the type scheme $\forall\overline{\alpha}.\ \tau$, where $\overline{\alpha}$ is the set of type variables free in τ but not in the type environment; assign the type $\forall\overline{\alpha}.\ \tau$ to X and in the extended type environment infer the type of t. The time complexity of type inference for λ^{ML} is EXPTIME [37].

Polymorphic recursion — In the let rule, it should be noted that when typing **let** $X : \sigma = t'$ **in** t **end**, although X has polymorphic type in t it must have monomorphic type while typing t'. In [46], this monomorphic typing rule for recursive definitions has been extended to one in which x is used polymorphically in t'; further, it is shown to be sound and to possess the principal typing property. Polymorphic recursive definitions are useful, but the type inference problem for this system — which is called the Milner–Mycroft calculus — is undecidable [33]; however, [33] presents some reasons why type inference with polymorphic recursion appears to be practical despite its undecidability.

21.3.7 Subtyping

Subtyping is a mechanism where, roughly speaking, the values comprising a type form a subset of values comprising another type. Subtyping appears in many programming languages. It appears in the form of coercions of atomic types — or automatic conversion of one type to another — in languages like C. In the language Caml, it appears in the form of record type subsumption, where a record of one type, τ_1, can be considered to also belong to another record type τ_2, provided the latter has fewer fields than the former, and the type in τ_2 of each common field is a subtype of the type of the field in τ_1. In object-oriented languages like C++ and Java, a subtyping relationship holds between a class and any class derived from it.

The semantics of subtyping has been well studied. In such semantics a type is regarded as a subtype of another if there is a coercion function from the semantic domain of values corresponding to the subtype, to the semantic domain of values that constitute the supertype. Such a coercion map "forgets" some information in its domain to yield the "coerced" element in the range. For instance, consider record types τ_1 and τ_2, where τ_1 has one integer-valued field, and τ_2 extends the type τ_1 with a Boolean valued field. We can interpret τ_2 by the type of pairs whose first component is an integer and second component is a Boolean; τ_1 can be interpreted as the type of 1-tuple of integers, or simply, integers. τ_2 is a subtype of τ_1, and this is written $\tau_2 \sqsubseteq \tau_1$. The obvious coercion map from τ_2 to τ_1 maps a pair to its first component, forgetting the rest of the contents of the pair.

This intuitive interpretation of subtyping suggests some properties. The subtyping relation is reflexive and transitive. How does subtyping interact with the function space operator? Suppose f is a function of type $\tau \to \nu$. If $\tau^- \sqsubseteq \tau$, then f can act on an element of τ^- because this element can be silently coerced before being acted on by f; the result would be an element of type ν; if $\nu \sqsubseteq \nu^+$, then the result can be coerced to type ν^+. Given the coercion maps from τ^- to τ, and ν to ν^+, we have arrived at a method of coercing a function of type $\tau \to \nu$, to $\tau^- \to \nu^+$. In other words, if $\tau^- \sqsubseteq \tau$ and $\nu \sqsubseteq \nu^+$, then $\tau \to \nu \sqsubseteq \tau^- \to \nu^+$. This idea is described by saying that the function type constructor, $. \to .$, is contravariant with respect to the type argument that appears to the left of the arrow symbol and covariant with respect to the type argument that appears to the right.

To add subtyping to a programming language, we can begin with a collection **Atom** of atomic types, and a subtyping relation on the type expressions, that is, a binary relation $\mathcal{C}$ on type expressions. For instance, we can take the set of atomic types to be **Atom** $= \{$**int**, **real**$\}$ and the subtyping relation to be $\mathcal{C} = \{$**int** $\sqsubseteq$ **real**$\}$. More generally, $\mathcal{C}$ can be a relation over arbitrary types, not merely atomic types. If the subtype relation is a relation over atomic types and type variables, we call the type system an *atomic subtyping system.*

We proceed to develop an extension of $\lambda^{\to}$ with subtyping. Reasoning about typing in this calculus involves two forms of judgments. One, as usual, is a form of judgment asserting that a term has a certain type. The other is a form of judgment asserting that a type is a subtype of another. Assumptions in the first judgment form are pairs — one component is a type environment Γ, and the other is a set of assumptions stating that a type is a subtype of another. We use the metavariable $\mathcal{C}$ to denote the subtyping assumptions. Thus, a judgment has either the form $\mathcal{C}; \Gamma \vdash t : \tau$, or $\mathcal{C} \vdash \tau_1 \sqsubseteq \tau_2$. The former asserts that under the assignment Γ, and subtyping assumptions $\mathcal{C}$, the term t has type τ. The latter asserts that under subtyping assumption $\mathcal{C}$, τ_1 is a subtype of τ_2. The intuitions developed in the previous paragraphs can now be formalized using the following rules:

$$\frac{\tau_1 \sqsubseteq \tau_2 \in \mathcal{C}}{\mathcal{C} \vdash \tau_1 \sqsubseteq \tau_2}\ (\textbf{axiom}) \qquad \frac{}{\mathcal{C} \vdash \tau \sqsubseteq \tau}\ (\textbf{reflexive})$$

$$\frac{\mathcal{C} \vdash \tau_1 \sqsubseteq \tau_2 \quad \mathcal{C} \vdash \tau_2 \sqsubseteq \tau_3}{\mathcal{C} \vdash \tau_1 \sqsubseteq \tau_3}\ (\textbf{transitive}) \qquad \frac{\mathcal{C} \vdash \tau_1 \sqsubseteq \tau_2 \quad \mathcal{C} \vdash \nu_1 \sqsubseteq \nu_2}{\mathcal{C} \vdash \nu_1 \to \tau_2 \sqsubseteq \tau_1 \to \nu_2}\ (\textbf{subtype} \to)$$

Finally, the rules of $\lambda^{\to}$ can be extended with the following subsumption rule:

$$\frac{\mathcal{C} \vdash \tau_1 \sqsubseteq \tau_2 \quad \Gamma; \mathcal{C} \vdash t : \tau_1}{\Gamma; \mathcal{C} \vdash t : \tau_2}\ (\textbf{subsumption})$$

21.3.7.1 Coherence

Before we get to type inference for this system we ask a simpler, but important, question. Given an explicitly typed term in this calculus, is there an algorithm to determine whether the term is typable? It is not hard to construct such an algorithm, but we notice that unlike other calculi we have encountered so far, explicitly typed terms can have multiple-type derivations. For instance, using the subtyping assumption $\mathcal{C} = \{$**int** $\sqsubseteq$ **real**$\}$, the term $\lambda x :$ **int**. x can be assigned the type **int** $\to$ **real** using two distinct derivations. One derivation first establishes that the term has type **int** $\to$ **int**, then uses the rules (**transitive**) and (**subtype** $\to$) to conclude that **int** $\to$ **int** $\sqsubseteq$ **int** $\to$ **real**, and finally uses the rule (**subsumption**) to conclude that $\{x :$ **int**$\}; \mathcal{C} \vdash \lambda x :$ **int**. $x :$ **int** $\to$ **real**.

A different derivation uses the rule (**subsumption**) to prove that $\{x :$ **int**$\}; \mathcal{C} \vdash x :$ **real**, and then uses the rule (**abs**) (see Table 21.1) to conclude that $\{x :$ **int**$\}; \mathcal{C} \vdash \lambda x :$ **int**. $x :$ **int** $\to$ **real**. Each derivation corresponds to inserting coercion functions differently in the two terms. Inserting coercion functions in a term is important in language implementation; a compiler for C++ needs to determine whether a coercion has to be applied to the right-hand side of an assignment and if so what the coercion function is; coercion applied to a pointer may translate to nontrivial pointer arithmetic. Consequently, if coercions can be introduced into a term in several different ways owing to multiple-type derivations, the term may evaluate differently for the different ways to insert coercions. The same issue arises when giving denotational semantics to terms, where differing ways of inserting coercion functions in a term can result in potentially assigning different meanings to the term. This has been referred to as the coherence problem for subtyping calculi.

An approach to the coherence problem of a calculus appears in [7, 17]. The general approach is to reduce the problem of coherence for a calculus to a set of equations between terms built from the coercion maps in the calculus. Consider the simply typed λ-calculus extended with a set of type constants and a collection of atomic subtyping relations. Let f_{τ_1,τ_2} denote the "standard" coercion from τ_1 to τ_2, where $\tau_1 \sqsubseteq \tau_2$, induced by the *a priori* coercions between types in the subtyping assumptions. Consider types τ_1, τ_2, ν_1, ν_2 such that $\tau_1 \sqsubseteq \tau_2$ and $\nu_1 \sqsubseteq \nu_2$; then coherence requires that the equation $f_{\nu_1 \to \tau_2, \tau_1 \to \nu_2}\langle a\rangle(b) = f_{\nu_1,\nu_2}\langle a(f_{\tau_1,\tau_2}\langle b\rangle)\rangle$ must hold. Here a is of type $\nu_1 \to \tau_2$, b is of type τ_1 and the notation $f\langle a\rangle$ stands for the application of the coercion function f to a. For a semantics to be coherent, this and other coherence equations must hold. From a compilation perspective, the coercion operators inserted by a compiler must satisfy these equations if evaluation is not to be influenced by the specific type derivation that is constructed.

In languages that support generic operators, it is important for coercions to interact well with generic operators. Consider a language that has types **int** and **real**, with **int** $\sqsubseteq$ **real**, and the generic operator $+$. In the presence of generic operators we have a form of ad hoc polymorphism: the operator $+$ is polymorphic but its instances at the various types do not necessarily have any systematic relationship with one another. If we have an assignment of the form $x := y + z$, where x is a real and y and z are integers, then there are a couple different ways of inserting coercion operators to achieve the requisite type casting. We must then require certain equalities to hold, for coherence reasons: one equation, for instance, is $f_{\mathbf{int},\mathbf{real}}(x) +_{\mathbf{real}} f_{\mathbf{int},\mathbf{real}}(y) = f_{\mathbf{int},\mathbf{real}}(x +_{\mathbf{int}} y)$. Reynolds [52] addresses the problem of designing an imperative language supporting implicit coercions and generic operators where the necessary coherence conditions are realized.

21.3.7.2 Extending Subtyping to Richer Type Disciplines

It is easy to see that given $\tau \sqsubseteq \tau'$ and $\nu \sqsubseteq \nu'$, $\tau \times \nu \sqsubseteq \tau' \times \nu'$. Likewise, $\tau + \nu \sqsubseteq \tau' + \nu'$. Now consider polymorphic types: when does subtyping hold between two universally quantified types? Suppose $\mathcal{C} \vdash \tau \sqsubseteq \nu$, and consider a type variable α that possibly occurs free in τ, ν or both. Also,[6] there is a coercion map, definable by the term f_α that coerces τ to ν. Then the term $\Lambda\alpha.\ f_\alpha$ defines the coercion map from $\forall\alpha.\ \tau$ to $\forall\alpha.\ \nu$. This informal argument shows that $\forall\alpha.\ \tau \sqsubseteq \forall\alpha.\ \nu$.

An interesting type construction called *bounded quantification* has been studied extensively in the literature [11]. As stated earlier, an element, e, of the type $\forall\alpha.\ \tau$ represents a family of entities indexed by the collection of all types; the entity indexed by type τ' is the type instantiation of e at type τ' and has the type $\tau[\tau'/\alpha]$. In like fashion, each element of the type $\forall\alpha \sqsubseteq \nu.\ \tau$ is a family of entities indexed by the family of types that are subtypes of ν. For a concrete example, suppose we have a language with record types and record subtyping, and bounded and unbounded quantification. Then the type $\forall\alpha.\ \alpha \to$ **int** is populated by polymorphic functions that can take arguments of any type and return an integer (barring constant-valued functions one would be hard put to come up with any functions of this type). Each member of the type $\forall\alpha \sqsubseteq \{b : \mathbf{int}\}.\ \alpha \to \mathbf{int}$, on the other hand, is a function that takes an argument whose type is a subtype of the record type with an integer-valued field labeled b, and returns an integer result. An example of a function of this type is the function that returns the value in the b field of its argument.

[6]We can make this more precise by arguing by induction on the derivation of $\mathcal{C} \vdash \tau \sqsubseteq \nu$ that a term exists for witnessing $\mathcal{C} \vdash \tau \sqsubseteq \nu$.

As with unbounded universally quantified types, we can deduce the rule for subtyping bounded quantified types:

$$\frac{\mathcal{C}, \nu_1 \sqsubseteq \nu_2 \vdash \tau \sqsubseteq \nu}{\mathcal{C} \vdash \forall\alpha \sqsubseteq \nu_1.\ \tau \sqsubseteq \forall\alpha \sqsubseteq \nu_2.\ \nu}(\textbf{subtype} - \forall\ \textbf{bounded})$$

For recursive types the situation is more subtle. In analogy with polymorphic types we might consider the rule (†): if $\tau_1 \sqsubseteq \tau_2$ then **rec** $\alpha.\ \tau_1 \sqsubseteq$ **rec** $\alpha.\ \tau_2$. Consider types ν^- and ν^+, and suppose $\nu^- \sqsubseteq \nu^+$. By rule (†), $\equiv$ **rec** $\alpha.\ \alpha \to \nu^- \sqsubseteq$ **rec** $\alpha.\ \alpha \to \nu^+$. However, we can unwind these latter two recursive types, which gives us the equality **rec** $\alpha.\ \alpha \to \nu^- =$ (**rec** $\alpha.\ \alpha \to \nu^-) \to \nu^-$, and **rec** $\alpha.\ \alpha \to \nu^+ =$ (**rec** $\alpha.\ \alpha \to \nu^+) \to \nu^+$. Therefore, we can conclude that (**rec** $\alpha.\ \alpha \to \nu^-) \to \nu^- \sqsubseteq$ (**rec** $\alpha.\ \alpha \to \nu^+) \to \nu^+$. In light of the rule, (**subtype** $\to$), the subtype relation is contravariant with respect to the argument that appears to the left of the $\to$ constructor. In particular, this last subtyping assertion is false.

Define an occurrence of a type variable in a type to be negative (positive) if it occurs to the left of an even (odd) number of $\to$ symbols. The problem with the rule (†) comes from the negative occurrence of the type variable bound by the type recursion operator. Notice that the (sole) occurrence of the variable in both **rec** $\alpha.\ \alpha \to \nu^+$ and **rec** $\alpha.\ \alpha \to \nu^-$ is negative. If no occurrence of the bound variable α, in the types τ_1 and τ_2, is negative, the rule (†) is sound; in the general case the following rule applies [5]:

$$\frac{\mathcal{C}, \alpha \sqsubseteq \beta \vdash \tau \sqsubseteq \nu}{\mathcal{C} \vdash \textbf{rec}\ \alpha.\ \tau \sqsubseteq \textbf{rec}\ \beta.\ \nu}\ (\textbf{subtype rec})$$

For types built from the function type constructor and type recursion, we have presented a collection of rules for deducing subtyping relationships: these include (**axiom**), (**reflexivity**), (**transitivity**), (**subtype** $\to$) and (**subtype rec**). We also need a rule that captures the interaction of subtyping and equality: a rule that states that if $\tau_1 = \tau_2$, $\tau_2 \sqsubseteq \tau_3$ and $\tau_3 = \tau_4$, then $\tau_1 \sqsubseteq \tau_4$. This in turn means that we must have rules to reason about type equality. Rules for reasoning about type equality were presented in Section 21.3.5.

21.3.7.3 Type Inference

A number of different calculi with subtyping have been the subject of study, from a type inference viewpoint. The typability problem for the simply typed λ-calculus with an arbitrary subtyping relation can be reduced to the problem of solving a system of subtyping constraints [13]. The computational characteristics of the solvability of these constraints depends on whether subtyping is atomic; and, if it is atomic, then it depends on the structure of the subtyping order relation on the atoms.

In practice, the subtyping relation is often atomic. An interesting example of a calculus where the subtyping relation is not atomic is the system of partial types [57], where there is a type constant denoted $\top$ (the catchall type) and every type is given to be a subtype of $\top$; the type $\top$ can be used to type certain subterms, which though meaningful may not be typable within the rigid confines of the type system. Many variants assume a type $\bot$ that is a subtype of all types in the system.

We say that the judgment $C'; \Gamma' \vdash t' : \tau'$ is an *instance* of $C; \Gamma \vdash t : \tau$ if a type substitution θ exists such that $C \vdash C'\theta$, $\Gamma\theta \subseteq \Gamma'$ and $\tau' = \tau\theta$. A variant of the preceding instance relation has been studied in [25] in the context of atomic typing. The problem of type inference for this calculus — independent of the specific subtyping relation — can be reduced to the problem of solving a set of subtype constraints over type expressions. Call a constraint set $\mathcal{C}$-consistent if a type substitution θ exists such that for each $\tau_1 \sqsubseteq \tau_2 \in C$, $\mathcal{C} \vdash \tau_1\theta \sqsubseteq \tau_2\theta$. The following results follow from [12]:

THEOREM 21.3. *Let $\mathcal{C}$ be a subtyping relation. An algorithm exists that accepts a type-free term t_0 and returns a quadruple (C, Γ, t, τ). Here C is a solvable constraint set, Γ is a type assignment, t is an explicitly typed term whose type erasure is t_0 and τ is a type. If for some term t', such that* **erase**$(()t') = t_0$*, some $\mathcal{C}$-consistent constraint set C', assignment Γ' and type τ', the judgment $C'; \Gamma' \vdash t' : \tau'$ is derivable, then C is $\mathcal{C}$-consistent, and $C'; \Gamma' \vdash t' : \tau'$ is an instance of $C; \Gamma \vdash t : \tau$.*

Thus, given a subtyping relation over types, determining typability requires determining whether a system of constraints is solvable. This issue has been studied widely in the literature on subtyping [4, 35]. Type inference amounts to returning a $\mathcal{C}$-consistent constraint set that satisfies the minimality property stated in the preceding theorem. For atomic subtyping without recursive types the problem of type inference is PSPACE complete; if the atomic subtyping relation is a disjoint union of lattice or trees, it is solvable in polynomial time. In the presence of recursive types, there is an EXPTIME solution to the problem. For simply typed λ-calculus with partial types, an algorithm was given by [48]; when the system is extended with recursive types, the problem is in PTIME. For $\lambda^{\rightarrow}$ extended with the types $\top$ and $\bot$ an $O(n^3)$ algorithm is available. For recursive types extended with $\top$, $\bot$ and for discretely ordered base types the problem is solvable in $O(n^3)$.

21.3.8 Typing Imperative Features

An interesting interplay occurs between polymorphism and imperative features. Consider a simple extension of λ^{ML} with mutable types — specifically, heap allocated reference types — and assume a call-by-value operational semantics. We introduce a new constructor for types, **Ref** :

$$\tau ::= ... \mid \mathbf{Ref}\ \tau$$

For any type τ the type **Ref** τ denotes a mutable cell on the heap that can store a value of type τ. We, therefore, introduce operations to create (and initialize) a mutable cell, dereference (extract the value stored in) a mutable cell and assign a value to a mutable cell:

$$t ::= ... \mid \mathbf{ref}(t) \mid \mathbf{deref}(t) \mid\ t := t$$

An operational semantics for this language extension can be given. We simply sketch the semantics (the reader interested in a rigorous definition may consult, e.g., [27]). The set of values is obtained by extending the set of values of λ^{ML} by values denoting locations. The operational semantics defines a rewrite relation on term–state pairs; a rewrite step has the form:

$$(s_{pre}, t) \longrightarrow (s_{post}, t')$$

where s_{pre} is the state prior to the one step evaluation and s_{post} is the state after the one-step evaluation. States are defined as partial functions defined on a finite domain of initialized locations; a state maps each location in its domain to the value contained therein. The dereferencing operation results in a rewrite where $s_{post} = s_{pre}$. The evaluation of a term **ref**(t), starting at state s, comprises (1) the (many step) evaluation of t to a value v and state s'; (2) extending the state s' with a new mutable cell containing the value v to yield the state s''; and (3) terminating in a term–state pair where the term part is a value denoting the newly created location, and the state is s''. Of course, (1) may not terminate at all. The evaluation of $t_1 := t_2$ comprises the evaluation of t_1 to a value denoting a location L, the evaluation of t_2 to a value v, and updating location L in the state, reached after the evaluation of t_1 and t_2, to contain the value v.

Based on this intuitive explanation, the following rules should be clear:

$$\frac{\Gamma \vdash t : \mathbf{Ref}\ \tau}{\Gamma \vdash \mathbf{deref}(t) : \tau} \quad \frac{\Gamma \vdash t : \tau}{\Gamma \vdash \mathbf{ref}(t) : \mathbf{Ref}\ \tau} \quad \frac{\Gamma \vdash t_1 : \mathbf{Ref}\ \tau \quad \Gamma \vdash t_2 : \tau}{\Gamma \vdash t_1 := t_2 : \mathbf{unit}}$$

unit is a type with exactly one element; from a typing point of view there is no need to distinguish between different assignment expressions; thus, a type with exactly one element suffices.

The problematic rule is the rule (**let**) from Section 21.3.6.2:

$$\frac{\Gamma, X : \forall\alpha_1, \ldots, \alpha_n.\ \tau' \vdash t : \tau \quad \Gamma, X : \tau' \vdash t' : \tau' \quad \alpha_i \notin \mathbf{fv}(\Gamma)}{\Gamma \vdash \mathbf{let}\ X : \forall\alpha_1, \ldots, \alpha_n.\ \tau' = t'\ \mathbf{in}\ t\ \mathbf{end} : \tau}$$

Consider the expression[7] **let X** : $\forall\alpha\cdot$ **Ref** (α *list*) = **ref**([]) **in** $X := [true]$; $car($**deref**$(X))+1$ **end**. Here, car is a list function that returns the first element of its (list-valued) argument. The reader can verify that this expression type checks if we admit the preceding rule; this is because [] has type $\alpha\ list$ and **ref**([]) has type **Ref** ($\alpha\ list$). The rule allows us to generalize the type of **ref**([]) to $\forall\alpha.$ **Ref** ($\alpha\ list$). Unfortunately, if X is given the type $\forall\alpha.$ **Ref** ($\alpha\ list$), we can give its various occurrences in the body of the let differing instances of this type. In the preceding expression, at one occurrence of X we assign it a value of type **bool** $list$; at the other occurrence we dereference X, instantiating its type to **int** $list$. Consequently, the evaluation of such an expression attempts to perform an integer operation on a Boolean value resulting in a runtime type error! This is a general problem that arises when imperative features interact with polymorphism. A similar example can be constructed for control operators, such as the Scheme-style call-current-continuation operator $call/cc$ [28].

One solution to the problem, implemented in an earlier version of Standard ML of New Jersey, is to treat specially type variables that occur in the types of imperative subexpressions, specifically by restricting type generalization over such variables [31]. A different solution — *value polymorphism* — is the observation that a restricted form of the rule (**let**) is type safe; the restriction is that polymorphic generalization of the let-bound variable x be allowed only if the expression t' (bound to x) is a value expression, that is, an expression — such as a λ-abstraction or a list value — that cannot be evaluated further. It has been argued empirically that this restriction does not significantly hamper language expressiveness in practice [62].

A third point of view has been argued in [39]. Consider the term **let** $X : \forall\alpha.\ \alpha \to \alpha =$ **ref**$(\lambda x.\ x)$ **in** ... **end**; if we make all type information fully explicit here — as is the case in the second-order λ-calculus — then we would have the term **let** $X : \forall\alpha.\ \alpha \to \alpha = \Lambda\alpha.$ **ref**$(\lambda x : \alpha.\ x)$ **in** ... **end**. In evaluating this latter term, we must specify the operational semantics of the type abstraction operator Λ. In the terminology of [39], a call-by-name strategy would not evaluate "beneath" the Λ; a call-by-value strategy would proceed to evaluate the term beneath the Λ. If we adopt a call-by-value strategy, $\Lambda\alpha.$ **ref**$(\lambda x : \alpha.\ x)$ evaluates to a reference containing the identity function; this will be bound to X and all occurrences of X in the body of the let refer to the same cell. In contrast, under call-by-name evaluation of Λ, $\Lambda\alpha.$ **ref**$(\lambda x : \alpha.\ x)$ is treated as a value (much like a λ-abstraction); at each type instantiation of x in the body of the let, $\Lambda\alpha.$ **ref**$(\lambda x : \alpha.\ x)$ is evaluated to a reference cell containing the identity function of the type corresponding to the type at which X is instantiated. Thus, each occurrence of (type instantiations of) X evaluates to a different reference cell. Such an operational semantics avoids the type safety problem we have described. As [39] points out "polymorphism by name has been criticized on the grounds of inefficiency and on the grounds that an imperative language with nonstrict constructs is error prone"; however, Leroy [39] argues that efficient compilation is possible, and that in practice rarely do programs behave differently under the two semantics.

[7]Here ";" is the sequencing operator. In a call-by-value language the expression $t_1; t_2$ is merely syntactic sugaring for $((\lambda x.\ t_2)\ \ t_1)$.

21.4 Data Abstraction and Representation Independence

The use of abstract data types is very common, and many languages provide constructs for defining abstract data types. For instance, Ada provides the package construct, CLU offers the cluster construct and the Modula family of languages offer the module construct. The following ML code illustrates MLs — now obsolete — **abstype** construct whose typing is quite similar to those for other languages:

```
abstype Stack  = stack of int list with
fun pop : Stack → int = ..
fun push : int → Stack  = .. ..
end
```

The preceding **abstype** definition defines a new type **Stack** and offers the functions **push** and **pop** to compute over this type. Further, it gives a specific implementation of the type **Stack**, representing it as an integer list. This representation of stacks is visible to the implementations of **push** and **pop**; however, outside of the **abstype** construct the representation is not visible. That is, a function that acts on a **Stack** argument cannot exploit the fact that **Stack** is implemented as an integer list. A **Stack** object is a black box, and the only means of performing any computation over stack objects is to use the interface functions provided.

The type system of the language enforces these restrictions. As a consequence of this enforcement, we have encapsulation: it is possible to alter the representation type — and correspondingly the definitions of the interface functions — of the abstract data type without breaking the typability of the rest of the program. A specific property, called *representation independence*, can be established for abstract data types: if the implementation (representation) of a data type can be replaced by an equivalent representation, the behavior of the clients of the abstract data type remains unaltered [44].

Abstract data types can be modeled using a form of types called *existential types* [14]. Starting with the type system $\lambda^{\rightarrow}$ or $\lambda^{\forall}$, we can add a new type construction of the form $\exists\alpha.\ \tau$. An element of the type $\exists\alpha.\ \tau$ should be thought of as consisting of a pair (τ_0, e), where τ_o is a type and e is an element of type $\tau[\tau_0/\alpha]$. The type τ_0 is the representation type (like the type **int list** in the preceding **abstype** definition), and e is the tuple consisting of the interface methods. Given such a pair, we can construct an element of the type $\exists\alpha.\ \tau$ using the **pack** construct. The construct **unpack** allows existential type interface functions to be extracted and — in keeping with typing rules that enforce the black box nature of abstract types — to perform computations using those functions.

The raw terms of this type system are given by adding the following two constructs:

$$t ::= \ldots \mid \mathbf{pack}^{\exists\alpha.\ \tau}(\tau_0,\ t) \mid\ \mathbf{unpack}^{\exists\alpha.\ \tau} t\ \mathbf{as}\ \alpha, f : \tau\ \mathbf{in}\ t'$$

The term $\mathbf{pack}^{\exists\alpha.\ \tau}(\tau_0,\ t)$ constructs an instance of the existential type $\exists\alpha.\ \tau$ from a representation type τ_0 and an implementation of the interface methods t, which satisfy the typing $\tau[\tau_0/\alpha]$. Thus, we have the typing rule:

$$\frac{\Gamma \vdash t : \tau[\tau_0/\alpha]}{\Gamma \vdash \mathbf{pack}^{\exists\alpha.\ \tau}(\tau_0,\ t) : \exists\alpha.\ \tau}$$

The term $\mathbf{unpack}^{\exists\alpha.\ \tau} t\ \mathbf{as}\ \beta, f : \tau[\beta/\alpha]\ \mathbf{in}\ t'$ acts on a term t of type $\exists\alpha.\ \tau$ as follows: it extracts the tuple of functions from t — the tuple having the type $\tau[\beta]$ for some unknown type β, binds it to the variable f and evaluates the term t' (which, possibly, contains free occurrences of f). This gives the typing rule:

$$\frac{\Gamma \vdash t : \exists\alpha.\ \tau \quad \Gamma, f : \tau \vdash t' : \tau' \quad \alpha \notin \mathbf{fv}(\tau')}{\Gamma \vdash \mathbf{unpack}^{\exists\alpha.\ \tau} t\ \mathbf{as}\ \alpha, f : \tau\ \mathbf{in}\ t' : \tau'}$$

The type variable α may not appear in τ'. This is to ensure that the variable α has only local significance in the construct — serving as a placeholder for the unknown representation type used in the construction of the term t of abstract type — does not escape.

In a realistic implementation **pack** and **unpack** are purely compile-time operations, with no computational significance. We define the operational semantics so that **unpack** can only be applied to **pack**. The only evaluation rule in the system is:

$$\mathbf{unpack}^{\exists\alpha.\,\tau}(\mathbf{pack}^{\exists\alpha.\,\tau}(\tau_0,\ t))\ \mathbf{as}\ \alpha,\ f:\tau\ \mathbf{in}\ t' \longrightarrow t'[\tau_0/\alpha, t/f]$$

We also extend the evaluation contexts as follows:

$$C[]\ ::=\ \ldots|\ \mathbf{unpack}^{\exists\alpha.\,\tau}C[]\ \mathbf{as}\ \alpha,\ f:\tau\ \mathbf{in}\ t'|\ \mathbf{pack}^{\exists\alpha.\,\tau}(\tau_0,\ C[])$$

as well as the valid values with:

$$v\ ::=\ \ldots|\ \mathbf{pack}^{\exists\alpha.\,\tau}(\tau_0,\ v)$$

These rules can be shown to be sound with respect to the operational semantics.

21.4.1 First-Class Abstract Types

Although the **abstype** syntax allows the creation of abstract data types, and the abstract types so defined can be used freely with other type constructors to construct more complex types, only one implementation is possible for an abstract type name in a program, namely, the one given in the abstype definition. Thus, an abstype is a composite of the implementation and interface type. The calculus of existential types suggests a way to overcome this limitation. An extension to ML that supports the notion of an abstract type with multiple implementation types is presented in [47]. This is accomplished by extending ML with an implicit existential quantifier. Specifically, in a data type declaration such as:

$$\texttt{datatype Stack} = \texttt{Stack of}\ \{\texttt{rep}: {'}\texttt{a}, \texttt{push}: {'}\texttt{a} * \texttt{int} \to {'}a, \texttt{pop}: {'}a \to \texttt{int}\}$$

the type variable $'\mathbf{a}$ is treated as implicitly existentially quantified; $\{\ldots\}$ is the ML notation for constructing records. The expression:

$$\begin{aligned}&\texttt{Stack}(\{\texttt{rep} = [], \texttt{push} = (\texttt{fn x}: \texttt{int list} * \texttt{int} => \ldots),\\ &\quad \texttt{pop} = (\texttt{fn x}: \texttt{int} => ..)\})\end{aligned}$$

represents a concrete stack (with custom implementation). This syntax is sugaring for:

$$\begin{aligned}\mathbf{pack}^{\tau}(\mathbf{int\ list},\ \{rep = [], push\ &= \lambda x:\mathbf{int\ list}\times\mathbf{int}\to\mathbf{int\ list}.\ \ ,\ldots, pop\\ &= \lambda x:\mathbf{int}\to\mathbf{intlist}.\ \})\end{aligned}$$

where $\tau = \exists\alpha.\ \alpha * (\alpha\times\mathbf{int}\to\alpha) * (\alpha\to\mathbf{int})$. The system enjoys principal typing and decidable type inference, and conservatively extends the underlying type discipline. It allows construction of heterogeneous aggregates of implementations of a given abstract type and dynamic dispatching with respect to the implementation type.

21.5 Modules

Module mechanisms — as found in the Modula language family, for instance — are very closely related to abstract data type mechanisms. However, their role is primarily to allow programmers to package logically distinct program parts. An important goal in the design of module mechanisms is to allow the secure combination of these parts to form complete programs, and to support separate compilation. However, as argued in [41], achieving some of these goals requires mechanisms different from abstract type mechanisms that hide the representation of types, and existential type mechanisms that hide the identity of the abstracted type.

Among languages offering module mechanisms, ML has one of the most sophisticated. In ML, modules are called *structures*. Type definitions, value bindings and function definitions are some of the bindings to be found in a structure. The notion analogous to types for modules is called a *signature* in ML. The left expression below defines an ML structure and binds it to the name *S*; one signature of this structure appears on the right:

```
structure S = struct             sig
 type char_set = char list;       type char_set;
 val empty: char_set= [];         empty: char_set;
 fun insert(x, s) = ..           val insert: char * char_set →
                                 char_set;
 fun member(e, s) = ..           val member: char * char_set →
                                 bool;
end                               end
```

The ML **type** definition does not create a new type; it merely creates a new type name, which is interchangeable with the type expression it is bound to in subsequent code. As a result, if we replace (some subset of the) occurrences of **char_set**, in the type declarations of the values in the preceding signature, by the type **char list** we still have a legitimate signature of the preceding structure. In other words, a structure can have multiple signatures.

As with abstract data types, the types defined in the module appear in the parameter or result types of the values and functions. Unlike abstract data types, however, it is possible (and desirable) to let the types defined in a structure to "escape." The type defined in the preceding structure may be referred to in later code as **S.t**.

A useful language feature is parametric modules — modules that are parameterized by some (compile-time) parameter. ML supports functors to support this mechanism; a (first-order) functor is essentially a function from structures to structures. The preceding structure *S* can be generalized to a *Set* functor:

```
functor Set(Element: ElementSig): SetSig signature ElementSig =
 struct                                   sig
  type SetType = Element.T list;           type t;
  val empty = [];                          val eq: T → T → bool;
  fun insert(x, s) = ..                   end
  fun member(e, s) = ..                  signature SetSig =
 end                                     sig
                                           type SetType;
                                           val empty: SetType;
                                           val insert: T * SetType →
                                           SetType
                                           val member: T * SetType → bool
                                         end
```

Such a functor takes a structure of signature **ElementSig** as an argument — a signature comprising a type and an equality test function at that type — and yields as result the structure representing sets of elements of the type specified in the argument structure; the result has signature **SetSig**. At compile time this functor can be applied to a structure containing the type **char** and the equality function on **char**; it will yield the char set structure *S* we defined earlier.

21.5.1 Dependent Types

Dependent types are a useful formalization of modules. The two forms of dependent types are the general sum type and the general product type. General sum types have the form $\Sigma x : \tau.\ \tau'$ and general product types have the form $\Pi x : \tau.\ \tau'$; here, τ and τ' are types, and the variable x possibly occurs free in τ'. An element of type $\Sigma x : \tau.\ \tau'$ is a pair consisting of an element a of type τ and an element of type $\tau[a/x]$; note the similarity with existential types then τ represents the collection of all types. Thus, a signature with a type component t and a value component of type τ (the type τ possibly containing occurrences of t) can be modeled by the type $\Sigma t : \mathbf{Type}.\ \tau$, where **Type** is the type of all types.[8] Note that we are treating individual types as values that can be computed over, and that values can appear in type expressions. That is, the space of values and types have become intertwined.

The type $\Pi x : \tau.\ \tau'$ represents a collection of functions indexed by the elements of type τ. An element of type $\Pi x : \tau.\ \tau'$ is a function mapping each element a of type τ to an element of the type $\tau'[a/x]$; note that if x does not occur free in τ' the type $\Pi x : \tau.\ \tau'$ is no different from the function type $\tau \to \tau'$. A functor signature where the argument signature is (modeled by the type) τ_a and the result signature is (modeled by the type) τ_r, and the argument name s possibly occurs in τ_r can be modeled as the dependent product type $\Pi s : \tau_a.\ \tau_r$.

Example 21.5
In a rich enough language of type expressions we might have a type expression **int_or_bool**(x), where x is an integer variable, and:

$$\mathbf{int\_or\_bool}(x) = \begin{cases} \mathbf{int} & \text{if } x > 0 \\ \mathbf{bool} & \text{otherwise} \end{cases}$$

The type $\Sigma x : \mathbf{int}.\ \mathbf{int\_or\_bool}(x)$ is populated by the elements $((\mathbf{Z}^- \cup \{0\}) \times \{\mathbf{true}, \mathbf{false}\}) \cup \mathbf{Z}^+ \times \mathbf{Z}$, where $\mathbf{Z}$ is the set of integers.

The type $\Pi x : \mathbf{int}.\ \mathbf{int\_or\_bool}(x)$ is populated by functions that map the positive integers to some integer, and nonpositive integers to a Boolean value. We can also think of $\Pi x : \tau.\ \tau'$ as representing a (possibly infinite) product of types, indexed by τ where the type indexed by element a of type τ has the type $\tau'[a/x]$. We can thus regard an element of type $\Pi x : \mathbf{int}.\ \mathbf{int\_or\_bool}(x)$ as a (Z-indexed) infinite sequence; the elements indexed by the positive integers are integers, and the remaining elements are Boolean values.

We introduce the term construct $[t,\ u]$ to create an element of sum type — operationally the construct merely creates a pair comprising t and u. The constructs **fst**(.) and **snd**(.) extract the first and second component of the pair. The corresponding typing rules are:

$$\frac{\Gamma \vdash t : \tau \quad \Gamma \vdash u : \tau'[t/x]}{\Gamma \vdash [t,\ u] : \Sigma x : \tau.\ \tau'} \qquad \frac{\Gamma \vdash t : \Sigma x : \tau.\ \tau'}{\Gamma \vdash \mathbf{fst}(t) : \tau} \qquad \frac{\Gamma \vdash t : \Sigma x : \tau.\ \tau'}{\Gamma \vdash \mathbf{snd}(t) : \tau[\mathbf{fst}(t)/x]}$$

We generalize and reuse the λ-abstraction and application constructs for creating and manipulating terms of generalized product types. Note that both rules generalize the corresponding rules in $\lambda^{\to}$ accounting for the possibility that the bound variable may occur free in the result type:

$$\frac{\Gamma, x : \tau' \vdash t : \tau}{\Gamma \vdash \lambda x : \tau'.\ \ t : \Pi x : \tau'.\ \tau} \qquad \frac{\Gamma \vdash t : \Pi x : \tau'.\ \tau \quad \Gamma \vdash t' : \tau'}{\Gamma \vdash (t\ \ t') : \tau[t'/x]}$$

[8]Momentarily, we have introduced impredicativity but as we will see shortly this can give way to a stratified system of types.

As we observed earlier — and the reader may wish to study Example 21.5 closely — in the dependent types of calculus, the types of terms are inextricably linked with the values of the terms. Up until now, the type systems have maintained a strict phase distinction: types are compile phase entities and term evaluation occurs at runtime, and the types of terms that need to be computable at compile time are not dependent on information — such as whether two integer values terms have the same value — that is at best knowable only at runtime. This phase distinction is crucial for performing compile-time type checking. In the present form of the dependent type calculus this crucial property is lost. Because we seek to model module constructs using the dependent type calculus, we must restrict the calculus if the property is to be recovered; this is achieved by restricting the type of indexes used in the construction of dependent sums and products to range over entities (like structures) that are evaluated at compile time. Further, we distinguish between different categories of terms and types. We have two categories of terms: core language terms, and the module language terms (structure and functor expressions). Paralleling this distinction, we have two categories of types: core language types and the module language types (structure and functor signatures).

We start with the type of *small types* that are the types to be found in the underlying core language — these consist of base types such as integers and Booleans, and are closed under various type constructions (in particular, the ones that we have encountered so far, in this chapter). This type will be denoted **Type**. Next, we define a subset of dependent sum types — those corresponding to the signatures of structures; these are called signatures as well and the terms inhabiting these types are called structures:

$$sig \ ::= \ \textbf{Type} | \ \ \Sigma s : \ sig \ \tau | \ \ \Sigma s : \ sig \ sig'$$

The first case corresponds to structures that only have a type component; the second corresponds to a structure with one structure component and one value component of type τ (which may contain occurrences of the structure identifier naming the first component); the third corresponds to a structure with two structure components, the signature of the second having a dependency on the first structure by way of occurrences of the first structure identifier in the signature of the second. Corresponding to these cases we have the following module language term syntax:

$$strexp \ ::= \ \nu_s | \ \ \tau | \ \ [strexp, \ t] | \ \ [strexp, \ strexp]$$

Here, ν_s is an infinite set of structure identifiers, τ is a core language type and t is a core language term. The grammar of types has an additional production that corresponds to the extraction of the type component of a structure; for a structure expression s that has a type component, the syntax is $\textbf{fst}(s)$. The grammar of core language terms, similarly, has a production corresponding to extracting the value component; for a structure expression s that has a term component, the syntax is $\textbf{snd}(s)$.

As for functors, the grammar of functor signatures is given by:

$$fsig ::= \Pi s : sig. \ sig | \ \ \Pi s : sig. \ fsig$$

The use of a distinct nonterminal $fsig$ allows us to avoid presenting formation rules for well-formed signatures. The first case corresponds to the case of a functor that takes an argument (structure) of a given signature and returns a structure of the specified signature. The second case corresponds to a curried functor: one that accepts a structure argument of the specified signature and returns a functor of the specified signature. The corresponding module language term syntax is enriched by the following:

$$strexp ::= ...| \ \ \lambda s : sig. \ \ strexp | \ \ (strexp \ \ \ strexp)$$

corresponding to functor abstraction and application. Given that the signatures admit only first-order functors, the type-checking rules ensure that only first-order functors are typable.

For this system we can give an operational semantics along expected lines. The additional module and core language term syntax contains as, usual, λ-terms, the construction of pairs and projections (extraction of components of a pair). It can be shown that if the core language is type safe, then so is the extension with the module type discipline described here.

It can be shown that this calculus respects the phase distinction. The module expressions (which include the core language types) are compile-time entities and the rest are runtime entities. The evaluation of compile-time entities does not involve the evaluation of runtime entities [30].

Two important issues are briefly mentioned here. The first is the notion of sharing constraints. Consider a functor, *compiler*, that accepts several structures that perform various compilation functions: the first structure is a collection of functions that perform lexical analysis yielding tokens of a certain token type; the second accepts a stream of tokens and parses them yielding a parse tree. Each of these structures must therefore contain a substructure that defines the type of tokens and a suite of token manipulation functions. The *compiler* functor would then invoke the tokenizing functions and pass the resulting token stream to the appropriate parsing functions. To type check the functor, *compiler*, there has to be a means of specifying that the token types in the lexical analysis and parsing modules are the same. Further, it would be reasonable to expect the token manipulation functions to be the same in both structures. In effect, there ought to be a mechanism to specify that certain substructures and types are the same. In ML, a mechanism called *sharing specification* serves this purpose.

A second important notion relates to the notion of structure transparency. Note that the preceding formalization does not in any way hide the definitions of the type names in a structure. It is possible to develop alternative semantics where this is not the case [29, 38]. The preceding properties are preserved with the addition of sharing specifications as well as adoption of these alternatives to structure transparency.

21.5.2 Higher Order Functors

The functors we have considered are functions over structures. A natural generalization is to consider higher order functions over structures. For instance, a second-order functor accepts first-order functors as arguments. This generalization is useful in practice [42].

An important issue that arises when we consider a module system with higher order functors is the phase distinction. A straightforward generalization to the higher order case would proceed as follows. The two-level stratification (core language types and module language types) gives way to an infinite hierarchy. We call each stratum a *kind*. The kind K_1 consists of all structure signatures (and thus **Type**). The kind K_2 consists of all types in K_1, and a type $\mathbf{Type}_1$ whose elements are the types in K_1; it is closed under the general sum and general product constructions, that is, K_2 also includes all types of the form $\Pi t : \tau.\ \tau'$ and $\Sigma t : \tau.\ \tau'$, where $\tau, \tau' \in K_2$. In general, kind K_{n+1} (1) contains all of K_n, (2) contains a type $\mathbf{Type}_n$ whose elements are the types in K_n and (3) is closed under the general sum and product constructions. The raw structure expressions are as before — built from the λ-abstraction, pairing and projection constructs. We can put down typing rules along obvious lines. The reader may consult [30] for more details.

Unfortunately, as pointed out in [30], this straightforward generalization admits problematic expressions that violate the phase distinction. Consider the following example:

$$((\lambda x : \mathbf{fst}(F([\mathbf{int},\ t'])).\ \ x)\ \ t)$$

where t is some core language term and F is a first-order functor variable.[9] Type checking this expression involves determining whether t has the type $F([\mathbf{int},\ t'])$. This type expression is very similar to the type expression $\mathbf{int\_or\_bool}(x)$ that we encountered earlier; even at the point (during compilation) where F is bound to an actual functor, the identity of the type $F([\mathbf{int},\ t'])$ cannot be known without knowing the value of t'. The term t', however, cannot be evaluated at compile time. Thus, type checking is no longer performable at compile time.

We outline the basic intuition underlying a solution here, referring the reader to [30] for a rigorous treatment of the matter. Consider a (first-order) functor F that acts on a structure of signature **sig type t; val x: T end** to produce a result of signature **sig type t'; val y: T' end**. Given the compile-time nature of this computation, the action of such a functor may be described as follows: the type t' can be a function of type t, that is, $t' = f(t)$ for some function f on types. The (runtime) value y can be a function of the value x, that is, $y = g(x)$ for some g. Type g must be a function polymorphic in the type t; the type of g can be of the form $\forall t.\ T \rightarrow T'[f(t)/t']$. Thus, such a functor should be regarded as belonging to the type $\Sigma f : \mathbf{Type} \Rightarrow \mathbf{Type}.\ \forall t.\ T \rightarrow T'[f(t)/t']$. Here $\mathbf{Type} \Rightarrow \mathbf{Type}$ is the type of type functions — specifically, unary type constructors. In other words, the "natural" generalization of the first-order module calculus to a higher order module calculus should preserve this interpretation of functors. As shown in [30], this interpretation of functors as higher order structures induces a nonstandard equational theory. Specifically, these equations allow the transformation of "mixed-phase" type expressions such as $A([\mathbf{int},\ t']$ to purely compile-phase type expressions.

21.6 Typing in Object-Oriented Languages

Broadly speaking, object-oriented languages fall in two camps: class-based languages and delegation-based languages. Class-based languages have a notion of classes; classes are compile-time entities that act as templates at runtime for creating objects. Classes support code reuse through inheritance; inheritance allows the creation of new classes through modification of existing classes by overriding their methods, or by extending them with new methods. Relatively fewer languages are delegation based, and support the more powerful notion of prototypes. A prototype is an object that supports method override, and extension by new methods at runtime. In such a language, objects can be created directly or by cloning an existing object (hence, the name *prototype*).

21.6.1 Object Calculi

Just as the type-free and typed λ-calculi are useful formalisms for a formal study of semantics and typing of procedural languages, object calculi [1, 3, 21, 22] are a useful formalism for studying the semantics and typing of object-oriented languages. We begin with a presentation of an untyped prototyping calculus. We fix a countably infinite set of variables $\mathcal{V}$, and a countably infinite set of method labels $\mathcal{L}$. The symbols x and y, with or without subscripting, range over $\mathcal{V}$; the symbol l, with or without subscripts, ranges over $\mathcal{L}$. $\mathcal{V}$ contains a distinguished variable **self**. Terms of the untyped object calculus are defined by the following grammar:

$$t ::= \mathcal{V} |\ \ \mathbf{proto\ self} \ll l_1 = t_1, ..., l_n = t_n \gg |t.l|t \leftarrow\!\!\!+\, l = t|\ \ t\{l := t\}$$

Intuitively, it is useful to think of objects as records, whose members — called methods — can be thought of as functions that accept (a reference to) the entire object — or *self* in object-oriented

[9]By abstracting this variable we get an expression that would be permissible in a system admitting second-order functors.

jargon — as their sole argument; the symbol **self** appears in the bodies of the methods of an object and is bound by the abstraction operator **proto**. The term **proto self**. $\ll l_1 = t_1, \ldots, l_n = t_n \gg$ represents an object with methods $l_1, \ldots, l_n$ of types $\tau_1, \ldots, \tau_n$, respectively. The term $t.l$ represents the invocation of the method, labeled l, of the object denoted by t. The term $t_1 \leftarrow\!\!+\; l = t_2$ represents object extension; it represents the object obtained by extending the object denoted by t_1 by a method labeled l whose body is t_2. The term $t_1\{l := t_2\}$ represents method override; it represents the object obtained by overriding the implementation of the l method of the object (denoted by) t_1, by the method body t_2. Note that this is not an imperative update; it creates a new object in all ways the same as t_1, save the method labeled l that, in the new object is implemented as t_2. Many object calculi[10] in the literature, and specifically the ones discussed here, are "functional" in character and not imperative. We can construct an object with a nonzero number of methods by starting out with an empty object and extending it method after method using the extension construct. However, later in our discussion, we deal with a situation where method extension cannot be supported and therefore the construct **proto self**. $\ll l_1 = t_1, \ldots, l_n = t_n \gg$ is taken as primitive. It is possible to code the untyped λ-calculus in this formalism [3]. However, to make our following examples more readable we work with an extension of the syntax of the preceding terms with λ-calculus constructs.

The operational semantics of the untyped object calculus is given by the following rules:

$$\begin{array}{l}
\textbf{proto self}.\ \ll l_1 = t_1, \ldots, l_n = t_n \gg .l_i \\
\quad \longrightarrow t_i[\textbf{proto self}.\ \ll l_1 = t_1, \ldots, l_n = t_n \gg /\textbf{self}] \\
\textbf{proto self}.\ \ll l_1 = t_1, \ldots, l_n = t_n \gg \leftarrow\!\!+\; l_{n+1} = t_{n+1} \\
\quad \longrightarrow \textbf{proto self}.\ \ll l_1 = t_1, \ldots, l_{n+1} = t_{n+1} \gg \\
\textbf{proto self}.\ \ll l_1 = t_1, \ldots, l_n = t_n \gg \{l_i := s\} \\
\quad \longrightarrow \textbf{proto self}.\ \ll l_1 = t_1, \ldots, l_i = s, \ldots, l_n = t_n \gg
\end{array}$$

and the following grammar of evaluation contexts:

$$E ::= [\,]|\ \ E[\,].l|\ \ E[\,]\{l := t\}|\ \ E[\,] \leftarrow\!\!+\; l = t$$

The first rewrite rule shows the invocation of method l_i; this results in substituting, in the body of the method, the entire object for the self parameter of the method. The second rule shows that object extension results in a new object obtained by adding one more method — with the specified name and body — added to the suite of methods that can be invoked on the object. The third rule shows that method override results in a new object obtained by replacing the method l_i by a new "implementation."

21.6.1.1 Object Types

The question of typing objects, in an object calculus such as the preceding one, has been an important area of study with a rich literature. One line of work has focused on regarding objects as records. To be able to encode object extension, the calculus of records must be extensible (i.e., support a construct to add new fields to the record). Several possible encodings are available for the untyped object calculus in a calculus of extensible records and we merely outline the flavor of some of the most successful encodings. The self-application model [36] encodes an object as a record and each method as a function that accepts a self-parameter; method invocation results in applying the corresponding function to self, that is, the object on which the method is invoked. The recursive record model [15] regards an object as a recursive record, and methods as closures with the self variable bound to the entire record. The treatment of inheritance relies on a calculus of records

[10]A calculus supporting imperative updates is studied in [2].

that supports record concatenation; and, in the typed setting, on a typed λ-calculus with recursive types and a form of polymorphism called F-bounded polymorphism [10]. Although these models are successful in "encoding" the untyped object calculus, unfortunately, they cannot be successfully carried over to typed systems. Specifically, many of the expected typabililty of object-oriented idioms and subtyping relations between object types do not hold if we interpret objects as records according to these encodings [3, 24]. Another attempt to study typing for object calculi by reducing them to typed λ-calculi is the existential model [34, 51]. This model encodes class-based object-oriented languages into $F^{\omega}_{\leq}$, an explicitly typed polymorphic λ-calculus with subtyping. The encoding is cumbersome; for critiques see [1, 3, 21, 24].

Here we present an account based on direct typing rules. What should object types look like? Consider the unidimensional point object:

$$point \;=\; \textbf{proto self}.\ \ll x = 0, move = \lambda dx.\ \textbf{self}\{x := \textbf{self}.x + dx\}\gg$$

This is an object with an x field that contains its location, and a *move* method that takes a numeric displacement argument and moves the object by adding this displacement to the x field. If we call the type of such an object **point**, and let the type of the implicit self parameters of the methods remain implicit, then the type of the move method is **int** $\rightarrow$ **point** and the type of the field x is **int**. Now, consider extending the object *point* by a field *color*. The method move is inherited by the new object from the old; even so, operationally the result of moving a colored point is a colored point, and if **colorpoint** is the type of the extended object, the type of the move method in the extended object ought to be **int** $\rightarrow$ **colorpoint**. This phenomenon is called *method specialization* [45].

Unfortunately, most practical typed object-oriented languages do not support method specialization. The inexpressibility of idioms such as this in practical, typed languages is one reason untyped languages such as Smalltalk are popular. Some languages, such as Self [58] and Eiffel [43], have a notion of *self type*; this is the type of the current object. Recall that **self**, in a method l, denotes the object on which a method is invoked, and this does not need to be the object that defined the method l and from which the current object was derived. In the same vein, the *self type*, **Self**, denotes the type of the object that *self* refers to in a method body l, and this does not need to be the type of the object in which the method l is defined. Thus, the type of *move* should be **int** $\rightarrow$ **Self** that formalizes the intuition that *move* returns an object of the same type as the object it was invoked on. This type **Self** has local meaning only, and thus should be considered bound in the type of the object. This suggests the object type syntax **proto Self**. $\ll l_1 : \tau_1, \ldots, l_n : \tau_n\gg$. This denotes the type of objects with methods $l_1, \ldots, l_n$ of types $\tau_1, \ldots, \tau_n$; the presence of the type **Self** in $\tau_1, \ldots, \tau_n$ refers to the type of the object that inherits the method. Thus, we should consider *point* as having the type **proto Self**. $\ll x : \textbf{int},\ move : \textbf{int} \rightarrow \textbf{Self}\gg$; *colorpoint* would then receive the type **proto Self**. $\ll x : \textbf{int},\ color : \textbf{Color},\ move : \textbf{int} \rightarrow \textbf{Self}\gg$.

21.6.1.2 Typing Rules

The set of explicitly typed raw object terms may be defined by the grammar:

$$t ::= \mathcal{V} \mid\ \lambda x : \tau.\ \ t \mid\ (t\ \ t) \mid\ \textbf{proto self}.\ll l_1 = t, \ldots, l_n = t\gg \ \mid\ t.\,l \mid\ t\{l := t\}$$

As with the simply typed λ-calculus, there are typing judgments of the form $\Gamma \vdash t : \tau$; here Γ is a type environment, t is an explicitly typed term and τ is a type. The rules for typing variables, λ-abstractions and applications are no different from before and, so, are not repeated. The typing rule for method invocation is unsurprising:

$$\frac{\Gamma \vdash t : \textbf{proto Self}.\ll l_1 : \tau_1, ..., l_n : \tau_n \gg}{\Gamma \vdash t.l_i : \tau_i}\ \textbf{(method inv)}$$

Now consider the rule for object extension that motivates the use of self types. The following rule appears to be a candidate:

$$\frac{\Gamma \vdash t : \textbf{proto Self}.\ \ll l_1 : \tau_1, \ldots , l_{n+1} : \tau_{n+1} \gg \qquad \Gamma, \textbf{self} : \tau \vdash t_{n+1} : \tau_{n+1}[\textbf{proto Self}.\ \ll l_1 : \tau_1, \ldots , l_{n+1} : \tau_{n+1} \gg /\textbf{Self}]}{\Gamma \vdash t \leftarrow\!\!\!+\ l_{n+1} = t_{n+1} : \textbf{proto Self}.\ \ll l_1 : \tau_1, \ldots , l_{n+1} : \tau_{n+1} \gg}$$

In the same vein, we have the following rule for method override:

$$\frac{\Gamma \vdash t : \textbf{proto Self}.\ \ll l_1 : \tau_1, \ldots , l_n : \tau_n \gg \qquad \Gamma, \textbf{self} : \tau \vdash t'_k : \tau_k[\textbf{proto Self}.\ \ll l_1 : \tau_1, \ldots , l_n : \tau_n \gg /\textbf{Self}]}{\Gamma \vdash t\{l_k := t'_k\} : \textbf{proto Self}.\ \ll l_1 : \tau_1, \ldots , l_n : \tau_n \gg}$$

Unfortunately, this type system is not sound. Consider the following example from [21]:

Example 21.6
Let:

$$p = \textbf{proto self}.\ \ll x = 3, move = \lambda dx\ .\ \textbf{self}\{x := \textbf{self}.x + dx\} \gg$$
$$q = \textbf{proto self}.\ \ll x = 3, move = \lambda dx\ .\ \textbf{self}\{x := p.move(dx)\} \gg$$

By repeated use of the preceding rules, we can assign the type **proto Self**. $\ll x : \textbf{int},\ move : \textbf{int} \to \textbf{Self} \gg$ to both expressions. However, if q is extended to an object with a *color* field, and *move* is invoked on the resulting object, an object without a *color* field is returned. This means that assigning the type **proto Self**. $\ll x : \textbf{int},\ move : \textbf{int} \to \textbf{Self} \gg$ to q is wrong; the return type of move is not the type of object that q is extended to be. It is left as an exercise to the reader to complete this argument by constructing a typable term expression that can result in a runtime type error when evaluated.

We need some additional machinery to formulate a powerful and sound type system. We sketch the key ideas here; a complete formalization is to be found in [21]. We introduce the notion of a *row*; a row represents a collection of typed method names. A row variable is a variable symbol denoting a row. A row R can be extended with additional typed method names; if R is an expression denoting a row then the expression $R|l : \tau$ represents the collection of methods denoted by R extended with the method l of type τ. The symbol [ ] denotes the empty collection of typed method names. We can generalize the notion of row variables to variables that represent functions from types to rows, from pairs of types to rows and so on. We can build more complex row expressions by abstracting over type variables, and applying row expressions to types. The set of row expressions is generated by the grammar:

$$R ::= \nu_{row}|\ \ [\ ]|\ \ R|l : \tau|\ \ \lambda\alpha.\ R|\ \ R(\tau)$$

where ν_{row} is an infinite set of row variables.

To extend a row by a typed method name, it must first be established that the row to be extended does not contain the method name. This can be achieved by means of a type system. To distinguish types of terms from the types of rows and types we call the latter kinds. The set of kinds is given by the grammar:

$$K := \textbf{Type}|\ \ \textbf{Type}^n \to [l_1, \ldots , l_n]$$

Type is the kind of all types. The kind expression $\textbf{Type}^n \to [l_1, \ldots , l_n]$ denotes row expressions that take n types as arguments returning a row that lacks the methods $l_1, \ldots , l_n$. A first-order row variable has kind $\mathbf{Type}^0 \to [l_1, \ldots, l_n]$ — which is abbreviated as $[l_1, \ldots, l_n]$ — for some set of labels $l_1, \ldots, l_n$ and denotes a row that lacks the methods $l_1, \ldots, l_n$. Let Δ denote a collection of assumptions of the form $r : K$, where $r \in \mathcal{V}_{row}$ and K is a kind. The judgment $\Delta \vdash R : K$ states that row expression R has kind K under the set of assumptions Δ. The rules for inferring judgments of this form are straightforward, and are omitted here; the interested reader may consult [21].

With this machinery in hand we can formulate the following typing rule for object extension:

$$\frac{\begin{array}{c}\Gamma \vdash t : \mathbf{proto\ Self}. \ll R | l_1 : \tau_1, \ldots, l_n : \tau_n \gg \quad \Gamma, \mathbf{self} : \mathbf{Type} \vdash R : [l_1, \ldots, l_{n+1}] \\ \Gamma, r_1 : \mathbf{Type} \to [l_1, \ldots, l_n, l_{n+1}] \vdash t_{n+1} : \tau_{n+1}[\mathbf{proto\ Self}. \ll r_1(\mathbf{Self}) | l_1 : \tau_1, \ldots, l_{n+1} : \tau_{n+1} \gg / \mathbf{Self}] \\ r \text{ not in } t\end{array}}{\Gamma \vdash t \leftarrow\!\!+ \, l_{n+1} = t_{n+1} : \mathbf{proto\ Self}. \ll R | l_1 : \tau_1, \ldots, l_{n+1} : \tau_{n+1} \gg}$$

The first assumption says that t is an object of a type whose methods include $l_1, \ldots, l_n$ with types $\tau_1, \ldots, \tau_n$, respectively; R is a row expression representing the other methods. The second assumption (in conjunction with the first) ensures that the object denoted by t does not contain a method labeled l_{n+1}. The third assumption ensures that in any extension of t, the body of the method to be added, l_{n+1}, has the type τ_{n+1} — where **Self** is replaced by the type of the corresponding extension **proto Self**. $\ll r(\mathbf{Self})\ | l_1 : \tau_1, , \ldots, l_{n+1} : \tau_{n+1} \gg$. The reader is encouraged to revisit the terms p and q in Example 21.6 and to verify that the rule assigns the type **proto Self**. $\ll x : \mathbf{int},\ move : \mathbf{int} \to \mathbf{Self} \gg$ to p, but not to q.

The rule for method override is similar in spirit. The rule is:

$$\frac{\begin{array}{c}\Gamma \vdash t_1 : \mathbf{proto\ Self}. \ll R | l_1 : \tau_1, \ldots, l_n : \tau_n \gg \\ r : \mathbf{Type} \to [l_1, \ldots, l_n] \vdash t' : \tau_i[\mathbf{proto\ Self}. \ll \mathbf{Self} | l_1 : \tau_1, \ldots, l_n : \tau_n \gg / \mathbf{Self}]\end{array}}{\Gamma \vdash t\{l_i := t'\} : \mathbf{proto\ Self}. \ll R | l_1 : \tau_1, \ldots, l_n : \tau_n \gg}$$

Given the rule for object extension, determining the rule for typing **proto Self**. $\ll l_1 = t_1, \ldots, l_n = t_n \gg$ is a simple exercise. This type system is type safe; type soundness is shown in [21].

21.6.1.3 Subtyping

In the context of record and object types two kinds of subtyping relationships are observed. We call a type τ a *width subtype* of τ' if τ has strictly more methods or fields than τ' and the types of the common fields are the same. A type, τ, is said to be a *depth subtype* if it has the same methods or fields, but the type of each field in τ is the same or is (recursively) a depth subtype of the corresponding field in τ'. Of course, both forms of subtyping can occur (and be intertwined) in a subtyping relationship.

Unfortunately, for the object calculus presented so far, no nontrivial subtyping relationships exist among the object types. Consider a method m that is present in τ and not in τ'; an object of type τ' can be extended with method m, whereas an object of type τ may not. In other words, contexts exist where an object of type τ' may be used but objects of type τ may not be. That is, if τ is wider than τ', then τ cannot be a subtype of τ'. This is a fundamental consequence of allowing objects to be extended.

Similarly, we can show that in the presence of method override depth subtyping does not hold. Consider the types $\tau = \mathbf{proto\ Self}. \ll a : \mathbf{proto\ Self}. \ll b : \mathbf{int} \gg, d : \mathbf{int} \gg$ and $\tau' = \mathbf{proto\ Self}. \ll a : \mathbf{proto\ Self}. \ll b : \mathbf{int}, c : \mathbf{int} \gg, d : \mathbf{int} \gg$. Now consider the following:

- $f = \lambda y : \tau.\ y\{a := \mathbf{proto\ self}. \ll b = 0 \gg\}$. The term f has type $\tau \to \tau$.
- $obj = \mathbf{proto\ self}. \ll a = \mathbf{proto\ self}. \ll b = 0, c = 0 \gg, d = x.a.c \gg$. The expression obj has type τ'.

If $\tau \sqsubseteq \tau'$, the application $(f \quad obj)$ would be type correct. This must not be so; the evaluation:

$$\begin{aligned}(f \quad obj) &\longrightarrow obj\{a := \textbf{proto self}.\ \ll b = 0 \gg\} \\ &\longrightarrow \textbf{proto self}.\ \ll a = \textbf{proto self}.\ \ll b = 0 \gg, d = x.a.c \gg \\ &\longrightarrow \textbf{proto self}.\ \ll b = 0 \gg .c\end{aligned}$$

fails because the last object does not have a c field. Intuitively, the reason that depth subtyping leads to type unsafety is that bodies of the various methods depend on one another to possess the types that they currently possess, and this must be preserved by method overrides. Depth subtyping allows a field to be overridden with another whose type is strictly less than what is expected.

Varying degrees of subtyping can be achieved by limiting the object calculus. In [22] two kinds of object types are defined — those that are extensible and those that are not. An extensible object may be sealed — subject to certain conditions on the types involved — yielding an object that is non-extensible. Nonextensible object types exhibit rich subtyping — they exhibit both width subtyping and depth subtyping. In the object calculi developed in [1, 3], method overrides are supported but not method extension. The result is a system where width subtyping is legal. Type inference has been studied for the object calculus of [1, 3] in [49].

21.6.2 Class-Based Languages

The discussion on object-oriented languages so far has been restricted to the delegation model. However, most commonly used object-oriented languages are class based. Many of the insights and results on typing object-based languages can be carried over to class-based languages. A satisfactory encoding must be able to deal with class inheritance, have access restrictions and be compositional. Several different encodings of class-based languages into the object calculus presented so far have been presented, and their relative strengths are studied in [23]. Bruce [8] describes a typed object-oriented language that supports classes directly. POLYTOIL, presented in [9], incorporates imperative features; the paper introduces the notion of matching, a relationship between object types that holds whenever the first is an extension of the second, regardless of the variance of the self-type variable. Eifrig, Smith and Trifonov [20] present a type-safe, class-based, object-oriented language with a rich feature set called I-LOOP; the type system is based on polymorphic recursively constrained types, for which a sound type-inferencing algorithm is presented.

21.6.3 Inheritance and Subtyping

In most commonly used languages, inheritance results in subtyping. For instance, if a Java class B inherits from a class A, then B is a subtype of A. This tight coupling between inheritance and subtyping requires language designers to rule out certain patterns of inheritance to avoid type unsoundness. For instance, consider the class **point** that supports an equality method. Now, derive a class, **colorpoint**, of colored points by adding an extra color field, and overriding the equality method to handle the additional field. Unfortunately, **colorpoint** should not be allowed to be a subtype of **point**; this is because the function definition $f(p : Point) = p.equal(p_0)$, where p_0 is a (colorless) point, can accept a **point** argument but not a **colorpoint** argument. The Eiffel language originally allowed this inheritance; and because in Eiffel inheritance implies subtyping, type safety was compromised. To allow for type safety, some languages (e.g., C++) forbid examples such as this by requiring that the type of an (overridden) method in a derived class be the same as the type of the method in the base class; the Trellis/Owl language only allows subclasses that are also subtypes. This is unfortunate because it forbids a legitimate idiom for code reuse. In [16], the authors argue that inheritance and subtyping are in fact distinct mechanisms for code reuse. They decouple the notions of subtyping and type inheritance, and propose a powerful form of polymorphism (F-bounded polymorphism) based on type inheritance (in contrast to one based on subtyping). The TOOPL [8] language, similarly, decouples inheritance from subtyping: the inheritance relation between object types is distinct from the subtype relation; the former is used to type check whether a class can be derived from another.

21.7 Further Reading

This chapter aims to serve as a tutorial introduction to types in programming languages. Research into the designs of rich and flexible type systems — particularly for object-oriented languages, module systems and languages supporting mobility [40] — are active areas of research; many references cited in this chapter provide good starting points for study of these topics. Also a gap exists between current language design, on the one hand, and the rich type disciplines studied in this chapter, on the other; many of these type disciplines are not amenable to complete type inference, and supplying explicit type information is cumbersome for the programmer. More research is needed to bridge this gap. Significant literature is available on the formal semantics of type disciplines — a topic that could not be covered in this chapter; the book cited in [27] is an excellent starting point. The issue of effective use of types in compilation [19] is another topic that is not discussed in this text: the use of types at runtime (both in the compiled code and in runtime systems such as garbage collectors) and the use of typed intermediate languages in compilers [55, 56] are active research areas.

References

[1] M. Abadi and L. Cardelli, A theory of primitive objects — second-order systems, in *Programming Languages and Systems — ESOP '94, 5th European Symposium on Programming*, D. Sannella, Ed., Vol. 788, *Lecture Notes in Computer Science*, Springer-Verlag, NewYork, 1994, pp. 1–25.

[2] M. Abadi and L. Cardelli, An imperative object calculus (invited paper), *Theor. Pract. Object Syst.*, 1(3), 151–166, 1995.

[3] M. Abadi and L. Cardelli, A theory of primitive objects: untyped and first-order systems, *Inf. Comput.*, 125(2), 78–102, 1996.

[4] A. Aiken and E.L. Wimmers, Type inclusion constraints and type inference, in *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, ACM Press, New York, 1993, pp. 31–41.

[5] R.M. Amadio and L. Cardelli, Subtyping recursive types, *ACM Trans. Programming Languages Syst.*, 15(4), 575–631, 1993.

[6] H. Barendregt, *The Lambda-Calculus*, North-Holland, Amsterdam, 1981.

[7] V. Breazu-Tannen, T. Coquand, C.A. Gunter and A. Scedrov, Inheritance as implicit coercion, in *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, C.A. Gunter and J.C. Mitchell, Eds., Foundations of Computing Series, MIT Press, Cambridge, MA, 1994, pp. 197–245.

[8] K.B. Bruce, Static Type Checking in Statically-Typed Object-Oriented Programming Language, in Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages, 1993, pp. 285–298.

[9] K.B. Bruce, A. Schuett and R. van Gent, PolyTOIL: A type-safe polymorphic object-oriented language, in *ECOOP '95 — Object-Oriented Programming, 9th European Conference*, W.G. Olthoff, Ed., *Lecture Notes in Computer Science*, Springer-Verlag, New York, Vol. 952, 1995, pp. 27–51.

[10] P. Canning, W. Cook, W. Hill, J. Mitchell and W. Olthoff, F-bounded quantification for object-oriented programming, in *Functional Programming and Computer Architecture*, 1989, pp. 273–280.

[11] L. Cardelli and P. Wegner, On understanding types, data abstraction, and polymorphism, *Computing Surv.*, 17(4), 471–522, 1985.

[12] J.C. Mitchell, Coercion and Type Inference (Summary), in Proceedings of the 11th ACM Symposium on Principles of Programming Languages, January 1984, pp. 175–185.

[13] J.C. Mitchell, Type inference with simple subtypes, *J. Functional Programming*, 1(3), 245–286, 1991; preliminary version appeared in Proceedings of the 11th ACM Symposium on Principles of Programming Languages, 1984, pp. 175–185.

[14] J.C. Mitchell and G.D. Plotkin, Abstract types have existential types, *ACM Trans. Programming Languages Syst.*, 10(3), 470–502, 1988; preliminary version appeared in Proceedings of the 12th ACM Symposium on Principles of Programming Languages, 1985.

[15] W. Cook, The Semantics of Inheritance, Ph.D. thesis, Brown University, Providence, RI, 1988.

[16] W. Cook, W. Hill and P. Canning, Inheritance is not subtyping, in *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, C.A. Gunter and J.C. Mitchell, Eds., Foundations of Computing Series, MIT Press, Cambridge, MA, 1994, pp. 427–459.

[17] P.-L. Curien and G. Ghelli, Coherence of subsumption, minimum subtyping and type-checking in $F_{\leq}$, in *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, C.A. Gunter and J.C. Mitchell, Eds., Foundations of Computing Series, MIT Press, Cambridge, MA, 1994, pp. 247–292.

[18] L. Damas and R. Milner, Principal type-schemes for functional programs, in *Proceedings of the 9th ACM Symposium on Principles of Programming Languages*, R. DeMillo, Ed., January 1982, ACM Press, New York, 1982, pp. 207–212.

[19] X. Leroy, Ed., *Proceedings of the Types in Compilation Workshop, Lecture Notes in Computer Science*, Vol. 1473, Springer-Verlag, New York, 1998.

[20] J. Eifrig, S. Smith and V. Trifonov, Type inference for recursively constrained types and its application to OOP, in *Mathematical Foundations of Programming Semantics, New Orleans*, Vol. 1, *Electronic Notes in Theoretical Computer Science*, Elsevier, Amsterdam, 1995, *www.elsevier.nl/locate/entcs/volume1.html.*

[21] K. Fisher, F. Honsell and J.C. Mitchell, A lambda calculus of objects and method specialization, *Nordic J. Computing (*formerly *BIT)*, 1, 3–37, 1994; preliminary version appeared in Proceedings of the IEEE Symposium on Logic in Computer Science, 1993, pp. 26–38.

[22] K. Fisher and J.C. Mitchell, A delegation-based object calculus with subtyping, in *Lecture Notes in Computer Science*, Vol. 965, Springer-Verlag, New York, 1995, p. 42.

[23] K. Fisher and J.C. Mitchell, The development of type systems for object-oriented languages, *Theor. Pract. Object Syst.*, 1, 189–220, 1996; preliminary version appeared in *Proceedings of the International Symposium on Theoretical Aspects of Computer Software, Lecture Notes in Computer Science*, Vol. 789, Springer-Verlag, New York, 1994, pp. 844–885.

[24] K. Fisher and J.C. Mitchell, On the relationship between classes, objects and data abstraction, *Theor. Pract. Object Syst.*, 4(1), 1998.

[25] Y.-C. Fuh and P. Mishra, Type inference with subtypes, in *2nd European Symposium on Programming, Nancy*, *Lecture Notes in Computer Science*, Vol. 300, Springer-Verlag, New York, 1988, pp. 94–114.

[26] J.-Y. Girard, Interprétation Fonctionelle et Élimination des Compures de l'Arithmétic d'Ordre Supérieur, Ph.D. thesis, Université Paris VII, 1972.

[27] C.A. Gunter, *Semantics of Programming Languages*, MIT Press, Cambridge, MA, 1992.

[28] R. Harper, B. Duba and D. MacQueen, Typing first-class continuations in ML, *J. Functional Programming*, 1994.

[29] R. Harper and M. Lillibridge, A Type-Theoretic Approach to Higher-Order Modules with Sharing, in Proceedings of the 21st ACM Symposium on Principles of Programming Languages, Portland, OR, 1994, pp. 123–137.

[30] R. Harper, J.C. Mitchell and E. Moggi, Higher-order modules and the phase distinction, in *Proceedings of the 17th ACM Symposium on Principles of Programming Languages*, ACM, Ed., ACM Press, New York, 1990, pp. 341–354.

[31] R. Harper, M. Tofte and R. Milner, *The Definition of Standard ML; Version 2*, MIT Press, Cambridge, MA, 1980.

[32] N. Heintze, Control-flow analysis and type systems, *Lect. Notes Comput. Sci.*, 983, 1995.

[33] F. Henglein, Type inference with polymorphic recursion, *ACM Trans. Programming Languages Syst.*, 15(2), 253–289, 1993.

[34] M. Hofmann and B.C. Pierce, A unifying type-theoretic framework for objects, *J. Functional Programming*, 1994.

[35] T. Jim and J. Palsberg, Type inference in systems of recursive types with subtyping, manuscript, 1999.

[36] S. Kamin, Inheritence in Smalltalk-80: A Denotational Definition, in Proceedings the 15th ACM Symposium on Principles of Programming Languages, 1988, pp. 80–87.

[37] P. Kannelakis and J. Mitchell, Polymorphic unification and ML typing, in *Proceedings of the 16th ACM Symposium on Principles of Programming Languages*, ACM Press, New York, 1989, pp. 105–115.

[38] X. Leroy, Manifest Types, Modules, and Separate Compilation, in Proceedings of the 21st ACM Symposium on Principles of Programming Languages, 1994.

[39] X. Leroy, Polymorphism by Name for References and Continuations, in Proceedings of the 20th ACM Symposium on Principles of Programming Languages, New York, 1993, pp. 220–231.

[40] G. Ghelli, L. Cardelli and A. Gordon, Mobility types for mobile ambients, *Lect. Notes Comput. Sci.*, 1644, 1999.

[41] D.B. MacQueen, Using Dependent Types to Express Modular Structure, in Proceedings 13th ACM Symposium on Principles of Programming Languages, 1986, pp. 277–286.

[42] D.B. MacQueen and M. Tofte, A semantics for higher-order functors, *Programming Languages and Systems — ESOP '94, 5th European Symposium on Programming*, D. Sannella, Ed., *Lecture Notes in Computer Science*, Vol. 788, Springer-Verlag, 1994, pp. 409–423.

[43] B. Meyer, *Eiffel: The Language*, Prentice Hall, Englewood Cliffs, NJ, 1991.

[44] J.C. Mitchell, Representation Independence and Data Abstraction (Preliminary Version), in *Conference Record of the 13th Annual ACM Symposium on Principles of Programming Languages, St. Petersburg Beach, FL*, ACM, January 1986, pp. 263–276.

[45] J.C. Mitchell, Toward a Typed Foundation for Method Specialization and Inheritance, in Proceedings of the 17th ACM Symposium on Principles of Programming Languages, 1990, pp. 109–124.

[46] A. Mycroft and R. O'Keefe, A polymorphic type system for PROLOG, *Artif. Intelligence*, 23(3), 295–307, 1984.

[47] M. Odersky and K. Laufer, An Extension of ML with First-Class Abstract Types, in ACM SIGPLAN Workshop on ML and its Applications, San Francisco, June 1992.

[48] P. O'Keefe and M. Wand, Type inference for partial types is decidable, in *ESOP '92, 4th European Symposium on Programming*, B. Krieg-Brückner, Ed., *Lecture Notes in Computer Science*, Vol. 582, Springer-Verlag, New York, 1992, pp. 408–417.

[49] J. Palsberg, Efficient inference of object types, *Inf. Computation*, 123(2), 198–209, 1995.

[50] J. Palsberg and P. O'Keefe, A type system equivalent to flow analysis, *ACM Trans. Programming Languages Syst.*, 17(4), 576–599, July 1995.

[51] B.C. Pierce and D.N. Turner, Simple type-theoretic foundations for object-oriented programming, *J. Functional Programming*, 4(2), 207–247, 1994.

[52] J. Reynolds, Using category theory to design implicit conversions and generic operators, in *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, C.A. Gunter and J.C. Mitchell, Eds., Foundations of Computing Series, MIT Press, Cambridge, MA, 1994, pp. 197–245.

[53] J.C. Reynolds, An introduction to the polymorphic lambda calculus, in *Logical Foundations of Functional Programming*, G. Huet, Ed., University of Texas Year of Programming Series, Addison-Wesley, Reading, MA, 1990, pp. 77–86.

[54] J.A. Robinson, A machine-oriented logic based on the resolution principle, *J. ACM*, 12(1), 23–41, 1965.

[55] Z. Shao, An Overview of the FLINT/ML Compiler, in Proceedings 1997 ACM SIGPLAN Workshop on Types in Compilation (TIC '97), Amsterdam, 1997.

[56] D. Tarditi, G. Morrisett, P. Cheng, R. Harper, C. Stone and P. Lee, TIL: A type-directed optimizing compiler for ML, *ACM SIGPLAN Notices*, 31(5), 181–192, May 1996; Proceedings of the 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation.

[57] S. Thatte, Type inference with partial types, in *Automata, Languages and Programming, 15th International Colloquium*, T. Lepistö and Arto Salomaa, Eds., *Lecture Notes in Computer Science*, Vol. 317, Springer-Verlag, New York, 1988, pp. 615–629.

[58] D. Ungar and R.B. Smith, SELF, the power of simplicity, *Lisp Symbolic Computation,* 4(3), July 1991, 187–205; preliminary version appeared in Proceedings of the ACM Symposium on Object-Oriented Programming: Systems, Languages, and Applications, 1987, pp. 227–241.

[59] P. Wadler and S. Blott, How to make ad-hoc polymorphism less ad hoc, in *Proceedings of the 16th Symposium on Principles of Programming Languages*, ACM Press, New York, 1989, pp. 60–76.

[60] P. Weis, M.V. Aponte, A. Laville, M. Mauny and A. Suárez, The CAML Reference Manual, INRIA-ENS, 1987.

[61] J.B. Wells, Typability and Type Checking in the Second-Order Lambda-Calculus are Equivalent and Undecidable, in Proceedings of the IEEE Symposium on Logic in Computer Science, 1994, pp. 176–185.

[62] A. Wright, Polymorphism for Imperative Languages without Imperative Types, Technical report TR93-200, Rice University, Houston, TX, February 21, 1996.

22
An Introduction to Operational Semantics

Sanjiva Prasad
Indian Institute of Technology, Delhi

S. Arun-Kumar
Indian Institute of Technology, Delhi

22.1 Introduction

The Objective of this chapter is to introduce to compiler developers the rudimentary concepts of *operational semantics* used in specifying the operational behavior of programs and systems, and for reasoning about them. There are already various excellent comprehensive introductions to syntax-directed approaches to operational semantics, most notably the seminal papers by Plotkin [63] and Kahn [38]. Some of that material has already been incorporated in standard text books on the semantics of programming languages and concurrency, such as those by Winskel [76], Gunter [25], Watt [73] and Hennessy [30]. Yet, though the concepts and techniques employed are mathematically simple and accessible, many compiler developers have not been exposed to them.

The material presented here is largely based on the seminal works mentioned above, and is aimed at presenting the ideas in an integrated form. It is tutorial in nature, oriented toward those interested

0-8493-1240-7/03/$0.00+$1.50

in relating language specification to compiler design. There are also several excellent surveys and references on research aspects in operational semantics, particularly in the context of semantics of computation [57] and of process algebra [2], intended for those who are already familiar with semantics issues in programming languages and concurrency.

Operational semantics involves giving a precise description of the behavior of a program or a system, namely, how it may execute or operate. As in any semantic enterprise, the intention in developing operational semantics is to give behavioral descriptions in rigorous mathematical terms, in a form that supports understanding and reasoning about the behavior of the systems under consideration. A mathematical model serves as the basis for analysis and verification. In fact, the very act of formalization can help remove misconceptions and focus attention on subtleties that may be glossed over in an informal description.

A clear operational semantics is an invaluable reference while developing language implementations, as was recognized over a quarter of a century ago by McCarthy [45], Landin [40, 42], Hoare and Lauer [32], Milner, Plotkin and various other researchers. Early examples of real-world languages provided with formal operational semantics include Algol 60 [43] and PL/I [60].

Formalism, per se, is not the only goal; defining the meaning of a programming language as the behavior induced by a particular implementation is a formal treatment. However, such an approach is not particularly satisfactory because the intention is to provide behavioral descriptions at a high level, divorced from implementation details to as great an extent as possible. Moreover, the high-level formalism should be readily accessible. Indeed, the attraction of using operational semantic approaches to programming languages is the relative simplicity of the formal mathematics and the associated techniques.

The past 20 or so years have seen, following seminal contributions by Plotkin [63], Milner [48, 49], Kahn [38], Hoare [35] and others, the development of syntax-directed "structural" frameworks that provide, to quote Plotkin, a "simple and direct method for specifying the semantics of programming languages," which require very little mathematical background, yet provide "concise, comprehensible semantic definitions." The definition of the mostly functional language Standard ML in a wholly operational semantic framework [56] is an excellent example of the power and versatility as well as the relative accessibility of these operational techniques. Other languages that have complete operational descriptions are Esterel [7, 8, 23] and Ada, (Reference [3] contains an early definition of Ada that employs the main ideas discussed here in a rigorous algebraic framework).

Although formalization is clearly important for research in programming language semantics, the aim of this chapter is to make modern approaches to operational semantics accessible to those involved in compiler design and development. It is therefore worthwhile to reiterate here why formal operational descriptions are important in the context of compiler design and implementation. As mentioned earlier, such descriptions provide an unambiguous definition of a language, which can serve as a reference for implementations. A structural operational semantics (SOS) style seems to be an increasingly favored style of providing comprehensive and comprehensible formal definitions of programming languages. Apart from the examples of Standard ML and Esterel cited earlier, SOS semantics have been provided for several languages including Java [17] and logic programming languages based on Prolog [31]. Also, these formal descriptions allow us to develop theories such as program equivalence or orderings, which serve as a semantically sound basis for assessing proposed program optimizations and static analysis techniques. Although it may be naive to expect the algebraic laws of equivalence (or ordering) to suggest optimizations, it is nevertheless expected that any optimization preserves the operational behavior of a program (or at least the important behavioral properties needed in the context of a particular computation). In addition, the operational descriptions give us a framework in which compiler verification can be formulated and carried out [20]. Finally, operational frameworks allow us to explore novel, alternative implementation techniques — by studying different abstract implementations that realize the same specifications. A noteworthy approach in this respect is that of Hannan [28] and his collaborators.

Structural operational techniques have been employed with great success for studying the correctness of compiler techniques and hardware implementations [70, 74, 77], for compiler verification [20, 36], for establishing type soundness following the work of Wright and Felleisen [75], for static program analysis [55] and for deriving proof rules for functional languages [65].

We also should mention that there are areas of crucial importance to compiler developers where operational techniques have not been seriously applied. An example is floating point computation, where to our knowledge, the intricacies of the numerical models proposed and used have not been adequately addressed in an operational framework.

22.1.1 Operational Descriptions at Different Levels of Abstraction

Formal operational descriptions of program execution can be presented in several different ways. In fact, having different descriptions may serve a useful purpose, especially because they are usually presented at varying levels of abstraction. In the following paragraphs, we give a brief overview of three broad levels of operational description, which have historically tended (roughly) to go from low-level to high-level descriptions for the same language, though notable exceptions have existed where implementations have been guided by higher-level specifications.

The very first step in providing a description of a language independent of any particular implementation is to concentrate on the abstract syntactic structure of programs in the language rather than the concrete syntax. This also has the advantage of having the ability to abstract over different concrete renderings of a concept in different languages (e.g., the syntax used for assignment in Pascal vs. that used in C). A relatively higher level semantic description than a particular implementation is achieved by translation of the abstract syntax into instructions of a simple machine, the description of which is given in abstract terms, typically as a finite collection of rules. Such an idealized machine is called an *abstract machine*.

Reasoning about programs using abstract machine descriptions consists of reasoning about the process of translation, and then reasoning about execution sequences of the abstract machine. A significant observation that greatly simplified the first aspect was the following: the abstract syntax of most languages is inductively characterized, and the translation to the machine instructions tends to be a mapping that preserves the abstract syntactic structure, often a homomorphic function.

A good early example of this kind of operational description is Landin's use of the so-called SECD machine to specify the operational semantics of a quintessential (call-by-value) functional language ISWIM [41]. Also well known is the Warren abstract machine (WAM) [72], used to specify the execution machinery for Prolog. Abstract machines are a popular (and often the first) method for specifying the execution semantics of a proposed language as well as for outlining an implementation. For instance, abstract machines were used in presenting the first formal operational descriptions of various extensions to the functional paradigm such as integrations of functional programming with concurrent programming models based on ideas from process algebra [13, 14, 22].

Although abstract machines provide higher level, implementation-independent specifications of program execution, it is not always clear how effective such techniques are in proving program properties, proving notions of program equivalence and developing a semantically justified algebra of programs. Moreover, proofs about program execution are (often tedious and cumbersome) induction arguments on execution sequences, using case analyses on which rule is employed at each step, with little reference to the original source programs and their structure.

A second and novel step was the development of SOS, where program behavior was expressed directly in terms of the source programs (and perhaps a few ancillary data structures) without any intervening translation to an abstract machine. The structural approach consists of providing an inductive definition of a relation describing program execution, which follows the inductive structure of the abstract syntax. Thus, in the operational setting, the approach adheres to a compositionality principle associated with Frege that "the meaning of a phrase can be obtained from the meaning of its components in a well-defined way," a feature of the Scott–Strachey style of denotational semantics. The standard presentation of the inductively defined relation is by using inference rules. The consequent of a rule defines a transition from a compound expression, which depends on the transitions for one or more of its components specified in the rule antecedent. This inductive approach based on abstract syntactic structure is also appropriate for formulating static semantics. An added bonus of using relations is that features such as nontermination and partiality, nondeterminism, error configurations and various others can easily be accommodated into the framework without having to resort to more difficult mathematical concepts.

The associated proof techniques are based on induction on the proof trees built using the inference rules, or equivalently — because the inference rules are presented in a syntax-directed manner — on the structure of the source program. Notions of program equivalence or ordering are stated directly in terms of the source programs rather than of via any other machinery, and thus the development of an algebra of programs gets facilitated. It is this aspect of structural induction that justifies the moniker "structural," because the other techniques also ultimately depend on program structure.

The pioneering works where the structural approach is articulated are those of Plotkin [63, 64], Milner [50] and Kahn [38]. However, instances of the structural approaches predate these publications — most notably the operational semantics of various λ-calculi [6]. Structural semantics come in a variety of flavors, and we broadly classify them as (1) "big-step," often called *natural* due to its connections with normalization in natural deduction proof systems [9, 38], and sometimes relational [56] or proof-theoretical [46]; and (2) "small-step," which is often called *reduction* following the terminology used in the λ-calculus [63]. Big-step semantics justify a complete execution sequence using a tree-structured proof whereas small-step semantics provide tree-structured justifications for each step of the sequence. However, situations exist where a "mixed-step" formulation is convenient. In contrast, abstract machine semantics consists of a sequence of steps, each justified as being an instance of a conditional rewrite rule.

Yet another dimension in the varieties of structural operational semantics is the use of labeled relations that allow the specification of the interaction between a program and its environment during execution. Most examples of labeled relations are in a small-step style, and abstract machines rarely use labeled relations at all.

One of the aims of this chapter is to convey to the reader the rudiments of these three kinds of operational semantics and their interrelationships and important syntactic properties, such as confluence and standardization. We endeavor to present these notions in frameworks that are as simple and familiar as possible, and assuming minimal concepts. Various aspects of these connections have been studied in great detail elsewhere, assuming varying degrees of familiarity with the concepts. Plotkin [63] covers a large variety of constructs in the reduction semantics framework. Some subtle issues arising in relating the big-step and small-step formulations are explored in [5]. Winskel's book [76] studies the relation between big-step and denotational semantics for simple imperative and functional languages. Hannan and Miller [46] present a framework for constructing abstract machines from big-step semantics for functional languages via a series of correctness-preserving transformations. Hannan further explores concrete realizations of the machines [27]. Plotkin [61] studies the connection between the reduction semantics of the call-by-value λ-calculus and its abstract machine (and for call-by-name, respectively), as well as how the calculi relate to one another by continuation-passing style (CPS) translations. An excellent reference covering much of this material in detail is [1].

22.1.2 Disclaimers

This chapter does not attempt to survey the variety of operational semantics frameworks used in the specification and implementation of programming languages. In particular, two major approaches have been neglected — those of action semantics [54] and evolving algebras, or abstract state machines [26]. Action semantics is based on ideas from universal algebra, and seeks to combine the salient positive features of denotational and operational approaches, without their weaknesses. The semantic specification is given around the basic actions in a system, and the approach addresses the important issues of readability and modularity of semantic frameworks. Even small language extensions sometimes necessitate major changes in the semantic rules. Such wholesale changes are avoided in Action semantic frameworks, which are naturally modular. Action semantics has been successfully used in diverse applications, being a very significant one in the area of compilation the work of Palsberg on provably correct compiler generation [58].

Evolving algebras, or abstract state machines as they are now called, are based on the idea of interpreting the dynamic semantic actions of a system as operators of an algebra that evolves during execution. The approach is very general and permits specification of a system at different levels of abstraction. The operational framework is closely related to conditional rewriting systems, and the theory also addresses the mathematical issue of algebraic models for rules. Furthermore, abstract state machine descriptions admit parallelism (concurrency) in an extremely natural way. They have been used extensively for describing a variety of systems and languages, such as Prolog [11] and Modula-2 [53], apart from use as a vehicle for understanding various concurrency features of Ada and other such intricacies.

We also concentrate on only three paradigms — imperative, functional and concurrent — and do not address issues in logic programming and object-oriented programming. We also do not examine seriously the issues that arise when different paradigms are integrated in a single language.

22.1.3 Relationship with Other Kinds of Semantics

An alternative to operational techniques for specifying the semantics of programming languages is providing mathematical models (i.e., denotational semantics); well-known textbooks on denotational semantics are [66, 68]. Denotational frameworks are also specified inductively on abstract syntax. The attraction of denotational methods is that they provide rich mathematical theory for reasoning about programs. Moreover, when the denoted objects are readily constructible in a computational framework, the semantics can be viewed as providing an immediate implementation of the language.

However, two questions immediately arise when providing a language with a denotational model. First, is such a model in (complete) agreement with operational intuition? Milner was the first to propose a criterion, called *full abstraction*, which formalizes this notion of complete agreement between the two forms of semantics. He convincingly argues that the operational semantics should be the reference (the "touchstone") for assessing mathematical models, rather than the converse, because operational models are (usually) set up with minimal preconceptions. The second question is whether there is indeed a unique mathematical model? Milner points out that any mathematical model can capture only some aspects of the operational behavior, whereas there may be diverse aspects that can be of interest — especially in nondeterministic computations. Because operational frameworks are relational, they can easily accommodate aspects such as nondeterminism, partiality, and erroneous computations with minimal reworking of definitions, whereas these may necessitate significant changes to the mathematical models used in a denotational description.

Another alternative to the operational approach is the so-called *axiomatic semantics* [33] in which the meaning of a programming construct is given using proof rules within a program logic. The orientation of the approach is toward proving program correctness with respect to logical specifications. Again, one could argue for the primacy of operational techniques to interpret and justify the soundness of the logical rules. Moreover, the formulation of operational semantics using inference rules in the SOS approaches together with the induced algebraic notions of equivalence or ordering on programs incorporates many aspects of the axiomatic approach into operational ones — compositionality, syntax orientation and proof theory, in particular.

It must be noted, however, that the three approaches are not mutually exclusive or conflictive. Each finds use when reasoning about programs, and often while employing a particular kind of approach, one may resort to another. For instance, while reasoning about the operational semantics (proving metatheorems), it may be convenient to use results from the denotational semantics because this enables one to abstract away irrelevant operational details and to use abstract mathematical concepts.

22.1.4 Structure of This Chapter

The rest of the chapter is structured as follows. In Section 22.2, we introduce various important rudimentary concepts used in describing the operational behavior of systems. We start with the notion of transition systems, and then proceed to providing meaning to the abstract syntax trees of expressions. We use a simple language of expressions to illustrate three different levels of operational description. We enrich the language with variables and then scoped local definitions. Section 22.3 presents the operational semantics for a simple imperative language. Various extensions of this language to incorporate nondeterminism and parallel execution, block structure, simple procedures and storage allocation are discussed. In Section 22.4, we discuss descriptions of higher order functions, referring to the λ-calculus and two evaluation strategies — call-by-name and call-by-value — together with environment machines for implementing these calculi. Then, in Section 22.5, we mention features of languages involving concurrency and interaction that are naturally modeled using labeled transitions, before concluding in Section 22.6.

22.2 Preliminaries

22.2.1 Transition Systems

The primary task involved in providing an operational description of a system is to specify the configurations of the system and the possible transitions between configurations. A *transition system* (TS) consists of a collection (usually a set) $\mathcal{S}$ of configurations and a binary relation on configurations $\longrightarrow \subseteq \mathcal{S} \times \mathcal{S}$ called the *transition relation*. We use the metavariable s to range over configurations. In most applications, a subset $\mathcal{I} \subseteq \mathcal{S}$, called *initial*, or *starting* configurations, is distinguished. *Terminal configurations* are those from which a transition is not possible — $\{s \in \mathcal{S} \mid \nexists s' : s \longrightarrow s'\}$. We denote the transitive closure of the transition relation by $\longrightarrow^{+}$ and its reflexive transitive closure by $\longrightarrow^{*}$. Termination arguments often require showing that the transition relation is well-founded.

A closely related notion is that of a labeled transition system (LTS). Let $\mathcal{L}$ be a set of labels, with l a typical label. An LTS consists of a set of configurations $\mathcal{S}$, the label set $\mathcal{L}$ and a relation $\_ \xrightarrow{\_} \_ \subseteq \mathcal{S} \times \mathcal{L} \times \mathcal{S}$ called the *labeled transition relation*. We write $s \xrightarrow{l} s'$ to mean that $\langle s, l, s' \rangle \in \_ \xrightarrow{\_} \_$. Often an LTS is presented as a collection of TSs sharing the same configurations $\mathcal{S}$, but with one transition relation for each label.

Example 22.1: Lexical Analysis

Lexical analysis can be cast as a TS. Let $\mathcal{M} = \langle Q, q_0 \in Q, \delta \subseteq Q \times \Sigma \times Q, F \subseteq Q \rangle$ be a finite state automaton recognizing a language over alphabet Σ, and let $\varsigma \in \Sigma^*$ be any string over that alphabet. Let ϵ denote the empty string, and let $a\varsigma$ denote the string starting with letter a followed by the suffix string ς.

Let $\mathcal{S} = Q \times \Sigma^*$ and let $\longrightarrow$ be defined as $\langle q,\ a\ \varsigma\rangle \longrightarrow \langle q',\ \varsigma\rangle$ if and only if $(q, a, q') \in \delta$. $\mathcal{I} = \{\langle q_0,\ \varsigma\rangle \mid \varsigma \in \Sigma^*\}$ is the set of initial configurations. Terminal configurations are of two kinds: those in $F \times \{\epsilon\}$ are "accepting" whereas those in $(Q - F \times \{\epsilon\}) \bigcup \{\langle q,\ b\ \varsigma\rangle \mid \neg\exists q' : (q, b, q') \in \delta\}$ are "nonaccepting."

Example 22.2: An Automaton is an LTS
Finite state (and indeed other) automata are examples of LTSs. The configurations $\mathcal{S}$ are the states of the automation, the labels the alphabet Σ and δ the labeled transition relation.

The example of automata also motivates a bunch of concepts important in operational semantics. We usually associate a notion of observation with a transition system (e.g., consumption of a string and termination in an accepting state in a finite state automaton), with respect to which transition systems are ascribed observable behaviors (e.g., strings accepted by the automaton). There can be different notions of what is observable for even the same transition system. Any given notion of observability yields a corresponding notion of equivalence or ordering between two transition systems based on their observable behaviors.

DEFINITION **22.1 (observational equivalence and ordering).** *TS_1 is said to be observably simulatable by TS_2, written $TS_1 \preceq TS_2$, if every observable behavior possible of TS_1 is also possible of TS_2. TS_1 and TS_2 are considered equivalent, denoted $TS_1 \approx TS_2$, if both have the same observable behaviors.*

Equivalence of two systems does not necessarily imply that one can be replaced by the other in any context, because some notions of equivalence may not be preserved under each and every construction possible in a class of transition systems.

The automata example also gives an idea of how an LTS can relate to a TS. The automaton LTS describes the control aspect of the TS in abstraction from the data (the string ς) on which it is run. The dichotomy between control and data is not the central issue, however. Rather, labels are used to indicate interaction between a component of a larger system with its context. This interaction can be of a variety of kinds, and hence diverse uses of labeled TSs exist. For example, a process receiving signals and performing some computation in response can be specified separately from the processes sending it signals. The use of labeled transitions permits the description of a component behavior separately from that of its context, with the labels specifying the interaction capabilities. Very crudely, a labeled TS can be turned into a corresponding unlabeled one by providing within the system "enough context" — thus "closing up" a description of an open system. Conversely, contexts can be used to label transitions. The main issue is to characterize interesting decompositions of systems into program fragment and context. This is still very much the subject of active research, with some recent promising results in this direction [44, 67].

Example 22.3: Parsing
We also encounter a TS in parsing. String generation can be thought of as a TS as follows. Let $\mathcal{G} = \langle \mathsf{N}, \mathsf{T}, \mathsf{P}, S \in \mathsf{N} \rangle$ be a context-free grammar. Let the configurations $\mathcal{S} = (\mathsf{N} \cup \mathsf{T})^*$ and let the transition relation $\longrightarrow$ be defined as $s \longrightarrow s'$ if and only if there exists a production $r \in \mathsf{P}$ such that $r \equiv X \rightarrow w$ for some $X \in \mathsf{N}$ and $w \in (\mathsf{N} \cup \mathsf{T})^*$, $s = s_1 X s_2$ and $s' = s_1 w s_2$. S is the unique starting configuration, and those terminal configurations that are in T^* and reachable from S are the generated strings.

This TS can be reversed to yield a TS for parsing. The production rules are used in the reverse direction; $\mathcal{I} = \mathsf{T}^*$ is the set of initial configurations, with a single "accepting" final configuration S and possibly many other terminal configurations that are "nonaccepting."

REMARK **22.1.** Plotkin's seminal paper [63, Chapter 1] lists several different examples of TSs or labeled TSs that one encounters in computer science — finite state automata, transducers, grammars of different types, k-counter machines, stack machines, Petri nets, Turing machines, Semi-Thue systems, Post-Systems, L-systems, Conway's Game of Life, push-down automata, tree automata, cellular automata and neural nets. In addition, many dynamic systems we encounter in daily life may be modeled as TSs. Games are good examples of TSs.

22.2.1.1 Properties

TSs provide a framework on which we can drape various formal verification exercises. Many of these involve establishing that a particular TS satisfies various kinds of properties. One such important property is totality. A TS is total if it has no terminal configurations (i.e., for every $s \in \mathcal{S}$ there exists $s' \in \mathcal{S}$ such that $s \longrightarrow s'$. Another common property is determinism: for every $s \in \mathcal{S}$, $|\{s' \mid s \longrightarrow s'\}| \leq 1$. These notions can also extend to labeled transitions, either "per label" or "across labels."

A crucial property in the preceding examples for lexical and grammatical analysis is reachability from designated initial configurations. Reachability is also used in proving safety properties of systems — no bad configuration is reachable from specified initial configurations.

Another property is what we call *properly terminating*, where all terminal configurations are good. This is an example of a liveness property — that something good can eventually happen.

Another important metaproperty is confluence: for any $s, s_1, s_2 \in \mathcal{S}$, whenever $s \longrightarrow^* s_1$ and $s \longrightarrow^* s_2$, then $s_3 \in \mathcal{S}$ exists such that $s_1 \longrightarrow^* s_3$ and $s_2 \longrightarrow^* s_3$. Stronger confluence properties are the so-called *diamond* properties. A TS exhibits the *strong diamond* property if for any $s, s_1, s_2 \in \mathcal{S}$, whenever $s \longrightarrow s_1$ and $s \longrightarrow s_2$ and $s_1 \neq s_2$, then $s_3 \in \mathcal{S}$ exists such that $s_1 \longrightarrow s_3$ and $s_2 \longrightarrow s_3$. A TS has a *weak diamond* property if whenever $s \longrightarrow s_1$ and $s \longrightarrow s_2$ and $s_1 \neq s_2$, then s_3 is reachable from s_1 and s_2 via the reflexive transitive closure of the transition relation, that is, $s_1 \longrightarrow^* s_3$ and $s_2 \longrightarrow^* s_3$.

Various properties follow from certain finiteness constraints on transition systems. A TS (or LTS) is called:

- *Finitely branching* if for every $s \in \mathcal{S}$, the set $\{s' \mid s \longrightarrow s'\}$ is finite
- *Finite* if it is finitely branching and $\longrightarrow$ is a well-ordering
- *Regular* if it is finitely branching and for each $s \in \mathcal{S}$, the set $\{s' \mid s \longrightarrow^* s'\}$ is finite, where $\longrightarrow^*$ is the reflexive transitive closure of $\longrightarrow$

In general, TSs whose transition relation can be characterized in a concise but abstract manner (usually as a set of rules) are of interest, because they usually admit effective techniques for establishing properties of those systems. Finite or inductively characterized TSs are extremely common, with induction and case analysis on (linear) sequences of transitions the most widely wielded proof methods for reasoning about execution sequences or, at a higher level, observable behavior.

22.2.2 Structural Operational Semantics for Expressions

22.2.2.1 Abstract Syntax

The abstract instead of the concrete syntax of a language is of interest while specifying the meaning of programs. Operational semantics descriptions manipulate these abstract syntactic objects and work wholly within syntax. For convenience, however, it may be necessary to augment the syntax with extrasyntactic data structures, but these entities can be shown to correspond in some obvious way to purely syntactic entities. The abstract syntax of programs can be inductively characterized (e.g., as trees). We use abstract grammars as a handy notational device for describing abstract syntactic categories.

We present three different kinds of operational description for an extremely simple language *Exp*; the presentation can be adapted to any language of first-order expressions.

Example 22.4: Simple Arithmetic Expressions

Let *Num* denote the denumerable set of numerals (in some radix), and let $\mathcal{X}$ be a denumerable set of variables, with x, y, z typical metavariables ranging over $\mathcal{X}$. *Exp* can be presented using the following abstract grammar, where e, e_1, e_2 are metavariables ranging over *Exp*, and n ranges over *Num*:

$$e \in Exp \; :: \; = x \mid n \mid (e_1 + e_2)$$

TABLE 22.1 Big-Step Semantics for Evaluating Simple Expressions

(var)	$\dfrac{}{\gamma \vdash x \Longrightarrow_e \gamma(x)}$	where $x \in dom(\gamma)$
(num)	$\dfrac{}{\gamma \vdash n \Longrightarrow_e n}$	
(add)	$\dfrac{\gamma \vdash e_1 \Longrightarrow_e n_1 \quad \gamma \vdash e_2 \Longrightarrow_e n_2}{\gamma \vdash (e_1 + e_2) \Longrightarrow_e n_3}$	where $n_3 = ADD(n_1, n_2)$

Expression evaluation consists of simplifying a given expression to a form that cannot be further simplified, hopefully to an element in a set of "good" canonical forms that we loosely call *values* (there are a variety of notions of value depending on the language). The first task in presenting operational semantics for expressions is to identify the set $\mathcal{V}$ of values. In the next few examples the set $\mathcal{V}$ is the set of numerals *Num*. The metavariable v ranges over $\mathcal{V}$.

For expressions containing variables, we need to know what the variables stand for to simplify them to values. Accordingly, we present the operational semantics with respect to a finite domain function called an *environment* γ: $\mathcal{X} \rightarrow_{fin} \mathcal{V}$, that maps variables to values. Let *Env* denote the set of such finite domain functions from variables to values.[1] Environments are an example of extrasyntactic constructions we employ in our operational description. We write $dom(\gamma)$ to mean the set $\{x \in \mathcal{X} \mid \gamma(x) \text{ defined}\}$. We work with finite domain functions because it is inappropriate to frame essentially syntactic ideas in terms of infinite structures. If γ_1 and γ_2 are finite domain functions, we denote by $\gamma_1[\gamma_2]$ the finite domain function with domain $dom(\gamma_1) \cup dom(\gamma_2)$ defined as:

$$\gamma_1[\gamma_2](x) = \begin{cases} \gamma_2(x) & \text{if } x \in dom(\gamma_2) \\ \gamma_1(x) & \text{if } x \in dom(\gamma_1) - dom(\gamma_2) \\ \text{undefined} & \text{otherwise} \end{cases}$$

22.2.2.2 Big-Step or Natural Semantics

We first present a big-step structural operational semantics, or natural semantics for *Exp*.

The big-step transition relation $\Longrightarrow_e \subseteq Env \times Exp \times \mathcal{V}(= Num)$ is defined inductively as the smallest relation closed under the inference rules given in Table 22.1. We read the relation $\gamma \vdash e \Longrightarrow_e n$ as "given environment γ, expression e can evaluate to value n." When the environment γ is not needed, and so can be arbitrary, we sometimes omit writing "$\gamma \vdash$."

This relation can be viewed as a TS with configurations $\mathcal{S} = (Env \times Exp)$. A transition $\gamma \vdash e \Longrightarrow_e v$ is understood as a transition $\langle \gamma,\ e \rangle \rightarrow \langle \gamma,\ v \rangle$, highlighting the fact that transitions leave γ unchanged.

The way these rules are used is that if we have an expression that matches the left side of the consequent (denominator) of a rule via a substitution ρ for the schematic variables, and if using the same substitution ρ, all the antecedents (statements in the numerator) can be inductively established while also respecting any side conditions, then the expression can evaluate to an expression of the form given on the right side of the consequent instanceed using ρ. Used in this manner, the rules can be seen as forming tree-structured justifications, or *proof trees*, of why an expression e can evaluate

[1]It is also possible to work with environments that are finite domain functions from variables to variable-free expressions instead of to values. The nature of the rules and results does not change, except perhaps in some minor details.

to a value n — the goal judgment ($e \Longrightarrow_e n$ in this case) is at the root, the leaves are axiom instances and the internal nodes correspond to rule instances with a branch for each antecedent.

The use of proof rules to specify transition systems is itself an area of research. Aceto, Fokkink and Verhoef [2] give an excellent summary of rule specifications, the meanings of the transition systems they specify and of various formats and the formal properties they guarantee (see also [47]).

Observe that the rules are syntax directed, in that there is a rule for each syntactic case. Further, in rules with antecedents, the consequent of the rule describes the evaluation of a compound expression; this evaluation depends on the evaluation of the component subexpressions, described in the rule's antecedents. The base cases of the relation $\Longrightarrow_e$ are the axioms (*num*) and *var*, which state that any numeral evaluates to itself because it is in canonical form, and that a variable evaluates to the value associated with it in the environment, respectively. Note, however, that instances of the rule *var* apply only when the side condition or *proviso* $x \in dom(\gamma)$ holds. The induction case is the rule (*add*). The rule may be read as "given γ, expression $(e_1 + e_2)$ can evaluate to numeral n_3 if expression e_1 can evaluate to a numeral n_1 with respect to γ, and e_2 to n_2 also with respect to γ, and where adding numerals n_1 and n_2 yields numeral n_3." We assume a syntactic routine *ADD* for adding numerals.

Note: The big-step relation is reflexive on values. The relation is not total on environments and *Exp*, because the *var* rule does not specify how to evaluate a variable $y \notin dom(\gamma)$.

Typical exercises involve studying various properties of this relation. For instance, assuming that the procedure *ADD* is functional and total, we can show that the relation $\Longrightarrow_e$ is indeed a partial function:

$$|\{n \mid \gamma \vdash e \Longrightarrow_e n\}| \leq 1 \ \text{ for all } \ \gamma \in Env \ \text{ and } \ e \in Exp$$

If $vars(e)$ is the set of variables in e, we can show:

Proposition 22.1. *For any $e \in Exp$, $\gamma \in Env$, if $vars(e) \subseteq dom(\gamma)$, there exists $n \in Num$ such that $\gamma \vdash e \Longrightarrow_e n$.*

Proof of this proposition is by induction on the structure of the proof tree of $\gamma \vdash e \Longrightarrow_e n$, which amounts to induction on the structure of e, because the relation is syntax directed.

Further, we can show that the big-step operational semantics agrees with any standard denotational semantics if the procedure *ADD* behaves in accordance with the corresponding mathematical operation. Let ρ be an assignment of values to variables, let $[\![n]\!]$ denote the number represented by numeral n and let $[\![e]\!]\rho$ be the denotation of e with respect to ρ.

Proposition 22.2. *For any $e \in Exp$, γ, ρ such that $vars(e) \subseteq dom(\gamma)$ and for all $x \in dom(\gamma)$, $\rho(x) = [\![\gamma(x)]\!]$: $\gamma \vdash e \Longrightarrow_e n$ if and only if $[\![e]\!]\rho = [\![n]\!]$.*

This result too is proved by induction on the structure of e.

22.2.2.3 Small-Step or Reduction Semantics

The big-step relation specifies what normal forms an expression may have. It is a high-level specification, is possibly nondeterministic and does not detail how the computation may be performed. It is inherently parallel; for example, in simplifying $(e_1 + e_2)$, no indication is given as to whether to simplify e_1 before e_2 or otherwise. Nor is any hint given on how to implement the relation with finite resources.

In contrast, a small-step, or reduction, relation is used to specify not merely what an evaluation may return, but also a strategy to achieve it. This approach is essentially the stepwise rewriting approach followed, for example, in junior school when teaching children to simplify arithmetic expressions, with the strategy specifying which subexpressions may be simplified at any stage.

TABLE 22.2 Small-Step Semantics for Arithmetic Expressions

(vbl)	$\dfrac{}{\gamma \vdash x \longrightarrow_1^e \gamma(x)}$	provided $x \in dom(\gamma)$
(add_0)	$\dfrac{}{\gamma \vdash (n_1 + n_2) \longrightarrow_1^e n_3}$	where $n_3 = ADD(n_1, n_2)$
(add_l)	$\dfrac{\gamma \vdash e_1 \longrightarrow_1^e e_1'}{\gamma \vdash (e_1 + e_2) \longrightarrow_1^e (e_1' + e_2)}$	
(add_r)	$\dfrac{\gamma \vdash e_2 \longrightarrow_1^e e_2'}{\gamma \vdash (e_1 + e_2) \longrightarrow_1^e (e_1 + e_2')}$	

Again, configurations are simple arithmetic expressions: $\mathcal{S} = Env \times Exp$ and $\mathcal{V} = Num$. The small-step relation $\longrightarrow_1^e \subseteq Env \times Exp \times Exp$ is between two expressions, given an environment. The important difference with big-step semantics is that expressions do not simplify "in one go" to a value, but instead simplify one step at a time to other expressions, and perhaps finally to values. The reduction relation is also defined inductively, using inference rules, which are syntax directed, but in a sense slightly different from that in the big-step semantics. Several rules may exist for the same syntactic construct, and some constructs may have no associated rules. Moreover, the case analysis is not strictly on syntactic structure but instead on an analysis of where simplification can take place in an expression. Small-step reduction relations are seldom transitive and are usually irreflexive.

Table 22.2 displays a reduction relation for evaluating simple arithmetic expressions. The rule (*vbl*) says that variables are simplified to the value specified in the given environment. As expected, the rule has a proviso requiring that the variable be in the domain of the environment. Note that no rule exists for numerals. The rule (add_0) can be understood as saying that $(n_1 + n_2)$ simplifies to the result of $ADD(n_1, n_2)$. The rules (add_l) and (add_r) are symmetrical; the former says that if e_1 can simplify to e_1', then $(e_1 + e_2)$ can simplify to $(e_1' + e_2)$ in a single step (similarly for simplifying e_2 first). Note also that the relation is nondeterministic, and involves localized rewriting.

Observe that it is possible for an expression, such as $((7 + 21) + y)$ where $y \notin dom(\gamma)$ for a given environment γ, to be reduced a few steps before it gets "stuck." This is in contrast to the big-step situation where no transition is possible for that expression with respect to such an environment γ.

An expression of the form $(n_1 + n_2)$, an instance of the left side in an axiom, is called a *redex*. Any reducible expression can be shown to contain a redex. Different small-step relations may be proposed that differ in which redex should be selected first for reduction.

Typical results about small-step semantics usually pertain to the reflexive transitive closure of the reduction relation. For instance, we can show the agreement with the big-step semantics:

Proposition 22.3. *For all $e \in Exp$, $\gamma \in Env$ and $n \in Num$: $\gamma \vdash e\ (\longrightarrow_1^e)^*\ n$ if and only if $\gamma \vdash e \Longrightarrow_e n$.*

This and similar results are proved by induction on the number of reduction steps involved in $\gamma \vdash e\ (\longrightarrow_1^e)^*\ n$, and within each reduction step, by an induction on the depth of the proof tree justifying the single reduction step. A corollary to the preceding proposition is that the reduction-down-to-values relation is a (partial) function, though such results can be shown from first principles without reference to the big-step semantics.

A more interesting result to show about the relation $\longrightarrow_1^e$ is whether it satisfies a strong diamond property. The proof of this property is by structural induction on the original expression, and analysis on how it could reduce to different expressions using induction on these justifications. This confluence result provides a direct proof that while reduction is nondeterministic, the input-output relation it induces is a function. (Totality of the input-output function is often shown by proving that a reduction relation is well-ordered.)

A confluence result can greatly simplify reasoning about program execution, because it essentially says that we do not need to consider each possible sequence but merely any one sequence to a point of confluence. Confluence properties can play an important role in compilation, because confluent systems admit simplifications in any order, including strategies that involve simplification of subexpressions in parallel or even in nondeterministic fashion; these may make sense in certain architectures such as those involving pipelining or multiple computational units. Nonconfluence should alert a compiler developer that a proposed optimization may in fact be unsound if it alters reduction order and ought therefore be avoided.

The small-step framework admits various restricted versions of reduction corresponding to specialized strategies, typically those that are deterministic or easier to implement. For instance, we could replace the (add_r) rule by more restrictive versions, for example:

$$(add_r^{lseq}) \qquad \frac{\gamma \vdash e_2 \longrightarrow_1^e e_2'}{\gamma \vdash (n + e_2) \longrightarrow_1^e (n + e_2')}$$

which allow simplification of the second summand only when the first is already a numeral. With these more restrictive rules, the reduction relation becomes deterministic; for any expression at most one reduction rule applies. The modified relation specifies a sequential left-to-right evaluation strategy. It is then important to prove that this strategy can simulate the original relation correctly in the sense that both relations have the same reflexive transitive closures when considering reductions down to values. This result is an example of standardization: if an expression can be reduced to a value by any strategy, it can be reduced by a standard sequence using a particular strategy.[2]

Standardization is useful in reasoning about program execution, because it allows one to transform any sequence of reductions to another one about which it is somehow easier to reason. Standardization results are often employed, for instance, in showing that certain reduction sequences are not possible. They can be important to a compiler writer, because they permit the use of possibly more efficient implementation strategies without having to sacrifice any generality. It must be emphasized that standardization is a very important syntactic metatheorem of TSs that applies only in systems whose extensional behavior (input–output) is deterministic.

Example 22.5

Phenomena such as nontermination sharpen the differences between various evaluation strategies. Consider a simple language of possibly nonterminating Boolean expressions given by the abstract grammar:

$$b := tv \mid \Omega \mid (b_1 \vee b_2) \qquad tv \in \{\mathbf{true}, \mathbf{false}\}$$

We define three different small-step relations (omitting the "$\gamma\ \vdash$" in the rules): $\longrightarrow_1^{comp}$ that evaluates all parts of a disjunctive Boolean expression:

$$\frac{}{\Omega \longrightarrow_1^{comp} \Omega} \qquad \frac{}{(tv_1 \vee tv_2) \longrightarrow_1^{comp} tv_3} \quad tv_3 = OR(tv_1, tv_2)$$

$$\frac{b_1 \longrightarrow_1^{comp} b_1'}{(b_1 \vee b_2) \longrightarrow_1^{comp} (b_1' \vee b_2)} \qquad \frac{b_2 \longrightarrow_1^{comp} b_2'}{(tv \vee b_2) \longrightarrow_1^{comp} (tv \vee b_2')}$$

[2]Richer languages may require more complicated standardization results.

$\longrightarrow_1^{ls}$ which is a left sequential evaluation:

$$\frac{}{\Omega \longrightarrow_1^{ls} \Omega} \qquad \frac{b_1 \longrightarrow_1^{ls} b_1'}{(b_1 \vee b_2) \longrightarrow_1^{ls} (b_1' \vee b_2)}$$

$$\frac{}{(\mathbf{true} \vee b) \longrightarrow_1^{ls} \mathbf{true}} \qquad \frac{}{(\mathbf{false} \vee b_2) \longrightarrow_1^{ls} b_2}$$

and $\longrightarrow_1^{par}$ which is parallel evaluation

$$\frac{}{\Omega \longrightarrow_1^{par} \Omega}$$

$$\frac{b_1 \longrightarrow_1^{par} b_1'}{(b_1 \vee b_2) \longrightarrow_1^{par} (b_1' \vee b_2)} \qquad \frac{b_2 \longrightarrow_1^{par} b_2'}{(b_1 \vee b_2) \longrightarrow_1^{par} (b_1 \vee b_2')}$$

$$\frac{}{(b_1 \vee \mathbf{false}) \longrightarrow_1^{par} b_1} \qquad \frac{}{(\mathbf{false} \vee b_2) \longrightarrow_1^{par} b_2}$$

$$\frac{}{(b_1 \vee \mathbf{true}) \longrightarrow_1^{par} \mathbf{true}} \qquad \frac{}{(\mathbf{true} \vee b_2) \longrightarrow_1^{par} \mathbf{true}}$$

If a Boolean expression b reaches normal form via $\longrightarrow_1^{comp}$, then it reaches the same normal form via $\longrightarrow_1^{ls}$, in which case it reaches the same normal form via $\longrightarrow_1^{par}$. However, the converse is not true:

$$(\mathbf{true} \vee \Omega) \longrightarrow_1^{par} \mathbf{true}$$

and:

$$(\Omega \vee \mathbf{true}) \longrightarrow_1^{par} \mathbf{true}$$

but:

$$(\mathbf{true} \vee \Omega) \longrightarrow_1^{ls} \mathbf{true}$$

whereas:

$$(\Omega \vee \mathbf{true}) \longrightarrow_1^{ls} (\Omega \vee \mathbf{true})$$

However, both:

$$(\mathbf{true} \vee \Omega) \longrightarrow_1^{comp} (\mathbf{true} \vee \Omega)$$

and:

$$(\Omega \vee \mathbf{true}) \longrightarrow_1^{comp} (\Omega \vee \mathbf{true})$$

22.2.2.4 Environment-Free Formulations

We pause briefly to remark that the formulation of the preceding relations using environments can be transformed to TSs that operate wholly within syntax. For this we need the notion of substitution.

DEFINITION 22.2 **(Substitution).** *A substitution σ is a finite domain function from $\mathcal{X}$ to Exp. Equivalently, it can be viewed as a total function that is almost everywhere identity. We write $e\sigma$ to denote applying σ to e yielding an expression obtained by simultaneously replacing in e every occurrence of variable x by the expression $\sigma(x)$ for each $x \in vars(e)$.*

An environment γ is a specific instance of a substitution. It can easily be shown that if $\gamma \vdash e\ (\longrightarrow_1^e)^*,\ n$ then $\vdash e\gamma\ (\longrightarrow_1^e)^*\ n$ (the variable-free case) and likewise for $\Longrightarrow_e$.

This observation may cause you to wonder why we introduced environments in the first place. The reason is that substitution is usually an expensive operation, whereas the environment data structure allows the computation to "look up" the expression to be substituted for a variable as and when it is needed. Moreover, the later sections show that environments arise naturally when we try to implement languages with block structure and functions. The environment-less formulation eases the presentation of the following notion of equality.

22.2.2.5 Operational Notions of Equality

Given a small-step relation such as $\longrightarrow_1^e$, it is often natural to define a notion of equality $=^e$ on expressions as the symmetrical reflexive transitive closure of the reduction relation. This is precisely the idea taught in junior school to show that two arithmetic expressions are equal.

Definition 22.3 (equality). *Expression* $e \ =^e \ e'$ *if there is a sequence of expressions* $e_1, \ldots, e_n$ *such that* $e \equiv e_1$, $e' \equiv e_n$ *and for each* $i : 1 \leq i \leq n-1$, *either* $e_i \ \longrightarrow_1^e \ e_{i+1}$ *or* $e_{i+1} \ \longrightarrow_1^e \ e_i$.

If the $\longrightarrow_1^e$ relation is weakly confluent, e and e' can be reduced to a common form.

22.2.2.6 Abstract Machines

A more common approach to specifying arithmetic expression evaluation, familiar to most computer scientists after an introductory data structures course, is by using a stack machine. This semantics is at a lower level than either the big-step or small-step semantics, because it departs from providing a specification of evaluation directly in terms of the source syntax, and also it employs additional data structures.

The opcodes of the machine are instructions for loading numerical constants, for adding numerals and for looking up bindings of variables. To avoid introducing new symbols, we employ the same symbols for the opcodes of the machine. Let $OpCodes$ be defined as sequences (strings) over the symbol +, numerals, variables in $\mathcal{X}$, with the idea that a variable is a lookup operation.[3]

$$OpCodes = (Num \cup \mathcal{X} \cup \{+\})^*$$

Consider now a postorder traversal of the abstract syntax tree of an expression in *Exp*. This is defined as a recursive function $compile : Exp \to OpCodes$. To enhance readability, we use ˆ to indicate string catenation:

$$compile(n) = n$$
$$compile(x) = x$$
$$compile((e_1 + e_2)) = compile(e_1)\hat{}compile(e_2)\hat{}+$$

Configurations of the abstract machine are triples consisting of an environment, a "stack" of numerals and a sequence of opcodes. Table 22.3 details the initialization and transitions (the relation $\longrightarrow\!\!\triangleright$) of the abstract machine. Observe that we have presented a (finite) set of possibly conditional rewrite rules in a two-dimensional syntax. The rules are operated by taking any configuration that matches via a substitution for the schematic variables (e.g., $\gamma, c, S, n, \ldots$), the pattern indicated in the left side of a rule, and replacing it with the configuration obtained by applying the same substitution to the right side of a rule. In this example the rewrite rules involved are deterministic and "regular," in that at most one rule applies and that no configuration can be rewritten to more than one configuration.

The machine is initialized with a given environment γ with respect to which expression e is to be evaluated, an empty stack and a sequence of opcodes corresponding to $compile(e)$. (For readability we have used the ML-like notation :: for sequence concatenation, writing e.g., $+ :: C'$ to specify a sequence beginning with + followed by sequence C'). Observe that no inference rules are available — merely rewrite rules, which are applied repeatedly until no rule applies. The moves depend primarily on the first opcode in the sequence. The "good" terminal states are those with an empty sequence

[3]In implementations, we can have a single opcode that is parametrized by a variable (or equivalently an address or index corresponding to the variable), and similarly a single opcode for loading constants.

TABLE 22.3 Evaluating Expressions Using an Abstract Stack Machine

$load(\gamma, e) \;=\; \left\langle \gamma,\ \left\lfloor \quad \right\rfloor,\ compile(e) \right\rangle$		
variables	$\langle \gamma,\ S,\ x :: C \rangle \;\mathrel{\relbar\!\!\rhd}\; \left\langle \gamma,\ \left\lfloor \begin{array}{c} \gamma(x) \\ S \end{array} \right\rfloor,\ C \right\rangle$	
constants	$\langle \gamma,\ S,\ n :: C \rangle \;\mathrel{\relbar\!\!\rhd}\; \left\langle \gamma,\ \left\lfloor \begin{array}{c} n \\ S \end{array} \right\rfloor,\ C \right\rangle$	
add	$\left\langle \gamma,\ \left\lfloor \begin{array}{c} n_2 \\ n_1 \\ S \end{array} \right\rfloor,\ + :: C \right\rangle \;\mathrel{\relbar\!\!\rhd}\; \left\langle \gamma,\ \left\lfloor \begin{array}{c} n_3 \\ S \end{array} \right\rfloor,\ C \right\rangle$	where $n_3 = ADD(n_1, n_2)$
$unload\left(\left\langle \gamma,\ \left\lfloor\ n\ \right\rfloor,\ \epsilon \right\rangle\right) = n$		

of opcodes and a single value on the stack, from which the results are "unloaded." The operational behavior induced by the abstract machine is exactly the same as the big-step $\Longrightarrow_e$ (and thus also the closure of the small-step relation).

PROPOSITION 22.4. *For all $e \in Exp$, $\gamma \in Env$ and $n \in Num$: $\gamma \;\vdash\; e \;\Longrightarrow_e\; n$ if and only if there exists a configuration s such that $load(\gamma, e)\ (\mathrel{\relbar\!\!\rhd})^*\ s$ and $unload(s) = n$*

The proof involves induction on e and on the number of $\mathrel{\relbar\!\!\rhd}$ steps in reaching the terminal configuration. In fact, several nontrivial lemmas need to be shown, which essentially state that the evaluation of an expression does not examine or disturb the part of the stack below its initial top, and that any expression results in a single value on the stack.

A typical result that has to be shown is along the lines of "for *any* stack S and code list C', if:

$$\left\langle \gamma,\ \left\lfloor S \right\rfloor,\ compile(e)\hat{\ }C' \right\rangle\ (\mathrel{\relbar\!\!\rhd})^*\ \left\langle \gamma,\ \left\lfloor S' \right\rfloor,\ C' \right\rangle$$

then $\lfloor S' \rfloor \;=\; \left\lfloor \begin{array}{c} n \\ S \end{array} \right\rfloor$ for some $n \in Num$." The proof is by induction on the length of the opcode sequence, but observe that we need to explicitly involve all "contexts" in which an expression may be evaluated — the universal quantification on all stacks S and "continuation" code C' — in the statement of this property.

The preceding abstract machine can be seen as an implementation of a left-to-right reduction. In general, standardization results help mediate the relationship between the abstract machine semantics and the reduction semantics.

22.2.2.7 Tuples, Records and Conditionals

We make a quick foray into giving rules for structured expressions. We consider pairs (the idea extends easily to tuples and records) and a simple conditional (which generalizes to *case* statements). We only point out that in the rules for conditionals, the test e_1 is first evaluated to a truth value before one of the branches e_2 or e_3 is selected.

We assume that our values $v ::= n \mid tv \mid \langle v_1,\ v_2\rangle$. The big-step rules for pairs and conditionals are:

$$(pair)\quad \frac{\gamma \vdash e_1 \Longrightarrow_e v_1 \quad \gamma \vdash e_2 \Longrightarrow_e v_2}{\gamma \vdash \langle e_1,\ e_2\rangle \Longrightarrow_e \langle v_1,\ v_2\rangle}$$

$$(if_t)\quad \frac{\gamma \vdash e_1 \Longrightarrow_e \textbf{true} \quad \gamma \vdash e_2 \Longrightarrow_e v_2}{\gamma \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \Longrightarrow_e v_2}$$

$$(if_f)\quad \frac{\gamma \vdash e_1 \Longrightarrow_e \textbf{false} \quad \gamma \vdash e_3 \Longrightarrow_e v_3}{\gamma \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \Longrightarrow_e v_3}$$

One possible set of small step rules are:

$$(pair_l)\quad \frac{\gamma \vdash e_1 \longrightarrow_1^e e_1'}{\gamma \vdash \langle e_1,\ e_2\rangle \longrightarrow_1^e \langle e_1',\ e_2\rangle}$$

$$(pair_r)\quad \frac{\gamma \vdash e_2 \longrightarrow_1^e e_2'}{\gamma \vdash \langle e_1,\ e_2\rangle \longrightarrow_1^e \langle e_1,\ e_2'\rangle}$$

$$(if_0)\quad \frac{\gamma \vdash e_1 \longrightarrow_1^e e_1'}{\gamma \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \longrightarrow_1^e \textbf{if } e_1' \textbf{ then } e_2 \textbf{ else } e_3}$$

$$(if_l)\quad \frac{}{\gamma \vdash \textbf{if true then } e_2 \textbf{ else } e_3 \longrightarrow_1^e e_2}$$

$$(if_r)\quad \frac{}{\gamma \vdash \textbf{if false then } e_2 \textbf{ else } e_3 \longrightarrow_1^e e_3}$$

We do not present the abstract machine rules but observe that new opcodes need to be introduced, and the compile function extended. The rule for a n-tuple-formation opcode takes n values off the stack forms an n-tuple that is then pushed onto the stack. A record formation operation requires a little more jugglery, for example, by sorting the fields according to a particular order (lexicographic, say) at the compilation stage, and traversing the abstract syntax tree accordingly. The opcode for conditional choice picks one of two continuations based on the value on the top of the stack. At an abstract level, it is possible to talk of compound opcodes $IF(c_1, c_2)$, which are realized on actual machines by jumps. Another trick, used by Plotkin [63, p. 18] as a motivating illustration for advocating more structure in operational descriptions, is to take the syntax apart and stash the continuations or markers, selecting the correct one based on the evaluation of e_1.

22.2.3 Private Definitions

Tennent's principle of qualification [69] suggests that *Exp* can be extended to include expressions that employ locally scoped definitions:

$$e \ :: \ = \ \cdots \mid \textbf{let } x \overset{def}{=} e_1 \textbf{ in } e_2$$

In **let** $x \overset{def}{=} e_1$ **in** e_2, the scope of the definition of x to e_1 is limited to e_2. The occurrences of variables in an expression are now of two kinds: those that are bound and those that are free. Define $fv\colon Exp \to \mathcal{X}$ as:

$$fv(x) = x \quad fv(f(e_1, \ldots, e_k)) = \bigcup_{i=i}^{k} fv(e_i)$$

$$fv(c) = \emptyset \quad fv(\textbf{let } x \overset{def}{=} e_1 \textbf{ in } e_2) = fv(e_1) \bigcup (fv(e_2) - \{x\})$$

The big-step rules are extended with:

$$\frac{\gamma \vdash e_1 \Longrightarrow_e n_1 \quad \gamma[x \mapsto n_1] \vdash e_2 \Longrightarrow_e n_2}{\gamma \vdash \textbf{let}\, x \overset{def}{=} e_1 \,\textbf{in}\, e_2 \Longrightarrow_e n_2}$$

Possible small-step rules are:

$$\frac{\gamma \vdash e_1 \longrightarrow_1^e e_1'}{\gamma \vdash \textbf{let}\, x \overset{def}{=} e_1 \,\textbf{in}\, e_2 \longrightarrow_1^e \textbf{let}\, x \overset{def}{=} e_1' \,\textbf{in}\, e_2}$$

$$\frac{\gamma[x \mapsto n_1] \vdash e_2 \longrightarrow_1^e e_2'}{\gamma \vdash \textbf{let}\, x \overset{def}{=} n_1 \,\textbf{in}\, e_2 \longrightarrow_1^e \textbf{let}\, x \overset{def}{=} n_1 \,\textbf{in}\, e_2'}$$

$$\frac{}{\gamma \vdash \textbf{let}\, x \overset{def}{=} n_1 \,\textbf{in}\, n_2 \longrightarrow_1^e n_2}$$

We postpone the presentation of an abstract machine that can correctly deal with scoping issues to our discussion of functions in Section 22.4, because the machinery needed there subsumes that needed here. Tennent's principle of correspondence relates definition mechanisms to parameter-passing and thus definition mechanisms get addressed in the operational semantics for function definition and call. It suffices to mention at this stage that the machines will now additionally have to stack environments (or structures containing them) to implement the lexical scoping of block structured languages.

Instead we discuss compound definitions. Consider the syntactic category *Defs* with meta-variable d:

$$d \ ::= \ x \overset{def}{=} e \mid d_1;d_2 \mid d_1 \| d_2$$

with $dv\colon \mathit{Defs} \to \mathcal{X}$ returning the defined variables, and fv extended to *Defs*:

$$dv(x \overset{def}{=} e) = \{x\} \quad dv(d_1;d_2) = dv(d_1) \bigcup dv(d_2) \quad dv(d_1 \| d_2) = dv(d_1) \uplus dv(d_2)$$

$$fv(x \overset{def}{=} e) = fv(e) \quad fv(d_1;d_2) = fv(d_1) \bigcup (fv(d_2) - dv(d_1))$$

$$fv(d_1 \| d_2) = fv(d_1) \bigcup fv(d_2)$$

Here $\uplus$ stands for disjoint union, defined only when the sets are actually disjoint.

The big-step semantics uses two mutually recursive (but nevertheless inductive) definitions: $\Longrightarrow_e$ as before and $\Longrightarrow_d \ \subseteq \mathit{Env} \times \mathit{Defs} \times \mathit{Env}$. Observe here that the values (canonical forms) for the $\Longrightarrow_d$ transition system are *environments*, which are extrasyntactic. The rules for $\Longrightarrow_d$ are:

$$\frac{\gamma \vdash e \Longrightarrow_e n}{\gamma \vdash x \overset{def}{=} e \Longrightarrow_d [x \mapsto n]}$$

$$\frac{\gamma \vdash d_1 \Longrightarrow_d \gamma_1 \quad \gamma[\gamma_1] \vdash d_2 \Longrightarrow_d \gamma_2}{\gamma \vdash d_1;d_2 \Longrightarrow_d \gamma_1[\gamma_2]}$$

$$\frac{\gamma \vdash d_1 \Longrightarrow_d \gamma_1 \quad \gamma \vdash d_2 \Longrightarrow_d \gamma_2}{\gamma \vdash d_1 \| d_2 \Longrightarrow_d \gamma_1 \cup \gamma_2}$$

To correctly implement scoping, $\Longrightarrow_d$ returns the incremental change to the environment obtained by processing a definition. In sequential definitions we first process d_1 with respect to γ, which we augment with the resulting environment to process d_2, whereas with simultaneous definitions, the same environment is used for elaborating the parallel definitions. In the last rule, because we assumed that $dv(d_1) \bigcap dv(d_2) = \emptyset$, the union of environments is well-defined.

Finally, because the principle of qualification may be applied to *Defs*, we obtain definitions that contain auxiliary local definitions:

$$d ::\ = \ldots \mid \textbf{local}\ d_1\ \textbf{in}\ d_2$$

$$dv(\textbf{local}\ d_1\ \textbf{in}\ d_2) = dv(d_2) \quad fv(\textbf{local}\ d_1\ \textbf{in}\ d_2) = fv(d_1) \bigcup (fv(d_2) - dv(d_1))$$

The big-step semantics of this construct is:

$$\frac{\gamma \vdash d_1 \Longrightarrow_d \gamma_1 \quad \gamma[\gamma_1] \vdash d_2 \Longrightarrow_d \gamma_2}{\gamma \vdash \textbf{local}\ d_1\ \textbf{in}\ d_2 \Longrightarrow_d \gamma_2}$$

The reduction semantics for *Defs* is somewhat more tricky (see [63, pp. 80–81]. The problem can perhaps be understood in trying to reduce **let** d **in** e when d is irreducible. Here, the bindings of d must somehow be augmented to the outer environment γ before e can be evaluated. Plotkin employs the expedient of treating environments as a canonical form of definitions, clarifying that they are not in the abstract syntax but merely in the control component in configurations:

$$\frac{\gamma[\gamma_1] \vdash e \longrightarrow_1^e e'}{\gamma \vdash \textbf{let}\ \gamma_1\ \textbf{in}\ e \longrightarrow_1^e \textbf{let}\ \gamma_1\ \textbf{in}\ e'}$$

Indeed, this mixing of extrasyntactic data structures (environments) with abstract syntax is a somewhat weak point about pure reduction semantics. Although the big-step formulations also use extrasyntactic constructions, their use is far more disciplined (Astesiano points out that various denotational-semantic relations can be presented in the same inductive framework employed by big-step semantics) [5].

22.2.3.1 Relation to Types

We finish this section with an important issue of how the operational semantics relates to type checking. Indeed, our presentation has avoided typing issues altogether, although they are a significant part of any structural semantic presentation. The relationship between typing and execution is particularly significant in strongly typed languages with compile-time type checking: Programs that type check correctly at compile time should not raise type errors at run time. This property can be guaranteed if expressions do not change type during execution. Such a theorem is called *subject reduction*. A typical subject reduction result (stated for small-step semantics, but an analogous statement holds for big-step semantics) is:

Let Γ be a set of assumptions of types of variables under which expression e has type τ (written $\Gamma \vdash e : \tau$). If γ is an environment that conforms to Γ (i.e., it binds variables to values having type according to Γ), and if $\gamma \vdash e \longrightarrow_1^e e'$, then $\Gamma \vdash e' : \tau$.

22.3 Imperative Languages

22.3.1 WHILE Language

We now move on to providing a simple imperative language WHILE with operational semantics. WHILE has nested within it a language of expressions (boolean expressions, in particular) and the operational semantics provides a good illustration of how semantics developed for one syntactic category can be employed in the inductive definition of another — transitions for expressions are employed in those for imperative commands.

22.3.1.1 Syntax

The syntax of commands $Comm$ in WHILE with typical metavariable c is given by the following abstract grammar, where metavariable e ranges over *Exp*, which we assume includes a sublanguage of boolean expressions:

$$\begin{array}{rcl} c & ::= & \textbf{skip} \\ & | & x{:=}e \\ & | & c_1;c_2 \\ & | & \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \\ & | & \textbf{while } e \textbf{ do } c \end{array}$$

22.3.1.2 Big-Step Semantics

The big-step semantics of WHILE is a relational specification of command execution. The imperative model of computation is based on the idea of making a series of small changes to a memory state. Commands can be thought of as state transformers — the basic action is that of assigning a value to a program variable. More complex actions are built up from the elementary ones using constructs for sequencing, conditional execution and iteration. For convenience, we include an identity transformation, namely, the command **skip**.

Let *State* consist of finite domain functions from $\mathcal{X}$ to $\mathcal{V}$. For simplicity, we assume that expression evaluation involves no side effects that change the state of memory. *State* is, at least as a first approximation, the same as *Env*. This abstraction gets taken apart in modeling other features. The set of configurations in the transition system is $(State \times Comm) \bigcup State$. The big-step transition relation $\Longrightarrow \subseteq (State \times Comm) \times State$ is defined as the smallest relation closed under the rules given in Table 22.4.

Command **skip** leaves the state unchanged. If an expression e evaluates to a value v in a state σ (given in terms of the big-step relation for expressions), the effect of an assignment $x{:=}e$ results in a state that is identical to σ, except that its value at variable x is now v. If c_1 transforms σ to σ_1 and c_2 transforms σ_1 to σ_2, then their sequential composition achieves the composite transformation of σ to σ_2. The rules for the conditional say that **if** e **then** c_1 **else** c_2 transforms σ as would command c_1 (c_2, respectively) depending on whether e evaluates to **true** or **false** in state σ. The *while* rules for the

TABLE 22.4 Big-Step Semantics for a Simple Imperative Language

skip	$\dfrac{}{\langle \sigma,\ \textbf{skip}\rangle \Longrightarrow \sigma}$
assign	$\dfrac{\sigma \vdash e \Longrightarrow_e v}{\langle \sigma,\ x{:=}e\rangle \Longrightarrow \sigma[x \mapsto v]}$
seq	$\dfrac{\langle \sigma,\ c_1\rangle \Longrightarrow \sigma_1 \quad \langle \sigma_1,\ c_2\rangle \Longrightarrow \sigma_2}{\langle \sigma,\ c_1;c_2\rangle \Longrightarrow \sigma_2}$
if_{true}	$\dfrac{\sigma \vdash e \Longrightarrow_e \textbf{true} \quad \langle \sigma,\ c_1\rangle \Longrightarrow \sigma_1}{\langle \sigma,\ \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2\rangle \Longrightarrow \sigma_1}$
if_{false}	$\dfrac{\sigma \vdash e \Longrightarrow_e \textbf{false} \quad \langle \sigma,\ c_2\rangle \Longrightarrow \sigma_2}{\langle \sigma,\ \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2\rangle \Longrightarrow \sigma_2}$
$while_{false}$	$\dfrac{\sigma \vdash e \Longrightarrow_e \textbf{false}}{\langle \sigma,\ \textbf{while } e \textbf{ do } c\rangle \Longrightarrow \sigma}$
$while_{true}$	$\dfrac{\sigma \vdash e \Longrightarrow_e \textbf{true} \quad \langle \sigma,\ c\rangle \Longrightarrow \sigma_1 \quad \langle \sigma_1,\ \textbf{while } e \textbf{ do } c\rangle \Longrightarrow \sigma_2}{\langle \sigma,\ \textbf{while } e \textbf{ do } c\rangle \Longrightarrow \sigma_2}$

indefinite iterator are also intuitive — if the boolean condition e evaluates to **false** in state σ, the loop is not entered; if e evaluates to **true** then if the body c of the loop is executed to reach state σ_1 and if executing the loop **while** e **do** c starting from σ_1 yields state σ_2, then σ_2 is the resulting state from executing the loop. Observe that this relational specification corresponds to partial correctness.[4]

The $\Longrightarrow$ relation is deterministic:

PROPOSITION 22.5. *If* $\langle\sigma,\ c\rangle \Longrightarrow \sigma_1$ *and* $\langle\sigma,\ c\rangle \Longrightarrow \sigma_2$ *then* $\sigma_1 = \sigma_2$.

There are some subtle technical issues about these rules that arise (e.g., in formal compiler verification exercises). As noted in [5], the rules for the **while** e **do** c yield an inductive definition, but one which is not structural. The two *while* rules can be coalesced into a single equivalent rule, which too is not structural:

$$\frac{\langle\sigma,\ \textbf{if}\ e\ \textbf{then}\ c;\textbf{while}\ e\ \textbf{do}\ c\ \textbf{else skip}\rangle \Longrightarrow \sigma'}{\langle\sigma,\ \textbf{while}\ e\ \textbf{do}\ c\rangle \Longrightarrow \sigma'}$$

Both these formulations involve a recursive definition, which while being concise and intuitive do not allow the use of structural induction. Fortunately, there is an equivalent formulation for the **while** e **do** c rule which is structural; this formulation employs an auxiliary inductively defined relation $\mathcal{F} \subseteq State \times State$:

$$\frac{\langle\sigma,\ \sigma'\rangle \in \mathcal{F}}{\langle\sigma,\ \textbf{while}\ e\ \textbf{do}\ c\rangle \Longrightarrow \sigma'}$$

where $\mathcal{F}$ is defined inductively by:

$$\frac{\sigma\ \vdash\ e \Longrightarrow_e \textbf{false}}{\langle\sigma,\ \sigma\rangle \in \mathcal{F}}$$

$$\frac{\sigma\ \vdash\ e\ \Longrightarrow_e\ \textbf{true}\quad \langle\sigma,\ c\rangle \Longrightarrow \sigma''\quad \langle\sigma'',\ \sigma'\rangle \in \mathcal{F}}{\langle\sigma,\ \sigma'\rangle \in \mathcal{F}}$$

We can now propose operational notions of equivalence and ordering between WHILE programs according to the following definition:

DEFINITION 22.4 (operational equivalence).
$c_1 \preceq c_2$ *whenever for all* σ*:* $\langle\sigma,\ c_1\rangle \Longrightarrow \sigma_1 \ \supset\ \langle\sigma,\ c_2\rangle \Longrightarrow \sigma_1$
$c_1 \approx c_2$ *whenever for all* σ*:* $\langle\sigma,\ c_1\rangle \Longrightarrow \sigma_1$ *if and only if* $\langle\sigma,\ c_2\rangle \Longrightarrow \sigma_1$

These notions are instances of the concepts of Definition 22.1 — the observable behavior of a command is how it transforms a given state to yield a resulting state.

Example 22.6
Here are some equivalences and ordering relations that can be seen as code improvements:

1. **skip** $\approx$ **while false do** c for all commands c.
2. **while true do** $c \preceq c'$ for all c, c' because the former is nonterminating.
3. Let $W \equiv$ **while** e **do** c. Then $W \approx$ **if** e **then** $c; W$ **else skip**.
4. c; **skip** $\approx c \approx$ **skip**; c for all c.
5. **if true then** c_1 **else** $c_2 \approx c_1$ and **if false then** c_1 **else** $c_2 \approx c_2$.

[4]In fact, it is possible to read the Hoare style axiomatic semantics for WHILE as a backward operational semantics on a nonstandard kind of state.

TABLE 22.5 Reduction Semantics for a Simple Imperative Language

$skip'$	$\dfrac{}{\langle \sigma,\ \mathbf{skip}\rangle \longrightarrow_1 \sigma}$
$assign'_1$	$\dfrac{\sigma \vdash e \longrightarrow_1^e e'}{\langle \sigma,\ x{:=}e\rangle \longrightarrow_1 \langle \sigma,\ x{:=}e'\rangle}$
$assign'_2$	$\dfrac{}{\langle \sigma,\ x{:=}v\rangle \longrightarrow_1 \sigma[x \mapsto v]}$
seq'_1	$\dfrac{\langle \sigma,\ c_1\rangle \longrightarrow_1 \langle \sigma_1,\ c'_1\rangle}{\langle \sigma,\ c_1;c_2\rangle \longrightarrow_1 \langle \sigma_1,\ c'_1;c_2\rangle}$
seq'_2	$\dfrac{\langle \sigma,\ c_1\rangle \longrightarrow_1 \sigma_1}{\langle \sigma,\ c_1;c_2\rangle \longrightarrow_1 \langle \sigma_1,\ c_2\rangle}$
if'_1	$\dfrac{\sigma \vdash e \longrightarrow_1^e e'}{\langle \sigma,\ \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\rangle \longrightarrow_1 \langle \sigma,\ \mathbf{if}\ e'\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\rangle}$
if'_{true}	$\dfrac{}{\langle \sigma,\ \mathbf{if\ true\ then}\ c_1\ \mathbf{else}\ c_2\rangle \longrightarrow_1 \langle \sigma,\ c_1\rangle}$
if'_{false}	$\dfrac{}{\langle \sigma,\ \mathbf{if\ false\ then}\ c_1\ \mathbf{else}\ c_2\rangle \longrightarrow_1 \langle \sigma,\ c_2\rangle}$
$while'$	$\dfrac{\langle \sigma,\ \mathbf{if}\ e\ \mathbf{then}\ c;\mathbf{while}\ e\ \mathbf{do}\ c\ \mathbf{else\ skip}\rangle \longrightarrow_1 \langle \sigma',\ c'\rangle}{\langle \sigma,\ \mathbf{while}\ e\ \mathbf{do}\ c\rangle \longrightarrow_1 \langle \sigma',\ c'\rangle}$

22.3.1.3 Reduction Semantics

We now move on to the reduction semantics for WHILE as a possibly more detailed description on how to realize an imperative language. The main difference now is the relation $\longrightarrow_1 \subseteq (\mathit{State} \times \mathit{Comm}) \times ((\mathit{State} \times \mathit{Comm}) \bigcup \mathit{State})$. The canonical (normal) forms for this relation are naturally those in *State*. Table 22.5 presents the reduction semantics.

The $\longrightarrow_1$ relation is easy to understand. Rule $skip'$ says **skip** does nothing. Executing an assignment first involves simplifying the expression e (repeatedly using rule $assign'_1$) down to a value v, which is then associated with x (rule $assign'_2$). Executing a sequential composition $c_1;c_2$ involves executing the first command c_1 until it is exhausted (repeatedly using rule seq'_1), at which stage we start the execution of c_2 from the resulting state σ_1 (rule seq'_2). In evaluating a conditional, we first evaluate the expression e to a Boolean value (repeatedly using rule if'_1). If that value is **true**, then c_1 is executed (rule if'_{true}) and if it is **false**, c_2 is executed (rule if'_{false}). The $while'$ rule is, again, somewhat harder to formalize concisely, and relies on the fact that **skip** is a NO-OP.

Proposition 22.6. *The big-step and reduction semantics define the same notion of program execution, that is, for all c and* σ*:* $\langle \sigma,\ c\rangle \Longrightarrow \sigma'$ *if and only if* $\langle \sigma,\ c\rangle\ (\longrightarrow_1)^* \sigma'$.

Remark 22.2. In fact, in [63], Plotkin uses what Astesiano calls a "mixed step" semantics for branching and iteration. For example, the rules for **while** he gives are:

$$\frac{\sigma \vdash e\ (\longrightarrow_1^e)^*\ \mathbf{true}}{\langle \sigma,\ \mathbf{while}\ e\ \mathbf{do}\ c\rangle \longrightarrow_1 \langle \sigma,\ c;\mathbf{while}\ e\ \mathbf{do}\ c\rangle} \qquad \frac{\sigma \vdash e\ (\longrightarrow_1^e)^*\ \mathbf{false}}{\langle \sigma,\ \mathbf{while}\ e\ \mathbf{do}\ c\rangle \longrightarrow_1 \sigma}$$

His small-step rules for the while command involve the transitive closure of the small-step reduction of expressions, equivalent to a big-step expression evaluation.

Recall that a small-step semantics can be thought of as moving "irrevocably forward" albeit nondeterministically, whereas big-step semantics can easily incorporate temporary undoable changes in describing subcomputations. Constructs that have a relatively simple big-step semantics but have difficult small-step semantics usually necessitate additional data structures such as stacks in the abstract machine for effecting the temporary changes involved in subcomputations.

22.3.1.4 Abstract Machine

The abstract machine for WHILE is a so-called stack-memory-code (SMC) machine, with a stack for evaluating expressions, a memory, or state, component and a code list. As illustrated by Plotkin [63, pp. 17–19], the transition semantics is somewhat messy: the transition relation is not directly in terms of syntactic structure. The linearization of this abstract syntax via a post-order traversal that worked well for expressions requires adjustments for constructs involving branching and iteration, where control points need to be stacked for further use or disposal. The reason is that execution of a program is no longer isomorphic to traversal of the abstract syntax tree, because a transition sequence can involve executing constructs in which an entire subtree may be ignored (branching and iteration), or may be revisited repeatedly (iteration).

22.3.2 Nondeterminism

Dijkstra's so-called *guarded choice language* is a quintessential imperative language involving nondeterminism:

$$c \ ::\ = \ \ldots \mid \mathbf{if}\ \square_{i=1}^{n}\ e_i \ \triangleright\ c_i\ \mathbf{fi} \mid \mathbf{do}\ \square_{i=1}^{n}\ e_i\ \triangleright\ c_i\ \mathbf{od}$$

Following are the big-step and mixed-step semantics for the new constructs. We use the mixed-step approach of Plotkin (or Astesiano) for reduction, because it yields a compact presentation (a pure small-step presentation is replete with the problems mentioned earlier). The big-step rules are:

$$\frac{\sigma \vdash e_j \Longrightarrow_e \mathbf{true} \quad \langle \sigma,\ c_j\rangle \Longrightarrow \sigma'}{\langle \sigma,\ \mathbf{if}\ \square_{i=1}^{n}\ e_i\ \triangleright\ c_i\ \mathbf{fi}\rangle \Longrightarrow \sigma'} \qquad (j \in \{1, \ldots, n\}$$

$$\frac{\sigma \vdash e_j \Longrightarrow_e \mathbf{true} \quad \langle \sigma,\ c_j;\mathbf{do}\ \square_{i=1}^{n}\ e_i\ \triangleright\ c_i\ \mathbf{od}\rangle \Longrightarrow \sigma'}{\langle \sigma,\ \mathbf{do}\ \square_{i=1}^{n}\ e_i\ \triangleright\ c_i\ \mathbf{od}\rangle \Longrightarrow \sigma'} \qquad (j \in \{1, \ldots, n\}$$

$$\frac{\wedge_{i=1}^{n} \sigma \vdash e_i \Longrightarrow_e \mathbf{false}}{\langle \sigma,\ \mathbf{do}\ \square_{i=1}^{n}\ e_i\ \triangleright\ c_i\ \mathbf{od}\rangle \Longrightarrow \sigma}$$

and in the mixed-step formulation:

$$\frac{\sigma \vdash e_j\ (\longrightarrow_1^e)^*\ \mathbf{true}}{\langle \sigma,\ \mathbf{if}\ \square_{i=1}^{n}\ e_i\ \triangleright\ c_i\ \mathbf{fi}\rangle \longrightarrow_1 \langle \sigma,\ c_j\rangle} \qquad (j \in \{1, \ldots, n\}$$

$$\frac{\sigma \vdash e_j\ (\longrightarrow_1^e)^*\ \mathbf{true}}{\langle \sigma,\ \mathbf{do}\ \square_{i=1}^{n}\ e_i\ \triangleright\ c_i\ \mathbf{od}\rangle \longrightarrow_1 \langle \sigma,\ c_j;\mathbf{do}\ \square_{i=1}^{n}\ e_i\ \triangleright\ c_i\ \mathbf{od}\rangle} \qquad (j \in \{1, \ldots, n\}$$

$$\frac{\wedge_{i=1}^{n} \sigma \vdash e_i\ (\longrightarrow_1^e)^*\ \mathbf{false}}{\langle \sigma,\ \mathbf{do}\ \square_{i=1}^{n}\ e_i\ \triangleright\ c_i\ \mathbf{od}\rangle \longrightarrow_1 \sigma}$$

In each set, the first two rules are really families of rules (one for each choice of j). The rule for guarded choice says that if any e_i evaluates to true in σ, then the corresponding c_i may be executed from σ. The rules for guarded iteration say that if any e_i evaluates to true in σ, then the corresponding c_i followed by the loop again may be executed from σ, whereas if all e_i evaluate to false, control exits the iteration construct.

An implementation (or abstract machine) has to use some mechanism for evaluating the guard expressions down to values choosing some order. To achieve some degree of fairness, a scheduler may be used to select the order in which guard expressions can be tried.

22.3.2.1 Parallel Execution

Many concurrent imperative languages allow parallel execution of threads, for instance in a **cobegin-coend** construct:

$$c \ ::= \ \ldots \mid c_1 \| c_2$$

Consider the following big-step semantics:

$$\frac{\langle \sigma,\ c_1 \rangle \Longrightarrow \sigma_1 \quad \langle \sigma,\ c_2 \rangle \Longrightarrow \sigma_2}{\langle \sigma,\ c_1 \| c_2 \rangle \Longrightarrow \sigma_1 \cup \sigma_2} \quad \text{provided } \ W(c_1) \cap W(c_2) = \emptyset$$

where $W(c_i)$ denotes the set of variables changed in c_i. The proviso ensures that the union is well-defined. Unfortunately, this rule does not correspond to our usual intuition of parallel computation. It is correct only when both threads do not use the contents of variables modified by the other (Bernstein's conditions); otherwise this specification is difficult to implement.

The small-step semantics are simpler to implement (and less fussy to specify):

$$\frac{\langle \sigma,\ c_i \rangle \longrightarrow_1 \langle \sigma',\ c_i' \rangle}{\langle \sigma,\ c_1 \| c_2 \rangle \longrightarrow_1 \langle \sigma',\ c_1' \| c_2' \rangle} \quad i \in \{1, 2\}$$

$$\frac{\langle \sigma,\ c_i \rangle \longrightarrow_1 \sigma'}{\langle \sigma,\ c_1 \| c_2 \rangle \longrightarrow_1 \langle \sigma',\ c_{3-i} \rangle} \quad i \in \{1, 2\}$$

What this suggests is that the granularity of abstraction that the big-step semantics seeks to impose in describing the operational behavior is inappropriate for concurrent computation. Also, using big-step semantics makes it difficult to describe visible side effects of a computation during non-terminating runs. Consequently, it is common to find most frameworks for concurrency (e.g., [50]) using [generally labeled] reduction semantics.

22.3.3 Blocks and Variable Declarations

Most-imperative languages are block structured, and employ scoped declarations of variables, which are (often) initialized before any command is executed. Moreover, we have not studied any constructs where imperative variables (which are really *named storage cells*) can have any structure. A more general treatment of imperative variables is to factor the notion of *State* into two maps: the first is an environment $\gamma \in \mathit{Env} = \mathcal{X} \rightarrow_{\mathit{fin}} \mathit{Loc}$; and the second a store $\sigma \in \mathit{Store} = \mathit{Loc} \rightarrow_{\mathit{fin}} \mathcal{V}$, where *Loc* is a set of storage addresses, or locations, and $\mathcal{V}$ is the set of (storable) values. Environments can also be used to model constant declarations by including $\mathcal{V}$ in the codomain of *Env*. The common practice is to have different environment components for constants, variables, procedures, types, classes and modules — whatever distinct nameable concepts appear in the language. In what follows, we assume that the appropriate environment component is looked up.

We ignore the issue of the types of the declared variables, because they are usually irrelevant for specifying the dynamic behavior. Consequently, variable declarations merely become lists of variables:

$$c \ ::= \ \ldots \mid \textbf{var}\ vd\ \textbf{begin}\ c\ \textbf{end} \qquad vd ::= x \mid vd; vd$$

The big-step relations now are $\Longrightarrow_e \subseteq ((\mathit{Env} \times \mathit{Store}) \times \mathit{Exp}) \times \mathcal{V}$ for expressions, $\Longrightarrow_c \subseteq \mathit{Env} \times (\mathit{Store} \times \mathit{Comm}) \times \mathit{Store}$ for commands, $\Longrightarrow_d \subseteq (\mathit{Env} \times \mathit{Store} \times \mathit{Decl}) \times (\mathit{Env} \times \mathit{Store})$ for declarations. (This is a little more general than we need for the moment, but will allow variable initializations during declarations, and can be invaluable in the specification of procedures).

The rules for expressions are now relative to a pair γ, σ and except for the variable lookup all other rules are otherwise unchanged. All the previous rules given for commands are now made relative to an environment γ, and $\Longrightarrow$ is now subscripted $\Longrightarrow_c$. We now give the new and changed rules for variable lookup, variable declarations, assignments and blocks (only for the big-step case — we encounter the same issues in blocks as we did in local declarations when attempting a small-step formalization):

$$\frac{}{\gamma, \sigma \vdash x \Longrightarrow_e v} \quad \text{where } v = \sigma(\gamma(x)), \text{ if defined}$$

$$\frac{\gamma, \sigma \vdash e \Longrightarrow_e v}{\gamma \vdash \langle \sigma,\ x{:=}e \rangle \Longrightarrow_c \sigma[\gamma(x) \mapsto v]} \quad \text{provided } x \in \mathit{dom}(\gamma)$$

$$\frac{}{\langle \gamma,\ \sigma,\ x \rangle \Longrightarrow_d \langle [x \mapsto l],\ \sigma[l \mapsto \bot] \rangle} \quad \text{where } l \notin \mathit{codom}(\gamma) \bigcup \mathit{dom}(\sigma)$$

$$\frac{\langle \gamma,\ \sigma,\ vd_1 \rangle \Longrightarrow_d \langle \gamma_1,\ \sigma_1 \rangle \quad \langle \gamma[\gamma_1],\ \sigma_1,\ vd_2 \rangle \Longrightarrow_d \langle \gamma_2,\ \sigma_2 \rangle}{\langle \gamma,\ \sigma,\ vd_1;vd_2 \rangle \Longrightarrow_d \langle \gamma_1[\gamma_2],\ \sigma_2 \rangle}$$

$$\frac{\langle \gamma,\ \sigma,\ vd \rangle \Longrightarrow_d \langle \gamma_1,\ \sigma_1 \rangle \quad \gamma[\gamma_1] \vdash \langle \sigma_1,\ c \rangle \Longrightarrow_c \sigma'}{\gamma \vdash \langle \sigma,\ \textbf{var}\ vd\ \textbf{begin}\ c\ \textbf{end} \rangle \Longrightarrow_c \sigma' \wr \mathit{dom}(\sigma)}$$

In assignments, we now use γ to determine the location corresponding to x, which is updated in the store. In variable declarations, fresh locations are generated and added to the store (initialized to an undefined value $\bot$), then bound to the variables in the environment. Observe that we have (somewhat idiosyncratically) the environments returned be increment (and hence undoable), whereas the changes to the store be cumulative (i.e., persistent). This approach is appropriate for small or mixed step semantics, and also for any extensions to procedures. At the abstract machine level, this hints that environments must necessarily be implemented using stacks whereas stores can be global, with careful control on accessibility of locations.

Also, the returned state in the execution of a block purges all components of the store that were created during execution of the block. This is to avoid the occurrence of locations inaccessible from the environment (i.e., garbage). Likewise, we have been careful to avoid the possibility of dangling references, namely, locations accessible from the environment but not present in the domain of the store — which can occur if we have a command **free**(x).

22.3.4 Procedures and Parameter Passing

We now introduce the possibility of declaring and calling procedures in the WHILE language. We consider only nonrecursive procedures, with a single variable. The extension to several variables and indeed to several variables with several different parameter-passing mechanisms is at least intuitively straightforward (though rather tedious to write as rules). The extension to recursive procedures, however, is not quite trivial. It involves the computation of fixed points by an iterative process akin to the case of the **while** command.

Indeed, we have met some of the scoping issues during our treatment of blocks (via Tennent's principle of correspondence, where any parameter passing mechanism corresponds to a definition mechanism and conversely). The issues of managing control during call and return are better treated in a more general setting of first-class abstractions in Section 22.4.

22.3.4.1 Parameterless Procedures

We first extend the language with facilities for declaring and calling procedures without parameters. It is then easy to extend this further to various parameter-passing conventions such as *call-by-value* and *call-by-reference*:

$$d \ ::= \ \dots \mid \textbf{sub}\ P \ = \ c$$

$$c \ ::= \ \dots \mid \textbf{call}\ P$$

As in most programming languages, we assume that the body c of the procedure P may refer to and modify nonlocal (free) variables that are visible by the usual rules of static scope.

Semantically a (parameterless) procedure is merely a state transformer with a name. Hence, it is necessary to include state transformers in the codomain of semantic environments:

$$\mathit{Proc}_0 \ = \ \mathit{Store} \rightarrow_p \mathit{Store}$$

$$\mathit{Env} \ = \ \mathcal{X} \rightarrow_{\mathit{fin}} (\mathit{Loc} + \mathit{Proc}_0 + \cdots)$$

Operationally, however, a procedure identifier merely represents sufficient information required to be able to execute the code of the procedure. Lexical scoping requires that variables in the body of the procedure take their bindings in the environment that the procedure was declared, instead of from the calling context. Hence, a procedure declaration modifies the environment by associating with the procedure name, the environment in which the declaration occurs and the body of the procedure. Such a data structure is called a *procedural closure*. We revisit closures in Section 22.4 while discussing function call in lexically scoped functional languages. As in the case of blocks and declarations, we assume that the state has two components, an environment γ, and a store σ:

$$\mathit{Sub}_0 \frac{}{\langle \gamma,\ \sigma,\ \textbf{sub}\ P \ = \ c\rangle \Longrightarrow_d \langle \gamma[P \mapsto \mathtt{proc0}\langle c,\ \gamma\rangle],\ \sigma\rangle}$$

$$\mathit{Call}_0 \frac{\gamma_1 \vdash \langle \sigma, c\rangle \Longrightarrow_c \sigma'}{\gamma \vdash \langle \sigma, \textbf{call}\ P\rangle \Longrightarrow_c \sigma'} \quad \gamma(P) = \mathtt{proc0}\langle c,\ \gamma_1\rangle$$

22.3.4.2 Procedures with Parameters

Extending the treatment to procedures with parameters, we consider, for simplicity, only procedures with a single parameter. We also consider only the call-by-value and call-by reference mechanisms. The extended language syntax is:

$$d \ ::= \ \dots \mid \textbf{sub}\ P\ (\textbf{val}\ x) = \ c \mid \textbf{sub}\ P\ (\textbf{var}\ x) = \ c$$

$$c \ ::= \ \dots \mid \textbf{call}\ P(e)$$

We require that the expression e can only be a variable symbol when the procedure P uses a **var** parameter. The mathematical domains for procedures of these kinds are:

$$\mathit{Proc}_v \ = \ (\mathit{Store} \times \mathcal{V}) \rightarrow_p \mathit{Store}$$

$$\mathit{Proc}_r \ = \ (\mathit{Store} \times \mathit{Loc}) \rightarrow_p \mathit{Store}$$

$$\mathit{Proc} \ = \ \mathit{Proc}_0 + \mathit{Proc}_v + \mathit{Proc}_r$$

$$\mathit{Env} \ = \ \mathcal{X} \rightarrow_{\mathit{fin}} (\mathit{Loc} + \mathit{Proc})$$

The corresponding closures used in the operational world now carry the formal parameters (marked with the name of the parameter-passing mechanism) in addition to the body of the procedure and its definition environment.

The operational rules for the new constructs are given as follows. In the rule $Call_v$, γ_1 is the environment of the declaration of the procedure P. It is necessary to allocate a new location l for the formal parameter x in which the value of the actual parameter obtained by evaluating e in the state $\langle\gamma,\ \sigma\rangle$ is stored. Finally, of course, the location l needs to be made inaccessible on exit from the procedure. Hence, the conclusion of the rule has the restriction of σ' to the domain of σ. The presence of the binding for P in γ_2 is a simple expedient to deal with recursion:

$$Sub_v \quad \frac{}{\langle\gamma,\ \sigma,\ \textbf{sub}\ P\ (\textbf{val}\ x)\ =\ c\rangle \Longrightarrow_d \langle[P \mapsto \texttt{proc}\langle\textbf{val}\ x, c, \gamma\rangle],\ \sigma\rangle}$$

$$Call_v \quad \frac{\gamma, \sigma\ \vdash\ e\ \Longrightarrow_e\ v \qquad \gamma_2 \vdash \langle\sigma[l \mapsto v], c\rangle \Longrightarrow_c \sigma'}{\gamma\ \vdash \langle\sigma,\ \textbf{call}\ P(e)\rangle \Longrightarrow_c \sigma' \wr dom(\sigma)}$$

where $\gamma_2 = \gamma_1[P \mapsto \gamma(P)][x \mapsto l],\quad l \notin codom(\gamma) \cup dom(\sigma)$ and $\gamma(P) = \texttt{proc}\langle\textbf{val}\ x, c, \gamma_1\rangle$.

In a similar vein we also define the semantics of procedures that use a reference parameter. Note that the formal parameter x is now associated with the location of the actual parameter y in invocation **call** $P(y)$. There are no new locations created, hence $dom(\sigma) = dom(\sigma')$. The effect of updating the formal parameter x within the procedure body is directly reflected in the contents of the location of the actual parameter:

$$Sub_r \quad \frac{}{\langle\gamma,\ \sigma,\ \textbf{sub}\ P\ (\textbf{var}\ x)\ =\ c\rangle \Longrightarrow_d \langle[P \mapsto \texttt{proc}\langle\textbf{var}\ x, c, \gamma\rangle],\ \sigma\rangle}$$

$$Call_r \quad \frac{\gamma_2 \vdash \langle\sigma, c\rangle \Longrightarrow_c \sigma'}{\gamma \vdash\ \langle\sigma, \textbf{call}\ P(y)\rangle \Longrightarrow_c \sigma'}$$

where $\gamma_2 = \gamma_1[P \mapsto \gamma(P)][x \mapsto \gamma(y)]$ and $\gamma(P) = \texttt{proc}\langle\textbf{var}\ x, c, \gamma_1\rangle$.

22.3.5 Runtime Allocation and Deallocation

One of the most nettlesome features of most programming languages is the use of pointers — their creation, access and disposal. Pointers are a major source of problems for users, implementors and language designers alike. It is therefore necessary to precisely define the semantics of dynamic memory allocation and deallocation. This is also a feature easier to treat operationally than denotationally. For simplicity, we consider pointers in isolation from block structure and procedures.

Briefly, one of the first problems with pointers is aliasing. The problem of aliasing is not an exceptional circumstance, as it is often the case that distinct dereferencing expressions refer to the same location on the heap. Hence, an assignment to one of the references might alter the value of some other seemingly unrelated expression. The second major problem is that it is fairly common to work with several logically distinct data structures in heap, where sharing of components occurs. Third, while discussing memory allocation and deallocation, it is important to treat definedness (a major source of runtime errors).

Calcagno, Ishtiaq and O'Hearn [16] have built on some previous work of Morris [**?**][10] to specify the semantics of aliasinallocation and disposal. For simplicity, the store is assumed to consist of two components — a stack, which holds the values of local variables, and a heap, which contains data that are dynamically created and destroyed. Naturally any access to the heap is from the stack. Any structure that is inaccessible from the stack is treated as garbage. The stack can be extended by declarations of local variables. Variable values can be modified by assignments. The heap, on the other hand, is assumed to consist of only one kind of data structure, namely, records, where each record has a fixed number of components indexed by tags.

We extend the language of expressions to include record component access and update. The metavariables $a, b, \dots$ range over tags (record components). An expression can also be a null pointer or access to a record component:

$$e \ ::= \ \dots | e.a$$

Correspondingly, the domain of denotable values of Ada for expressions is extended to) include locations and a special value *null*. The changes that are needed in the various domain definitions are listed below:

$$Tag = \{a, b, \cdots\}$$

$$\mathcal{V} = \cdots + Loc + \{null\}$$

$$Stack = \mathcal{X} \rightarrow_{fin} \mathcal{V}$$

$$Heap = Loc \rightarrow_{fin} (Tags \rightarrow \mathcal{V})$$

$$Store = Stack \times Heap$$

A store σ is a pair (st, hp), containing a stack st and a heap hp. Both the stack st and the heap hp are partial functions. Their domains are denoted $dom(st)$ and $dom(hp)$, respectively. The domain $dom(st)$ includes only the variables in the current scope and $dom(hp)$ includes only the locations allocated so far and is finite.

Two distinct variables x and y could point to the same record on the heap (i.e., $x.a$ and $y.a$ could be aliases). However, two distinct variables cannot be aliases because variables are on the stack and not on the heap. Moreover the "l-values" of variables cannot be modified. For any variable x that may be a pointer to a record on the heap, $x.a$ represents access to a component a.

Example 22.7
We consider linked lists with the two constructors — hd and tl. For any list variable x (on stack) $x.hd$ dehis notes the value of the first element in the list (if the list is nonempty), whereas $x.tl$ denotes a location from which the rest of the liste. Hence, $x.tl.hd$ is the value of the second element of the list (if a second element exists). We also allow for a special value $null$ to be stored in x, to denote the empty list. Hence, if the list (written ML style) `[1, 2, 3]` is the value of the variable x on stack, then we require $x \in dom(st)$ and three locations $\{l_1, l_2, l_3\} \subseteq dom(hp)$ such that:

$$st(x) \ = \ l_1$$

$$hp(l_1)(hd) \ = \ 1, \qquad hp(l_1)(tl) \ = \ l_2$$

$$hp(l_2)(hd) \ = \ 2, \qquad hp(l_2)(tl) \ = \ l_3$$

$$hp(l_3)(hd) \ = \ 3, \qquad hp(l_3)(tl) \ = \ null$$

Clearly, it follows that $x.tl.tl.hd = 3$ where "." is left associative.

The operational rule for the new expression is given next. The rules for other expressions are as given in Table 22.4. We use the metavariable l to range over $Loc + \{null\}$, and v ranges over actual values that are not locations:

$$ref_{loc} \quad \frac{(st, hp) \ \vdash \ e \ \Longrightarrow_e \ l \qquad l \in dom(hp)}{(st, hp) \ \vdash \ e.a \ \Longrightarrow_e hp(l)(a)}$$

Because it is now possible for assignment commands to allow the assignments of pointer expressions, we require two rules for the assignment command. The first rule defines the assignment of values to variables on the stack. Depending on the type of the variable, it may be either an integer value or a rather than[5] We use the metavariable vl to denote that it may be drawn from either values or $Loc + \{null\}$.

We use $h[l.a \mapsto v]$ to abbreviate $h[l \mapsto h(l)[a \mapsto v]]$. The rules for assignment are shown as follows:

$$assign_{var}\ \frac{(st, hp)\ \vdash\ e\ \Longrightarrow_e\ vl}{\langle (st, hp),\ \ x := e\ \rangle \Longrightarrow_c (st[x \mapsto vl], hp)}$$

$$assign_{ref}\ \frac{(st, hp)\ \vdash\ e_1\ \Longrightarrow_e\ l \quad (st, hp)\ \vdash\ e_2\ \Longrightarrow_e\ vl \qquad l \in dom(hp)}{\langle (st, hp),\ \ e_1.a := e_2\ \rangle \Longrightarrow_c (st, hp[l.a \mapsto vl])}$$

Having defined the semantics of references, we are now ready to augment the language with commands for allocation and deallocation of memory. We then extend the language of commands to include the two Pascal-like commands:

$$c\ ::=\ \ldots \mid \mathbf{new}(x) \mid \mathbf{free}(e)$$

Command **new**(x) nondeterministically selects a location not currently in $dom(hp)$ and initializes the record with the value "$\perp$". We use $hp[l.* \mapsto \perp *]$ to denote that all components of the record $hp(l)$ are initialized to $\perp$. Similarly, **free**(e) simply removes the location denoted by e from $dom(h)$. The rules given include:

$$new\ \frac{l \notin dom(hp)}{\langle (st, hp),\ \mathbf{new}(x)\rangle\ \Longrightarrow (st[x \mapsto l], hp[l.* \mapsto \perp *])}$$

$$free\ \frac{(st, hp)\ \vdash\ e \Longrightarrow_e l \quad l \in dom(hp)}{\langle (st, hp),\ \mathbf{free}(e)\rangle\ \Longrightarrow (st, hp \backslash l)}$$

In the rule for **free**(e), $h\backslash l$ denotes the fact that the heap is no longer defined for l (as opposed to being filled with value "$\perp$"). The preceding rules give us a flavor of how operational rules may be used to specify implementation intuition to a large extent. In [16], the authors also show how these rules may be used to justify axiomatic rules for reasoning locally about aliasing and dynamic memory allocation and deallocation.

22.4 Functions and Higher Order Forms

Applying the principle of abstraction [69] to expressions or commands allows us to form abstracts that may be invoked, usually with different parameters. These abstract forms are called *functions* and *procedures*, respectively. Abstract expressions (with a single parameter) are written as $\lambda x.e$. λ *binds* the variable x within the scope of the "body" e. An abstract a can be invoked by applying it to an argument e, written as $(a\ e)$; such calls belong to the syntactic category over which the abstract is formed.

The situation becomes more interesting in higher order languages, which admit such abstracts as first-class values — abstracts can themselves be bound to variables, passed as arguments and returned as results of other functions. The various issues related to functions and procedures, in particular, the correct formulation of lexical scoping and of recursive function definitions, can be explored in

[5]The issue of types is something that needs to be addressed by a static semantics, as pointed out elsewhere. It is not properly the concern of a dynamic semantics. Thus, we continue to believe that all the constructs we use are type safe.

a higher ordrathertthan ional language with only single paraltr funcisions. The generalizations to procedures and multiple parameters is a matter of detailing, but does not need very much by way of new concepts. Indeed, these two issues lexical scoping and recursion are of vital importance — early implementations of Lisp implemented dynamic scoping because of a rather simplistic implementation of recursion.

Exp is now extended to:

$$e \ ::= \ \ldots \lambda x.e \mid (e_1 \ e_2)$$

We look at an extremely simple quintessential functional language, called the λ-calculus. Indeed, Landin explicated the block structured features of Algol by relating them to the λ-calculus. The operational semantics for various flavors of the λ-calculus are given in a purely syntactic manner (involving no extra-syntactic constructs such as environments). From these, various environment-based formulations can be constructed to realize the semantics in an efficient manner.

22.4.1 λ-Calculus

The syntax of the pure λ-calculus is:

$$e \ ::= \ x \mid \lambda x.e_1 \mid (e_1 \ e_2)$$

Expressions (or terms) are variables, abstractions on expressions, or applications of one expression (putatively a function) to another (an positivent). Other kinds of values and expressions such as those we have examined so far can be added together with their computation rules to obtain an applied λ-calculas. Although applied λ-calculi raise interesting issues and problems, the pure calculus itself exhibits several important concepts. Plotkin's seminal papers [61, 62] are good examples of detailed studies of many of the fundamental issues.

Definition 22.5 (free and bound variables). *An occurrence of a variable x in a term e is bound if it appears in a subterm $\lambda x.e'$. All occurrences of variables that are not bound or binding are free. The function fv returns the set of free variables in a term:*

$$fv(x) = \{x\} \quad fv(\lambda x.e) = fv(e) - \{x\} \quad fv((e_1 \ e_2)) = fv(e_1) \cup fv(e_2)$$

Bound variables may be systematically renamed without altering the intended meaning of an expression. By systematic, we mean that two hitherto different variables are not suddenly identified, in particular, no previously free variable is suddenly "captured" and bound. We identify expressions that differ only in the choice of names of bound variables, a notion called α-equivalence. Expressions with no free variables are called closed.

The major metaoperation for syntactic manipulation in any λ-calculus is substitution.

Definition 22.6 (substitution). *We write $e[e'/x]$ to denote the term obtained by substituting e' for all free occurrences of variable x in term e. Substitution is defined using a case analysis on e:*[6]

$$x[e'/x] = e'$$

$$y[e'/x] = y \qquad y \equiv x$$

$$(e_1 \ e_2)[e'/x] = (e_1[e'/x] \ e_2[e'/x])$$

$$\lambda y.e_1[e'/x] = \lambda z.(e_1[z/y][e'/x]) \quad z \notin fv(e_1) \cup fv(e')$$

[6]This version of the definition factors in α-equivalence whereas most treatments do not.

TABLE 22.6 β-Reduction in the λ-Calculus

(β)	$\dfrac{}{(\lambda x.e_1\ e_2) \longrightarrow_1 e_1[e_2/x]}$
(ξ)	$\dfrac{e \longrightarrow_1 e'}{\lambda x.e \longrightarrow_1 \lambda x.e'}$
(op)	$\dfrac{e_1 \longrightarrow_1 e_1'}{(e_1\ e_2) \longrightarrow_1 (e_1'\ e_2)}$
(arg)	$\dfrac{e_2 \longrightarrow_1 e_2'}{(e_1\ e_2) \longrightarrow_1 (e_1\ e_2')}$

Because substitution avoids capture of free names, it perforce avoids the possibility of accidental dynamic binding.

It is often convenient to use the notion of contexts in examining the structure of a term.

Definition 22.7 (context). *A context is a λ-term with a "hole" given bthe y the followingtrathe ct of expressions grammar:*

$$C\ ::=\ [\,]\mid(C\ C)\mid\lambda x.C\mid e$$

One-hole contexts athe re characterized as:

$$C^1\ ::=\ [\,]\mid(C^1\ e)\mid(e\ C^1)\mid\lambda x.C^1$$

Definition 22.8 (reduction). *A redex is any term of the form $(\lambda x.e_1\ e_2)$. Any term containing a redex as a subterm is called reducible. The β-reduction rule is:*

$$C^1[((\lambda x.e_1)\ e_2)]\ \longrightarrow_1\ C^1[e_1[e_2/x]]$$

where $C^1[\,]$ is any one-hole context.

An alternative formulation of β-reduction is given in Table 22.6. Some important results about β-reduction are mentioned next.

Lemma 22.1 (substitution and β-reduction). *If $e \longrightarrow_1 e'$ then $e_1[e/x]\ (\longrightarrow_1)^*\ e_1[e'/x]$ and $e[e_1/x] \longrightarrow_1 e'[e_1/x]$.*

Proposition 22.7 (local confluence). *β-reduction satisfies the weak diamond property.*

Theorem 22.1 (Church–Rosser). *β-reduction is confluent.*

Theorem 22.2 (standardization). *If $e(\longrightarrow_1)^*e'$ then $e(\longrightarrow_1^{standard})^*e'$ where standard is the reduction relation in which the leftmost outermost redux is reduced.*

Proposition 22.8 (fixed points). *There exists a closed λ-calculus term Y, called a fixed point combinator, such that $(Y\ e)\ \longrightarrow_1^*\ (e\ (Y\ e))$, for any e.*

TABLE 22.7 Call-by-Value β-Reduction

(β_v)	$\dfrac{}{(\lambda x.e_1\ v) \longrightarrow_1^v e_1[v/x]}$
(op)	$\dfrac{e_1 \longrightarrow_1^v e_1'}{(e_1\ e_2) \longrightarrow_1^v (e_1'\ e_2)}$
(arg_v)	$\dfrac{e_2 \longrightarrow_1^v e_2'}{(v\ e_2) \longrightarrow_1^v (v\ e_2')}$

22.4.2 Relationship with Functional Languages

Almost all functional languages disallow reduction "below" a λ — Redexes appearing in terms of the form $\lambda x.e$ are not considered. In other words, such weak reduction does not have the ξ-rule. Hence, not all results (confluence!) shown for the λ-calculus automatically transfer to functional languages based on them. Moreover, certain results do not hold for typed frameworks. For instance, fixed point combinators do not exist in simply typed λ-calculi.[7] Finally, we should mention that programming languages are concerned with closed terms only.

Two commonly used strategies for reducing terms are (weak) call-by-value (or eager) and (weak) call-by-name (or lazy).[8] In what follows, we present different formulations of these two strategies and how they are realized.

22.4.2.1 Call-by-Value

The basic notion in call-by-value (*cbv*) is that arguments to a function are evaluated before evaluation of the function body commences. The notion of value is crucial — they include all abstractions: $v \in Val ::= \lambda x.e$. Values are irreducible, but not conversely.

We first present the big-step formulation for call-by-value evaluation:

$$\frac{}{v \Longrightarrow_v v} \qquad \frac{e_1 \Longrightarrow_v \lambda x.e_1' \quad e_2 \Longrightarrow_v v_2 \quad e_1'[v_2/x] \Longrightarrow_v v}{(e_1\ e_2) \Longrightarrow_v v}$$

In the small-step framework, this notion is formulated as shown in Table 22.7.

Alternatively, the *cbv* strategy can be explained by using the β_v reduction rule in the following *cbv* evaluation contexts:

$$E_v^1 ::= [\,] \mid (E_v^1\ e) \mid (v\ E_v^1)$$

22.4.2.2 Call-by-Name

Call-by-name (*cbn*), in contrast, does not simplify arguments before function call. The big-step *cbn* rules are:

$$\frac{}{v \Longrightarrow_n v} \qquad \frac{e_1 \Longrightarrow_n \lambda x.e_1' \quad e_1'[e_2/x] \Longrightarrow_n v}{(e_1\ e_2) \Longrightarrow_n v}$$

Note that arguments are not evaluated before substituting them for the formal parameter in the function body. This may result in an expression being evaluated multiple times — each copy of

[7]They can exist in languages with reflexive types or recursive types.

[8]Various researchers actually distinguish between call-by-value and eager (or call-by-name and lazy), which we gloss over here.

TABLE 22.8 Call-by-Name β-Reduction

(β)	$\dfrac{}{(\lambda x.e_1\ e_2)\ \longrightarrow_1^n\ e_1[e_2/x]}$
(op)	$\dfrac{e_1\ \longrightarrow_1^n\ e_1'}{(e_1\ e_2)\ \longrightarrow_1^n\ (e_1'\ e_2)}$

an argument is evaluated separately. The advantage of *cbn* over *cbv* is that arguments that are not needed are not evaluated. An important static analysis technique is strictness analysis, in which *cbv* evaluation can safely be used instead of *cbn*. The small-step formulation of the *cbn* rules is given in Table 22.8.

Alternatively, the *cbn* strategy can be explained by using the β rule in the following *cbn* evaluation contexts:

$$E_n^1\ ::=\ [\,]\ |\ (E_n^1\ e)$$

22.4.2.3 Context Machines

The notion of evaluation context permits a simple transformation (due to Felleisen and Wright [75]) of reduction semantics into an abstract machine. We illustrate the idea for *cbv* reduction. A similar machine can be constructed for *cbn* reduction.

We first characterize basic evaluation contexts F^v:

$$F^v\ ::=\ ([\,]\ e)\ |\ (\lambda x.e\ [\,])$$

By using the fact that any nontrivial *cbv* evaluation context can be expressed as the composition of basic evaluation contexts $F_1^v[F_2^v[\ldots F_k^v[\ ]\ldots]]$, (the trivial context [ ] can be considered as corresponding to the case where $k = 0$), we define a "context stack machine" as follows. The machine has two components — a stack of basic evaluation contexts *FS*, and the current expression e. Transitions are defined by cases depending on the structure of e and then of *FS*:

$$\left\langle \left\lfloor \begin{matrix} ([\,]\ e) \\ FS \end{matrix} \right\rfloor,\ v \right\rangle \longrightarrow\!\!\triangleright \left\langle \left\lfloor \begin{matrix} (v\ [\,]) \\ FS \end{matrix} \right\rfloor,\ e \right\rangle$$

$$\left\langle \left\lfloor \begin{matrix} ((\lambda x.e)\ [\,]) \\ FS \end{matrix} \right\rfloor,\ v \right\rangle \longrightarrow\!\!\triangleright \langle \lfloor FS \rfloor,\ e[v/x] \rangle$$

$$\langle \lfloor FS \rfloor, (e_1\ e_2) \rangle \longrightarrow\!\!\triangleright \left\langle \left\lfloor \begin{matrix} ([\,]\ e_2) \\ FS \end{matrix} \right\rfloor,\ e_1 \right\rangle$$

The machine is started in configuration $\langle \lfloor\quad\rfloor,\ e \rangle$ for any closed e and terminates with context stack empty and value v.

Now if we define function *crunch* as:

$$crunch\langle \lfloor\quad\rfloor,\ e \rangle\ =\ e$$

$$crunch\left\langle \left\lfloor \begin{matrix} F_n^v \\ FS \end{matrix} \right\rfloor,\ e \right\rangle\ =\ crunch\langle \lfloor FS \rfloor,\ F_n^v[e] \rangle$$

it is easy to show that:

$$\langle \lfloor FS \rfloor,\ e \rangle \longrightarrow\!\!\triangleright^*\ \langle \lfloor\quad\rfloor,\ v \rangle \text{ if and only if } crunch\langle \lfloor FS \rfloor,\ e \rangle\ \Longrightarrow_v\ v$$

22.4.3 Closures and Environment Machines

As mentioned earlier in passing, substitution is an expensive operation, because it involves traversing the term in which the substitution is performed (as well as α-conversion to prevent name capture). Environments are a convenient ancillary structure used to record the bindings for variables in substitutions.

22.4.3.1 Closures

Suppose environments were, as before, represented by finite domain functions from variables to values. Suppose we considered an environment γ in which f was bound to $\lambda x.e$ and proposed a rule for function call:

$$\frac{\gamma \vdash e_1 \Longrightarrow_e v_1 \quad \gamma[x \mapsto v_1] \vdash e \Longrightarrow_e v}{\gamma \vdash f(e_1) \Longrightarrow_e v}$$

The problem with this rule is that if e contains free variables other than x, lexical scoping may be violated if the binding for f was made in an environment other than γ, because in the call, these free variables would take their value (if they could) from γ. Although the problem is more acute in higher order languages, it nevertheless exists in simple block structured procedures as well, which is why we disallowed nested procedures in Section 22.3.3. It is therefore necessary to "pack in the prevalent environment" when making the binding for f. Such a pair is called a *closure*. We define:

$$\mathit{Clos} \subseteq \mathit{Exp} \times \mathit{Env} \qquad \mathit{Env} = \mathcal{X} \rightarrow_{\mathit{fin}} \mathit{Clos}$$

In an applied calculus, there can be other kinds of values apart from closures.

Closures permit a correct treatment of lexical scope, and thus remedy the lacuna in our treatment of procedures. They can also correctly handle recursive functions (and other recursive data structures that are possible in a lazy language). Let vcl range over closures of the form $\ll\lambda x.e, \gamma\gg$. We give a big-step description for *cbn* and *cbv* simplifications of closures, which are basically restatements of the rules for $\Longrightarrow_n$ and $\Longrightarrow_v$. Very roughly, the judgments used for closure evaluation under strategy X $\ll e, \gamma\gg \Longrightarrow_{cl}^{X} \mathit{vcl}$ correspond to judgments $\gamma \vdash e \Longrightarrow_X v$ for expression evaluation, and where value closure vcl "unravels" to value v:

$$\frac{\gamma(x) \Longrightarrow_{cl}^{n} \mathit{vcl}}{\ll x, \gamma\gg \Longrightarrow_{cl}^{n} \mathit{vcl}}$$

$$\frac{\ll e_1, \gamma\gg \Longrightarrow_{cl}^{n} \ll\lambda x.e', \gamma'\gg \quad \ll e', \gamma'[x \mapsto \ll e_2, \gamma\gg]\gg \Longrightarrow_{cl}^{n} \mathit{vcl}}{\ll(e_1\ e_2), \gamma\gg \Longrightarrow_{cl}^{n} \mathit{vcl}}$$

For *cbv* the rules are:

$$\frac{\gamma(x) \Longrightarrow_{cl}^{v} \mathit{vcl}}{\ll x, \gamma\gg \Longrightarrow_{cl}^{v} \mathit{vcl}}$$

$$\frac{\begin{array}{c}\ll e_1, \gamma\gg \Longrightarrow_{cl}^{v} \ll\lambda x.e', \gamma'\gg \quad \ll e_2, \gamma\gg \Longrightarrow_{cl}^{v} \mathit{vcl}_2 \\ \ll e', \gamma'[x \mapsto \mathit{vcl}_2]\gg \Longrightarrow_{cl}^{v} \mathit{vcl}\end{array}}{\ll(e_1\ e_2), \gamma\gg \Longrightarrow_{cl}^{v} \mathit{vcl}}$$

It is also possible to formulate a calculus of closures [19] and study properties such as confluence of its reduction relation, which is weak in the sense that reduction does not occur below abstractions.

22.4.3.2 Abstract Machines

The big-step semantics suggests using a stack of closures that are yet to be simplified, or which are awaiting their arguments. Using this insight, environment machines can be developed, manipulating closures.

An environment machine for *cbn* due to Krivine is:

$$
\begin{aligned}
\langle \ll x, \gamma \gg,\ \lfloor S \rfloor \rangle &\relbar\!\!\rhd \langle \gamma(x),\ \lfloor S \rfloor \rangle \\
\langle \ll (e_1\ e_2), \gamma \gg,\ S \rangle &\relbar\!\!\rhd \left\langle \ll e_1, \gamma \gg,\ \left\lfloor \begin{array}{c} \ll e_2, \gamma \gg \\ S \end{array} \right\rfloor \right\rangle \\
\left\langle \ll \lambda x.e, \gamma \gg,\ \left\lfloor \begin{array}{c} cl \\ S \end{array} \right\rfloor \right\rangle &\relbar\!\!\rhd \langle \ll e, \gamma[x \mapsto cl] \gg,\ \lfloor S \rfloor \rangle
\end{aligned}
$$

The machine configurations consist of a *current* closure to be simplified and a stack of closures that are (yet-to-be evaluated) arguments to the current term. The first rule is a lookup. The second rule stacks the closure consisting of argument l_2 together with the current environment (in which it should be evaluated) onto the stack of yet-to-be-evaluated closures. The third rule starts the evaluation of the body in a closure after extending the environment with a binding of formal x to the argument closure cl, which is atop the stack.

The corresponding environment machine for *cbv* is:

$$
\begin{aligned}
\langle \ll x, \gamma \gg,\ \lfloor S \rfloor \rangle &\relbar\!\!\rhd \langle \gamma(x),\ \lfloor S \rfloor \rangle \\
\langle \ll (e_1\ e_2), \gamma \gg,\ \lfloor S \rfloor \rangle &\relbar\!\!\rhd \left\langle \ll e_1, \gamma \gg,\ \left\lfloor \begin{array}{c} \searrow \\ \ll e_2, \gamma \gg \\ S \end{array} \right\rfloor \right\rangle \\
\left\langle vcl,\ \left\lfloor \begin{array}{c} \searrow \\ \ll e, \gamma \gg \\ S \end{array} \right\rfloor \right\rangle &\relbar\!\!\rhd \left\langle \ll e, \gamma \gg,\ \left\lfloor \begin{array}{c} \swarrow \\ vcl \\ S \end{array} \right\rfloor \right\rangle \\
\left\langle vcl,\ \left\lfloor \begin{array}{c} \swarrow \\ \ll \lambda x.e, \gamma \gg \\ S \end{array} \right\rfloor \right\rangle &\relbar\!\!\rhd \langle \ll e, \gamma[x \mapsto vcl] \gg,\ S \rangle
\end{aligned}
$$

The *cbv* machine is not much different, except that both operator and operand are to be evaluated before application. For this, two markers $\searrow$ and $\swarrow$ are used to indicate that the closure below it on the stack is the argument and operator of an application, respectively. The third rule swaps the evaluated operand and unevaluated operators between the current-closure and the top-of-stack positions.

Both machines are loaded with a closure consisting of a closed term and empty environment, with an empty stack. The *unload* function involves unfolding the resulting closure, using the packaged environment to obtain the terms bound to variables (recursively unfolding closures).

22.4.3.3 SECD Machine

The prototypical machine used for *cbv* evaluation of a functional language was the SECD machine [41]. This machine works with two stacks — S for already evaluated expressions and "dump" D for managing control during function call and return — an environment E and a list of opcodes C. Stack S is used in much the same way as the stack is used for expression evaluation — the closures to which expressions evaluate are pushed onto it. Dump D is used as a repository for storing the calling context (the current environment, the subexpressions already evaluated prior to the call, and the code to be evaluated after the call) when a function call is made; this context can then be restored from the top of the dump on completion of a function call.

To avoid introducing new symbols, we use (following [61]) the λ-terms themselves as opcodes, with one additional opcode for function application *app*. The empty sequence is denoted by E.

$$\left\langle \left\lfloor \begin{matrix} cl \\ S \end{matrix} \right\rfloor, \gamma, \epsilon, \left\lfloor \begin{matrix} \langle S',\ \gamma',\ c'\rangle \\ D \end{matrix} \right\rfloor \right\rangle \longrightarrow \left\langle \left\lfloor \begin{matrix} cl \\ S' \end{matrix} \right\rfloor, \gamma', c', D \right\rangle$$

$$\langle \lfloor S \rfloor, \gamma, x :: c, \lfloor D \rfloor \rangle \longrightarrow \left\langle \left\lfloor \begin{matrix} \gamma(x) \\ S \end{matrix} \right\rfloor, \gamma, c, \lfloor D \rfloor \right\rangle$$

$$\langle \lfloor S \rfloor, \gamma, \lambda x.e :: c, \lfloor D \rfloor \rangle \longrightarrow \left\langle \left\lfloor \begin{matrix} \ll\lambda x.e, \gamma\gg \\ S \end{matrix} \right\rfloor, \gamma, c, \lfloor D \rfloor \right\rangle$$

$$\langle \lfloor S \rfloor, \gamma, (e_1\ e_2) :: c, \lfloor D \rfloor \rangle \longrightarrow \langle \lfloor S \rfloor, \gamma, e_1 :: e_2 :: app :: c, \lfloor D \rfloor \rangle$$

$$\left\langle \left\lfloor \begin{matrix} cl \\ \ll\lambda x.e, \gamma'\gg \\ S \end{matrix} \right\rfloor, \gamma, app :: c, \lfloor D \rfloor \right\rangle \longrightarrow \left\langle \lfloor , \rfloor \gamma'[x \mapsto cl], e, \left\lfloor \begin{matrix} \langle S,\ \gamma,\ c\rangle \\ D \end{matrix} \right\rfloor \right\rangle$$

The first rule describes function return; it says that if the current call has no remaining instructions, the calling context is restored from the dump — the returned value placed atop the caller stack, and the environment and code list of the caller are restored. The second rule is a variable look-up. The third rule forms and places a closure atop the value stack. The fourth rule is really a compilation rule that evaluates operator and operand expressions of an application (it is possible to separate the execution and compilation phases). The fifth rule is the actual function call rule. It assumes that the operand (argument) closure *cl* sits above the operator (function) closure atop the stack. Closure *cl* is bound to the formal argument x in the operator closure environment, the operator closure code is now made the code list, and the calling context is placed atop the dump. As indicated earlier, the calling context consists of the stack below the operator and operand closures, the calling environment and the remaining code list.

The SECD machine has been used as a template for a variety of block-structured languages, as we discuss later. Plotkin [61] has related the abstract machine semantics with the big-step and reduction semantics of an applied *cbv* λ-calculus using standardization to establish the correspondence.

22.4.3.4 Other Abstract Machines

Various other abstract machine implementations exist that we cannot describe here. One such class of machines is based on a translation of the λ-calculus into combinatory logic and an implementation of these combinators [71]. A special class of implementations are based on graph reduction (see [37] and various references therein for an accessible treatment of such implementations). The main operations of these machines involve performing rearrangements of a syntax tree (or graph) according to certain combinators or directors [39]. Also significant is the categorical abstract machine [15], which is based on operative features of categorical models of λ-calculi, and the closely related machine derived by Hannan and Miller [46].

22.4.4 Implementation Issues Related to Environments

The abstract machines seem rather profligate in the structures they employ. Fortunately, there are rather efficient implementations of environments, and closures using stacks, pointers and allocation on stack and heap. The observation that the called function never looks at the caller's stack in the SECD machine suggests that the value stack does not need storing, only the (re)storing of the stack pointer. Likewise, entire code lists and environments do not need to be stowed away on the dump; pointers to them suffice.

Efficient environment implementation and management are crucial. First, the environment is maintained as a stack of references to local frames. Then variables are replaced by a fast indexing scheme relative to a frame pointer (*cf.* de Bruijn indices in the λ-calculus).

22.4.4.1 Recursion

Special mention must be made about recursive functions. As mentioned earlier, simply typed languages cannot have a Y combinator, so a special mechanism is needed to build closures for recursive functions and recursive data structures. A simple idea is to build a circular reference into the environment component of the closure for a recursive function. This is achieved using two opcodes, introducing a level of indirection in environments.[9] The first opcode places a reference to a dummy closure. The closure for the recursive function is created using this augmented environment, and a second opcode overwrites the reference to the dummy reference with a pointer to the new closure, thus building the cycle (see [12, 29] for a simple implementation).

22.4.4.2 Local Definitions

Local definitions may be implemented in correspondence to the parameter-passing mechanism, employing the equivalence:

$$(\lambda x.e_2\; e_1) \;\approx\; \mathbf{let}\; x \overset{def}{=} e_1 \;\mathbf{in}\; e_2$$

or its generalization to more structured definitions. However, such a crude approach is rarely followed, because it is inefficient. Exploiting the fact that the environment used for e_2 is an extension of that used for e_1, much simpler, direct methods are possible, in particular, by employing finer grain opcodes that facilitate stack manipulation and making definitions and recursive definitions.

22.4.4.3 Extensions

The SECD framework is fairly robust, and can easily be extended to deal with a variety of language extensions, including side effects. Adding a store component and related opcodes [13] allows support for imperative features. Similarly, input and output streams can be accommodated, as also can communication and concurrency primitives (a general choice operator is difficult to incorporate) [22].

22.4.4.4 Procedures in Imperative Languages

By the principle of abstraction, the notion of closures carries over to command abstracts. Of course, some aspects are simpler (languages with higher order procedures are rare beasts), whereas issues pertaining to stores are somewhat more involved. In particular, showing that the allocation and deallocation of locations is done correctly is an important part of proving that the language and implementation are free of storage insecurities.

The typical call-stack management in traditional imperative languages can be seen as an implementation where three different stack structures — S for temporary computation, E for the environment and D for the dump — are "multiplexed" onto one physical stack.

22.4.5 Control Operators

We briefly discuss here the operational semantics for an extension of the λ-calculus with control operators that can pass or throw away the current evaluation context. Control operators allow functional programs to handle features such as concurrent threads, exceptions and *call/cc*.

[9]This already exists in most pointer-based implementations of environments.

They support a technique used in modern compilers, namely, that of passing continuations [4]. Moreover, the environment machines given earlier have simple extensions to deal with these new control operators.

The syntax is extended with two new unary operations, which are also redexes:

$$e ::= \ \dots \ \mid \mathcal{C}e \mid \mathcal{A}e$$

whose reduction rules, stated in contextual form, are:

$$(\mathcal{C}) \quad \frac{}{E[\mathcal{C}e] \ \longrightarrow_1^e \ (e\ (\lambda x.\mathcal{A}E[x]))} \quad x \notin fv(e)$$

$$(\mathcal{A}) \quad \frac{}{E[\mathcal{A}e] \ \longrightarrow_1^e \ e}$$

In the rule $(\mathcal{A})$, the abort operator throws away the current evaluation context, whereas in the rule $(\mathcal{C})$, the control operator passes an abstracted form of the current evaluation as an argument to the expression e.

Another well-known control operator is call with current continuation (*call/cc*), with the following operational rule:

$$(call/cc) \quad \frac{}{E[call/cc(\lambda k.e)] \ \longrightarrow_1^e \ ((\lambda k.(k\ e))\ (\lambda x.\mathcal{A}E[x]))} \quad x \notin fv(e)$$

an equivalent of which can be expressed in terms of $(\mathcal{C})$ and $(\mathcal{A})$.

22.4.5.1 Environment Machines for Control Operations

Recall that the stack component S of an environment machine represents the context E of the current expression under evaluation. The control operators manipulate this evaluation context. Therefore, operations to encapsulate and manipulate the stack are introduced: a new kind of closure $retr(S)$ is added that corresponds roughly to $\lambda x.\mathcal{A}E[x]$.

The new rules for the Krivine machine are:

$$\langle \ll \mathcal{C}e, \gamma \gg, \ \lfloor S \rfloor \rangle \longrightarrow \langle \ll e, \gamma \gg, \ \lfloor retr(S) \rfloor \rangle$$

$$\langle \ll \mathcal{A}e, \gamma \gg, \ \lfloor S \rfloor \rangle \longrightarrow \langle \ll e, \gamma \gg, \ \lfloor \quad \rfloor \rangle$$

$$\left\langle retr(S), \ \left\lfloor \begin{array}{c} cl \\ S' \end{array} \right\rfloor \right\rangle \longrightarrow \langle cl, \ \lfloor S \rfloor \rangle$$

The manipulations of the context are fairly clear: in the first rule, the current stack is encapsulated and presented as an argument to the closure corresponding to e. The abort operator throws away the current stack. In the third rule, the encapsulated stack is restored, in place of the existing stack S'.

The *cbv* environment machine uses the same rules as before with three additional rules for manipulating the stack. Of these, the second rule (for abort) is the same as the rule in the extension of the Krivine machine:

$$\langle \ll \mathcal{C}e, \gamma \gg, \ \lfloor S \rfloor \rangle \longrightarrow \langle \ll e, \gamma \gg, \ \left\lfloor \begin{array}{c} \searrow \\ retr(S) \end{array} \right\rfloor \rangle$$

$$\langle \ll \mathcal{A}e, \gamma \gg, \ \lfloor S \rfloor \rangle \longrightarrow \langle \ll e, \gamma \gg, \ \lfloor \quad \rfloor \rangle$$

$$\left\langle vcl, \ \left\lfloor \begin{array}{c} \swarrow \\ retr(S) \\ S' \end{array} \right\rfloor \right\rangle \longrightarrow \langle vcl, \ \lfloor S \rfloor \rangle$$

If $retr(S)$ corresponds to $\lambda x.\mathcal{A}E[x]$, and S' corresponds to context $E'[\ ]$, then the last rule can be seen as implementing the reduction sequence:

$$E'[(\lambda x.\mathcal{A}E[x]\ v)] \longrightarrow_1^v E'[\mathcal{A}E[v]] \longrightarrow_1^v E[v]$$

22.4.5.2 Translating the Control Operators Away

An important result [21, 24, 61] is that these control operators can be translated away by so-called *CPS transformations* into purely functional languages. We introduce the idea here only to indicate how operational techniques are used in language translations, because a proper treatment of CPS is well beyond the scope of this chapter. We present one such translation, which lets us interpret call-by-value reduction as call-by-name reduction [61]:

$$\overline{x} = \lambda k.(k\ x) \qquad \overline{(e_1\ e_2)} = \lambda k.(\overline{e_1}\ (\lambda m.(\overline{e_2}\ (\lambda n.((m\ n)\ k)))))$$
$$\overline{a} = \lambda k.(k\ a) \qquad \overline{\mathcal{C}e} = \lambda k.(\overline{e}\ (\lambda m.((m\ (\lambda n.\lambda d.(k\ n)))\ (\lambda x.x))))$$
$$\overline{\lambda x.e} = \lambda k.(k\ (\lambda x.\overline{e})) \qquad \overline{\mathcal{A}e} = \lambda k.(\overline{e}\ (\lambda x.x))$$

Various interesting theorems can be shown about this CPS translation. For example:

THEOREM 22.5. *For any pure λ-expression e:* $(\overline{e}\ (\lambda x.x)) \Longrightarrow_n v$ *if and only if* $(\overline{e}\ (\lambda x.x)) \Longrightarrow_v v$.

THEOREM 22.6. *For any λ-expression e without control operators, and of base type (not of a function type):*[10] $e\ (\longrightarrow_1)^*\ v$ *if and only if* $(\overline{e}\ (\lambda x.x))\ (\longrightarrow_1)^*\ v$.

22.5 Labeled Transition Systems and Interactive Programs

The formulations we have presented so far have used TSs without labels. Until now we have concentrated on programs in isolation from their operating environment. However, programs interact with their execution environment, at the very least for input and output of data. Even in an isolated computer, various interactions occur with peripheral devices such as disks, printers, file systems and libraries. Also interactions take place with forked processes, interrupt handlers, etc.

The picture we have presented so far can be sustained when interaction with the environment is clearly separated from computation. However, programming nowadays is increasingly interactive, and all programming languages provide facilities for interaction with the environment. In addition, several languages provide features for concurrent and distributed execution. Interactions can take the form of remote procedure calls, or communication in a network, cluster or distributed computing environment, interspersed in the computation. In other words, interaction becomes an integral part of computation.

Central to this kind of interactive computing are the concepts of *process* and *communication* (the texts [30, 35, 51] provide excellent introductions to the area). A program and its environment can be considered two processes that communicate with each other. These two processes may themselves consist of collections of interacting processes.

When integrating interaction into computation, certain issues arise in providing structured operational descriptions. First, the Fregean principle of compositionality should still be applicable. Second,

[10]Such expressions can be considered complete programs in a typed λ-calculus. The result depends on strong normalization of the typed λ-calculus.

the fact that processes interact while executing concurrently brings in its own complexity because interactions may alter the state of a program nondeterministically. Third, the fact that a program operates correctly only under circumstances where the environment fulfills certain obligations implies that both the program and its environment (regarded as a process) cooperate in achieving certain goals. Specifications must clearly define interfaces of interaction that constrain the kinds of inputs a process can receive, the outputs it can produce and how it synchronizes with other components in a system. Finally, one cannot place unreasonable restrictions on the environment. For example, it would be unreasonable to expect a remote server to operate at the same speed as one or several of its clients. Hence, concurrent execution, in general, implies that different processes execute at different speeds and interactions are the only means of achieving certain synchronizations.

22.5.1 Labels and Behavior

Labels are a convenient device to indicate interaction between a program and its environment during execution. They carry information about communication capabilities of processes and are often crucial to the changes in state that processes incur. They are also used to determine and resolve nondeterministic choices in the execution of a process when it has the possibility of interacting with several other processes at the same time.

In TSs confluence, determinacy and termination are important properties and two sequential systems are considered equal if they compute the same function between input and output states. Concurrent systems, on the other hand, are generally nondeterministic (mostly nonconfluent), and often infinite-state, nonterminating systems; also they may not be computing a particular relation or function. What are the corresponding notions of behavioral properties in LTSs? The crucial properties of such systems concern their interaction capabilities. Any equality relation on such systems must naturally relate to the communication capabilities of the individual processes that make up the system.

Various notions of behavior can be associated with an LTS, based on the idea that the observable behavior of a process depends on the sequences of labeled transitions it can perform. However, there is little consensus yet on what is the right notion of behavior. A simple, language-theoretical notion of program behavior is the set of sequences (finite or infinite) of labels or traces. A process p has trace $\varsigma = l_1 l_2 \ldots \in \mathcal{L}^\omega = \mathcal{L}^* \bigcup \mathcal{L}^{\text{inf}}$ if it can perform a sequence of labeled transitions $p \xrightarrow{l_1} p_1 \xrightarrow{l_2} p_2 \ldots$. Two processes are considered trace equivalent if they have the same traces.

Other notions of behavior take into account the communication capabilities (and incapabilities) at each intermediate state, thus being sensitive to the possibility of deadlock — inability to perform a transition with a particular label — in some sequences of transitions (see Examples 22.8 and 22.9 that follow). We present only one such finer notion, called *bisimulation* [59].

The intuition is that this notion of equivalence identifies a pair of processes, if starting from equivalent states they have the same interaction possibilities, the success of each of which puts them again in states that may be considered equivalent.

Definition 22.9.

- *A binary relation $\mathcal{R}$ on process configurations is a* simulation *if whenever $s_1 \mathcal{R} s_2$, for any $l \in \mathcal{L}$, if $s_1 \xrightarrow{l} s_1'$, then there exists a configuration s_2' such that $s_2 \xrightarrow{l} s_2'$ and $s_1' \mathcal{R} s_2'$.*
- *$\mathcal{R}$ is a* bisimulation *if $\mathcal{R}$ and $\mathcal{R}^{-1}$ (the symmetrical inverse of $\mathcal{R}$) are both simulations.*
- *The collection of bisimulation relations is closed under inverse, composition and arbitrary union. The largest bisimulation called* bisimilarity *is denoted $\approx$ and is an equivalence relation.*

Proving two labeled transitions systems are bisimilar involves proposing and proving a particular relation is a bisimulation. Bisimulation equivalence, or bisimilarity, is a finer notion of equivalence than trace equivalence, because it distinguishes more programs than trace equivalence does. In particular, it is sensitive to the potential for deadlock behavior — two processes with the same traces are distinguished if on some trace, one of them can reach a state where some particular actions are possible whereas the other cannot reach such a corresponding state on that same trace. In fact, bisimilarity is the finest deadlock-sensitive equivalence relation on processes obtained from examining their observable behavior. In practice, a variety of notions can be considered bisimulations, for different notions of labeled transition, for difference in the precise characterization of the labeled actions or for difference in what exactly is observable, etc. Different characterizations may also exist for a single notion of bisimulation, with alternative characterizations supporting different styles of reasoning. There are also a variety of different notions of equivalence that lie between trace equivalence and bisimulation, some of which are fairly natural notions of equivalence to work with. A full exploration of these issues is beyond the scope of this chapter; a quick introduction is provided in [2].

22.5.2 Communicating Sequential Processes

We illustrate the use of LTSs in semantic specification through a language based on communicating sequential processes (CSP) due to Hoare [34, 35]. The language extends the language of guarded choice (which already includes nondeterminism) with new constructs for communication and concurrent execution. The semantics we give here is a simplification of a presentation due to Plotkin [64].

It is often difficult to present purely big-step or purely small-step semantics for interactive programming languages, which incorporate internal evaluation of expressions. This is because communicating concurrent systems are best described using small-step descriptions, because they can account for interleavings and interactions from intermediate states (particularly important in notions of behavior sensitive to deadlock), whereas expressions are evaluated in entirety (and can easily be specified in a big step).

The syntax for CSP is as follows:

$$io ::= P?in \mid Q!out$$

$$g ::= e \mid e; io$$

$$\begin{aligned} c ::= {} & x := e \mid P?in \mid Q!out \mid c; c \\ & \mid \textbf{if } \Box_{i=1}^{n} g_i \ \triangleright\ c_i \textbf{ fi} \mid \textbf{do } \Box_{i=1}^{n} g_i \ \triangleright\ c_i \textbf{ od} \end{aligned}$$

$$S ::= [\ \|_{i=1}^{n} P_i :: c_i\ ]$$

where io stands for input and output communication statements; g for guards, which are boolean expressions, optionally followed by a communication. Commands include communication statements, assignments and guarded choice and iteration constructs. A program S consists of a collection of named processes. For simplicity we assume that concurrent execution takes place only at the topmost level (i.e., processes cannot have subprocesses that themselves execute concurrently). Every process has a name that is known to other processes. Communication between processes is by synchronized handshaking or rendezvous, wherein two named processes that need to exchange values wait at matching input and output commands, respectively, before consummating the communication. The command $P?in$ indicates that the current process can wait to input a value from the process named P, and $Q!out$ represents a desire to output a value out to the process named Q; the sending process is willing to wait until Q is ready to input the value.

Example 22.8
Assume there is a printer shared by two processes P_1 and P_2. Both processes and the printer are modeled as CSP processes, which together form a closed system:

$$\begin{array}{ll} [& P_1 :: \textbf{do } \neg done_1 \;\triangleright\; local_{1,1};\, PR!e \;\textbf{od};\, PR!eot;\, local_{1,2} \\ \| & P_2 :: \textbf{do } \neg done_2 \;\triangleright\; local_{2,1};\, PR!e;\, \textbf{od}\; PR!eot;\, local_{2,2} \\ \| & PR :: \textbf{do } \Box_{i=1}^{2}\; \textbf{true};\, P_i?v \;\triangleright\; \textbf{do } v \neq eot \;\triangleright\; print(v);\, P_i?v;\; \textbf{od od} \\ ] & \end{array}$$

The printer process *PR* waits until one of the two processes P_1, P_2 is ready to begin transmission, with the first value. In case both processes want to output to the printer, *PR* has to make a choice. Having chosen to communicate with one of them, the printer does not serve the other process until the chosen one sends an end-of-transmission (eot) signal. The printer process never terminates because it keeps waiting indefinitely for P_1 or P_2 to communicate with it.[11] It is possible for one process to monopolize the printer and prevent the other process from ever gaining access. The commands $local_{i,j}$ represent local computation in the processes P_1 and P_2.

Each process has its own state and the states of the different processes are disjoint. All changes in state σ_i of a process P_i are due to local assignments or receipt of input from another process. The set of global states defined as:

$$State \;=\; \bigotimes_{i=1}^{n} State_i$$

is the Cartesian product of the sets of the states of individual processes, where $State_i$ is the set of states of the process P_i. The metavariable $\bar{\sigma}$ denotes the global state and each σ_i stands for the state of process P_i. The labels we use for our LTS consist of the set of possible communications, defined as:

$$\begin{aligned} Inputs &= \{P?v \mid P \text{ is a process name and } v \in \mathcal{V}\} \\ Outputs &= \{P!v \mid P \text{ is a process name and } v \in \mathcal{V}\} \\ \mathcal{L} &= Inputs \cup Outputs \cup \{\varepsilon\} \end{aligned}$$

The label ε signifies local computation that involves no interaction with other processes. λ is a metavariable that ranges over $\mathcal{L}$.

The semantics of the commands in a process P_i are given in Table 22.9. We assume that $j \neq i$. The *Input* rule says that process P_i attempting to receive a value from process P_j can, on receipt of *any* value v from P_j, bind v to a variable x in its local state. Expression e is evaluated to a value v before the process attempts to send it to P_j, the statement terminating if and when P_j accepts this communication. Assignment is considered an internal action that does not affect other processes, and the transition is labeled with ε. In the rules *Seq* and *Int* we abstract from the internal computations of a process by coalescing local changes of state (labeled with ε) into a single labeled transition. The last rule abstracts from local computations and highlights an interaction, whenever there is one. Observe that the *Int* rules are not syntax directed.

[11]This interpretation is at variance with the so-called distributed termination convention that Hoare originally proposed in the language. However, we find our interpretation more suitable for server processes. It also illustrates that we are now in an arena where we deal with systems that do not necessarily always terminate. Indeed in concurrent systems, guaranteeing properties such as deadlock-freedom, nontermination and freedom from starvation may be more important.

TABLE 22.9 Mixed-Step Semantics for CSP Commands

Input	$\dfrac{}{\langle \sigma_i,\ P_j?x\rangle \xrightarrow{P_j?v} \sigma_i[x \mapsto v]}$	
Output	$\dfrac{\sigma_i \vdash e \Longrightarrow_e v}{\langle \sigma_i,\ P_j!e\rangle \xrightarrow{P_j!v} \sigma_i}$	
Assign	$\dfrac{\sigma_i \vdash e \Longrightarrow_e v}{\langle \sigma_i,\ x := e\rangle \xrightarrow{\varepsilon} \sigma_i[x \mapsto v]}$	
Seq	$\dfrac{\langle \sigma_i,\ c_1\rangle \xrightarrow{\varepsilon} \sigma_i' \quad \langle \sigma_i',\ c_2\rangle \xrightarrow{\varepsilon} \sigma_i''}{\langle \sigma_i,\ c_1; c_2\rangle \xrightarrow{\varepsilon} \sigma_i'}$	
Int₁	$\dfrac{\langle \sigma_i,\ c\rangle\ (\xrightarrow{\varepsilon})^* \xrightarrow{\lambda} (\xrightarrow{\varepsilon})^*\ \langle \sigma_i',\ c'\rangle}{\langle \sigma_i,\ c\rangle \xrightarrow{\lambda} \langle \sigma_i',\ c'\rangle}$	$\lambda \neq \varepsilon$
Int₂	$\dfrac{\langle \sigma_i,\ c\rangle\ (\xrightarrow{\varepsilon})^* \xrightarrow{\lambda} (\xrightarrow{\varepsilon})^*\ \sigma_i'}{\langle \sigma_i,\ c\rangle \xrightarrow{\lambda} \sigma_i'}$	$\lambda \neq \varepsilon$

TABLE 22.10 Labeled Semantics for CSP Processes

$Process_i$	$\dfrac{\langle \sigma_i,\ c_i\rangle \xrightarrow{\lambda} \langle \sigma_i',\ c_i'\rangle}{\langle \sigma_i,\ p_i\rangle \xrightarrow{\lambda}_p \langle \sigma_i',\ p_i'\rangle}$
$Par_{interleave}$	$\dfrac{\langle \sigma_i,\ p_i\rangle \xrightarrow{\varepsilon}_p \langle \sigma_i',\ p_i'\rangle}{\langle \bar{\sigma},\ S\rangle \xrightarrow{\varepsilon}_p \langle \bar{\sigma}',\ S'\rangle}$
Par_{sync}	$\dfrac{\langle \sigma_i,\ p_i\rangle \xrightarrow{P_j!v}_p \langle \sigma_i',\ p_i'\rangle \qquad \langle \sigma_j,\ p_j\rangle \xrightarrow{P_i?v}_p \langle \sigma_j',\ p_j'\rangle}{\langle \bar{\sigma},\ S\rangle \xrightarrow{\varepsilon}_p \langle \bar{\sigma}',\ S'\rangle}$

We now deal with the parallel composition of processes. The transitions of processes (as opposed to commands) are also labeled (e.g., $\xrightarrow{\lambda}_p$) and have a subscript p to distinguish them from the transition relation $\longrightarrow$ (used in Table 22.9) for command transitions.

For readability, we follow the following notational conveniences in Table 22.10.

- For any global state $\bar{\sigma}$, σ_k denote the kth component of the n-tuple ($1 \leq k \leq n$).
- For each k, $1 \leq k \leq n$, $p_k \equiv P_k :: c_k$ and $p_k' \equiv P_k :: c_k'$.
- In rules $Par_{interleave}$ and Par_{sync}:

$$S \equiv [\|_{k=1}^n\ p_k], \qquad S' \equiv [\|_{k=1}^n\ p_k']$$

- In rule $Par_{interleave}$:

$$\sigma_k' = \begin{cases} \sigma_i' & \text{if } k = i \\ \sigma_k & \text{otherwise} \end{cases}, \qquad c_k' \equiv \begin{cases} c_i' & \text{if } k = i \\ c_k & \text{otherwise} \end{cases}$$

- In rule Par_{sync}:

$$\sigma'_k = \begin{cases} \sigma'_i & \text{if } k = i \neq j \\ \sigma'_j & \text{if } k = j \neq i \\ \sigma_k & \text{otherwise} \end{cases}, \qquad c'_k \equiv \begin{cases} c'_i & \text{if } k = i \neq j \\ c'_j & \text{if } k = j \neq i \\ c_k & \text{otherwise} \end{cases}$$

In Table 22.10:

- The rule Par_{sync} treats a closed system of processes. Hence, all communications between components of the system are internal to the system.
- The system of processes terminates only if every process in the system terminates. In other words, configurations of the form $\langle \bar{\sigma},\ [\|_{k=1}^{n} P_k :: \circ] \rangle$ (where "$\circ$" denotes an empty continuation) are the only terminal configurations.
- If the system reaches a stuck configuration, then it is said to be *deadlocked*. In other words, a configuration $\langle \sigma,\ S \rangle$, which is not terminal and such that $\langle \bar{\sigma},\ S \rangle \overset{\varepsilon}{\not\longrightarrow}_p$ is deadlocked.

Table 22.11 contains the rules for guards using yet another labeled transition system, which is then used in giving the semantics of the conditional and iterations constructs (Table 22.12).

TABLE 22.11 Mixed-Step Semantics for Guards

$$\frac{\sigma \vdash e_j \Longrightarrow_e \mathbf{true}}{\langle \sigma,\ e_j \triangleright c_j \rangle \xrightarrow{\varepsilon}_g \sigma}$$

$$\frac{\sigma \vdash e_j \Longrightarrow_e \mathbf{true} \quad \langle \sigma,\ io_j \rangle \xrightarrow{\lambda} \sigma'}{\langle \sigma,\ e_j; io_j \rangle \xrightarrow{\lambda}_g \sigma'}$$

TABLE 22.12 Semantics of **if** $-$ **fi** and **do** $-$ **od**

Let $\mathbf{IF} \equiv \mathbf{if}\ \Box_{i=1}^{n} g_i \ \triangleright\ c_i\ \mathbf{fi}$

and $\mathbf{DO} \equiv \mathbf{do}\ \Box_{i=1}^{n} g_i \ \triangleright\ c_i\ \mathbf{od}$

$$\frac{\langle \sigma,\ g_j \rangle \xrightarrow{\lambda}_g \sigma'}{\langle \sigma,\ \mathbf{IF} \rangle \xrightarrow{\lambda} \langle \sigma',\ c_j \rangle} \qquad (j \in \{1, \ldots, n\})$$

$$\frac{\langle \sigma,\ g_j \rangle \xrightarrow{\lambda}_g \sigma'}{\langle \sigma,\ \mathbf{DO} \rangle \xrightarrow{\lambda} \langle \sigma',\ c_j;\ \mathbf{DO} \rangle} \qquad (j \in \{1, \ldots, n\})$$

$$\frac{\bigwedge_{i=1}^{n} \sigma \vdash e_i \Longrightarrow_e \mathbf{false}}{\langle \sigma,\ \mathbf{DO} \rangle \xrightarrow{\varepsilon} \sigma}$$

The following example illustrates some of the distinctions that can arise due to nondeterminism.

Example 22.9

Compare the process *PR* in Example 22.8 with the following alternative version:

$$PR' :: \mathbf{do}\ \Box_{i=1}^{2}\ \mathbf{true} \triangleright P_i?v;\ \mathbf{do}\ \ v \neq eot \triangleright print(v);\ P_i?v;\ \ \mathbf{od}\ \ \mathbf{od}$$

The major difference between *PR* and *PR*′ is in their deadlock behavior. Whereas *PR* may wait until one of the processes is ready to communicate with it, *PR*′ is forced to make a commitment to wait on one of the two processes say P_1, regardless of whether P_1 wants to communicate with it. *PR*′ clearly exacerbates the possibilities of deadlock in the system. Therefore, *PR* and *PR*′ cannot be considered equivalent as processes.

22.5.3 Extensions

We conclude this discussion with some language features that can easily be modeled in the framework of LTSs.

22.5.3.1 Input and output

Commands are extended with input and output primitives:

$$c \ ::= \ \ldots \ \mid \ \mathbf{read}(x) \ \mid \ \mathbf{write}(e)$$

Input and output are really special cases of communication, but instead of interacting with a named process, values are taken from and added to stream data structures. The command level rules are (following the convention mentioned earlier):

$$\mathit{Read}\ \frac{}{\langle \sigma_i,\ \mathbf{read}(x)\rangle \xrightarrow{?v} \sigma_i[x \mapsto v]}$$

$$\mathit{Write}\ \frac{\sigma_i \ \vdash \ e \Longrightarrow_e v}{\langle \sigma_i,\ \mathbf{write}(e)\rangle \xrightarrow{!v} \sigma_i}$$

Two new kinds of labels are added, for reading and writing:

$$l \in \mathcal{L} \ ::= \ \ldots \mid !v \mid ?v$$

At the global configuration level, (global) input and output streams are added. The labels generated at the command level are discharged at the top level, with the corresponding manipulations of the input and output (I/O) streams ς_i, ς_o:

$$\mathit{Read}\ \frac{\langle \sigma_i,\ c_i\rangle \xrightarrow{?v} \langle \sigma_i',\ c_i'\rangle}{\langle \sigma_i, p_i, v\varsigma_i, \varsigma_o\rangle \xrightarrow{\varepsilon}_p \left\langle \sigma_i', p_i', \varsigma_i, \varsigma_o\right\rangle}$$

$$\mathit{Write}\ \frac{\langle \sigma_i,\ c_i\rangle \xrightarrow{!v} \langle \sigma_i',\ c_i'\rangle}{\langle \sigma_i, p_i, \varsigma_i, \varsigma_o\rangle \xrightarrow{\varepsilon}_p \left\langle \sigma_i', p_i', \varsigma_i, \varsigma_o v\right\rangle}$$

22.5.3.2 Dynamic Process Creation

Consider a command **fork**(P, c), which dynamically creates a new process named P executing the command c. At the command level, the effect of this command returns the state unchanged, but generates a new kind of label $\Phi(\langle \sigma_i,\ P\ ::\ c\rangle)$. The state σ_i is cloned and packaged into the label:

$$\frac{}{\langle \sigma_i,\ \mathbf{fork}(P, c)\rangle \xrightarrow{\Phi(\langle \sigma_i,\ P::c\rangle)} \sigma_i}$$

where P is a new process name.

At the global configuration level, the label $\Phi(\langle\sigma_i,\ P :: c\rangle)$ is discharged, by creating a new process with its own local state:

$$\frac{\langle\sigma_i,\ p_i\rangle \overset{\Phi(\sigma_i, P::c)}{\longrightarrow}_p \langle\sigma_i',\ p_i'\rangle}{\langle\bar{\sigma},\ S\rangle \overset{\varepsilon}{\longrightarrow}_p \langle\bar{\sigma''},\ S''\rangle}$$

$S'' = [(\|_{k=1}^{n} p_k') \mid P :: c]$ and $\bar{\sigma''} = \bar{\sigma'} \otimes \sigma_i$, where we continue with the notational convention mentioned earlier. That is, the vector of process code has an $n+1^{th}$ component $P :: c$ the local state of which has a fresh copy of σ_i as its initial local state. The rule applies only under the assumption that P is a globally fresh process name.

22.6 Conclusion

We have seen the use of structural operational semantics both as a concise formalism and as a method of precisely defining the dynamic semantics of programming language constructs. The conciseness of the formalism makes it far easier to study and comprehend the potential bottlenecks that an implementor is likely to face. Because the semantics is syntax driven and the rules are essentially syntactic, it is also possible in many cases to generate prototypical implementations of new and so far untried constructs quickly with the help of scanning and parsing tools. One such tool for concurrent systems is the Process Algebra compiler of North Carolina [18].

In the case of both imperative and functional languages, we have chosen the semantics of a small core, built up new constructs and features and given them meaning. However, in general, an existing programming language cannot be extended by adding new features to it, without first considering how the existing features of the language interact with the new ones.

In many cases, the implementation strategies become clearer through such a rule-based exposition of the semantics. In certain cases, of course, we have chosen to define rules that are consistent with and model current implementation strategies.

We have not treated the semantics of structured data in general. We have also not treated the semantics of types or static semantic analysis. Although this is a major omission and is important for compiling, it would have taken us too far afield. Another significant omission is the semantics of modules, classes and objects much of which is still an area of active research. The Reference Section contains several references that the reader may consult to learn more about the work in the area.

References

[1] R.M. Amadio and P.-L. Curien, *Domains and Lambda-Calculi*, Cambridge University Press, London, 1998.

[2] L. Aceto, W. Fokkink and C. Verhoef, Structural operational semantics, in *Handbook of Process Algebra*, J.A. Bergstra, A. Ponse and S.A. Smolka, Eds., Elsevier, Amsterdam, 2000.

[3] E. Astesiano, C. Bendix Nielsen, N. Botta, A. Fantechi, A. Giovini, P. Inverardi, E. Karlsen, F. Mazzanti, G. Reggio and E. Zucca, The Trial Definition of Ada, Deliverable 7 of the CEC MAP Project: The Draft Formal Definition of ANSI/MIL-STD 1815 Ada, CEC MAP, 1986.

[4] A.W. Appel, *Compiling with Continuations*, Cambridge University Press, London, 1992.

[5] E. Astesiano, Inductive and Operational Semantics, in *Formal Description of Programming Concepts*, Springer-Verlag, 1991, pp. 53–134.

[6] H.P. Barendregt, *The Lambda Calculus, Its Syntax and Semantics*, Vol. 103, *Studies in Logic and the Foundation of Mathematics*, North-Holland, Amsterdam, 1984.

[7] G. Berry and L. Cosserat, The **Esterel** synchronous programming language and its mathematical semantics, in S.D. Brookes, A.W. Roscoe and G. Winskel, Eds., *Seminar on Concurrency, Lecture Notes in Computer Science*, Vol. 197, Springer-Verlag, New York, 1984, pp. 389–448.

[8] G. Berry and G. Gonthier, The ESTEREL synchronous programming language: design, semantics, implementation, *Sci. Comput. Programming*, 19(2), pp. 87–152, 1992.

[9] R. Burstall and F. Honsell, A natural deduction treatment of operational semantics, in *Proceedings of FST & TCS 8, Foundations of Software Technology and Theoretical Computer Science, Lecture Notes in Computer Science*, Vol. 287, Springer-Verlag, New York, 1987.

[10] R. Bornat, Proving pointer programs in Hoare Logic, in *Mathematics of Program Construction*, 2000, pp. 102–126.

[11] E. Borger and P.H. Schmitt, A formal operational semantics for languages of type prolog III, in *CSL*, 1990, pp. 67–79.

[12] L. Cardelli, Compiling a Functional Language, in *Proceedings of 1984 Symposium on LISP and Functional Programming*, 1984, pp. 208–217.

[13] L. Cardelli, Amber, in *Combinators and Functional Programming Language*, G. Cousineau, P-L. Curien and B. Robinet, Eds., *Lecture Notes in Computer Science*, Vol. 242, Springer-Verlag, New York, 1986.

[14] L. Cardelli, The amber machine, in *Combinators and Functional Programming Languages*, G. Cousineau, P.-L. Curien and B. Robinet, Eds., *Lecture Notes in Computer Science*, Vol. 242, Springer-Verlag, New York, 1986.

[15] G. Cousineau, P. Curien, and M. Mauny, The Categorical Abstract Machine, *Functional Programming Languages and Computer Architecture*, In J.-P. Jouannaud, Ed., *Lecture Notes in Computer Science*, Vol. 201, Springer-Verlag, New York, 1985, pp. 50–64.

[16] C. Calcagno, S. Ishtiaq and P.W. O'Hearn, Semantic Analysis of Pointer Aliasing, Allocation and Disposal in Hoare Logic, in *Proceedings 2nd International Conference on Principles and Practice of Declarative Programming*, Maurizio Gabbrielli and Frank Pfenning, Eds., ACM Press, New York, 2000.

[17] P. Cenciarelli, A. Knapp, B. Reus and M. Wirsing, An Event-Based Structural Operational Semantics of Multi-Threaded Java, in *Formal Syntax and Semantics of Java*, 1999, pp. 157–200.

[18] R. Cleaveland, E. Madelaine and S. Sims, A front-end generator for verification tools, in *Tools and Algorithms for the Construction and Analysis of Systems*, E. Brinksma, R. Cleaveland, K.G. Larsen and B. Steffen, Eds., *Lecture Notes in Computer Science*, Vol. 1019, Springer-Verlag, New York, 1995, pp. 153–173.

[19] P.-L. Curien, An abstract framework for environment machines, *Theor. Comput. Sci.*, 82(2), 389–402, 1991.

[20] F.Q.B. da Silva, Correctness Proofs of Compilers and Debuggers: An Approach Based on Structural Operational Semantics, Ph.D. thesis, Laboratory for Foundations of Computer Science, Computer Science Department, Edinburgh University, Edinburgh, EH9 3JZ, Scotland, September 1992; available as LFCS Report Series ECS-LFCS-92-241 or CST-95-92.

[21] M. Felleisen, D. Friedman, E. Kohlbecker and B. Duba, A syntactic theory of sequential control, *Theor. Comput. Sci.*, 52(3), 205–237, 1987.

[22] A. Giacalone, P. Mishra and S. Prasad, Facile: A symmetric integration of concurrent and functional programming, *Int. J. Parallel Programming*, 18(2), 121–160, 1989.

[23] G. Gonthier, Sémantiques et Modèles d'Exécution des Langages Réactifs Synchrone; Application à **Esterel**, Thèse d'informatique, Université d'Orsay, 1988.

[24] T.G. Griffin, The formulae-as-types notion of control, in *Conference Record 17th Annual ACM Symposium on Principles of Programming Languages, POPL '90, San Francisco, CA*, January 17–19, 1990, ACM Press, New York, 1990, pp. 47–57.

[25] C.A. Gunter, *Semantics of Programming Languages: Structures and Techniques, Foundations of Computing*, MIT Press, Cambridge, MA, 1992.

[26] Y. Gurevich, Evolving algebras: an attempt to discover semantics, in *Current Trends in Theoretical Computer Science*, G. Rozenberg and A. Salomaa, Eds., World Scientific, 1993, pp. 266–292.

[27] J. Hannan, Making abstract machines less abstract, in *Functional Programming Languages and Computer Architecture, 5th ACM Conference*, J. Hughes, Ed., *Lecture Notes in Computer Science*, Vol. 523, Springer-Verlag, Heidelberg, 1991, pp. 618–635.

[28] J. Hannan, Operational semantics-directed compilers and machine architectures, *ACM Trans. Programming Languages Syst.*, 16(4), 1215–1247, 1994.

[29] P. Henderson, *Functional Programming: Application and Implementation*, Prentice Hall International, Englewood Cliffs, NJ, 1980.

[30] M. Hennessy, *Algebraic Theory of Processes*, MIT Press, Cambridge, MA, 1988.

[31] S. Haridi, S. Janson and C. Palamidessi, Structural operational semantics of AKL, *Future Generation Comput. Syst.*, 1992.

[32] C.A.R. Hoare and P.E. Lauer, Consistent and complementary formal theories of the semantics of programming languages, *Acta Inf.*, 3, 135–153, 1974.

[33] C.A.R. Hoare, An axiomatic basis for computer programming, *Commun. ACM*, 12(10), 1969.

[34] C.A.R. Hoare, Communicating sequential processes, *Commun. ACM*, 21(8), 666–677, 1978.

[35] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice Hall International, Englewood Cliffs, NJ, 1985.

[36] J. Hannan and F. Pfenning, Compiler verification in Seventh Annual IEEE Symposium on Logic in Computer Science, IEEE, 1992, pp. 407–418.

[37] S.L.P. Jones, *The Implementation of Functional Programming Languages*, Prentice Hall International, London, 1987.

[38] G. Kahn, Natural semantics, in *Proceedings of STACS '87*, F.J. Brandenburg, G. Vidal-Naquet and M. Wirsing, Eds., *Lecture Notes in Computer Science*, Vol. 247, Springer-Verlag, Heidelberg, 1987, pp. 22–39.

[39] R. Kennaway and R. Sleep, Director strings as combinators, *ACM Trans. Programming Languages Syst.*, 10(4), pp. 602–626, 1988.

[40] P.J. Landin, The mechanical evaluation of expressions, *Comput. J.*, 6(5), 308–320, 1964.

[41] P.J. Landin, An abstract machine for designers of computing languages, in *Proc. IFIP Congr.*, 1965, pp. 438–439.

[42] P.J. Landin, A correspondence between ALGOL 60 and Church's lambda-notation, Part I, *Commun. ACM*, 8(2), pp. 89–101, 1965.

[43] L.P. Lauer, Formal Definition of Algol 60, Technical report TR.25.088, IBM Lab, Vienna, 1968.

[44] J.J. Leifer, Operational Congruences for Reactive Systems, Ph.D. thesis, University of Cambridge Computer Laboratory, 2001.

[45] J. McCarthy, Towards a Mathematical Science of Computation, in *Information Processing 1962*, C.M. Popplewell, Ed., 1963, pp. 21–28.

[46] D. Miller and J. Hannan, From operational semantics to abstract machines: preliminary results, in *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, ACM Press, New York, 1990.

[47] M. Miculan, The expressive power of structural operational semantics with explicit assumptions, in *Types for Proofs and Programs*, H. Barendregt and T. Nipkow, Eds., Springer-Verlag, New York, *Lecture Notes in Computer Science*, Vol. 806, 1994, pp. 263–290.

[48] R. Milner, Processes: A mathematical model of computing agents, in *Proceedings Logic Colloquium 1973*, H.E. Rose and J.C. Shepherdson, Eds., North-Holland, Amsterdam, 1973, pp. 158–173.

[49] R. Milner, Program semantics and mechanized proof, in *Foundations of Computer Science II*, K.R. Apt and J.W. de Bakker, Eds., Mathematical Centre, Amsterdam, 1976, pp. 3–44.

[50] R. Milner, *A Calculus of Communicating Systems, Lecture Notes in Computer Science*, Vol. 92, Springer-Verlag, New York, 1980.

[51] R. Milner, *Communication and Concurrency*, Prentice Hall International, Englewood Cliffs, NJ, 1989.

[52] J. Morris, A general axiom of assignment and linked data structure, in *Theoretical Foundations of Programming Methodology*, M. Broy and G. Schmidt, Eds., 1982, pp. 25–41.

[53] J. Morris, Algebraic Operational Semantics for Modula 2, Ph.D. thesis, University of Michigan, 1988.

[54] P.D. Mosses, *Action Semantics*, Vol. 26, *Cambridge Tracts in Theoretical Computer Science*, Cambridge University Press, London, 1992.

[55] D. Le Metayer and D. Schmidt, Structural operational semantics as a basis for static program analysis, *ACM Comput. Surv.*, 28, 340–343, 1996.

[56] R. Milner, M. Tofte, R. Harper and D. MacQueen, *The Definition of Standard ML* (Revised), MIT Press, Cambridge, MA, 1997.

[57] C.-H.L. Ong, Correspondence between operational and denotational semantics: the full abstraction problem for PCF, in *Handbook of Theoretical Computer Science*, S. Abramsky, Ed., Vol. 3, Oxford University Press, Oxford, 1999.

[58] J. Palsberg, A provably correct compiler generator, in *ESOP '92, 4th European Symposium on Programming, Proceedings*, B. Krieg-Bruckner, Ed., Vol. 582, Springer-Verlag, New York, 1992, pp. 418–434.

[59] D. Park, Concurrency and automata on infinite sequences, in *5th GI Conference*, P. Deussen, Ed., *Lecture Notes in Computer Science*, Vol. 104, Springer-Verlag, New York, 1981, pp. 167–183.

[60] PL/I Definition Group, Formal definition of PL/I version 1, report TR25.071, American National Standards Institute, 1986.

[61] G.D. Plotkin, Call-by-name, call-by-value and the lambda-calculus, *Theor. Comput. Sci.*, 1, 125–159, 1975.

[62] G.D. Plotkin, LCF considered as a programming language, *Theor. Comput. Sci.*, 5, 223–256, 1977.

[63] G.D. Plotkin, A Structural Approach to Operational Semantics, report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[64] G.D. Plotkin, An operational semantics for CSP, in *Proceedings IFIP TC2 Working Conference on Formal Description of Programming Concepts — II, Garmisch-Partenkirchen*, D. Bjørner, Ed., North-Holland, Amsterdam, 1983, pp. 199–225.

[65] D. Sands, From SOS Rules to Proof Principles: An Operational Metatheory for Functional Languages, in *Conference Record 24th ACM Symposium on Principles of Programming Languages, Paris, France*, 1997, pp. 428–441.

[66] D.A. Schmidt, *Denotational Semantics: A Methodology for Language Development*, Allyn & Bacon, 1986.

[67] P. Sewell, From rewrite rules to bisimulation congruences, in *Proceedings of CONCUR '98*, *Lecture Notes in Computer Science*, Vol. 1466, Springer-Verlag, New York, 1998, pp. 269–284.

[68] J. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, Cambridge, MA, 1977.

[69] R.D. Tennent, *Principles of Programming Languages*, Prentice Hall International, 1981.

[70] S. Tini, An axiomatic semantics for Esterel, *Theor. Comput. Sci.*, 269, 2001.

[71] D.A. Turner, A new implementation technique for applicative languages, *Software Pract. Exp.*, 9(1), pp. 31–49, 1979.

[72] D.H.D. Warren, An Abstract Prolog Instruction Set, Technical note 309, SRI International, Menlo Park, CA, 1983.

[73] D.A. Watt, *Programming Concepts and Paradigms*, Prentice Hall, New York, 1990.

[74] S. Weber, B. Bloom and G. Brown, Compiling Joy to Silicon: A Verified Silicon Compilation Scheme, in Proceedings of the Advanced Research in VLSI and VLSI and Parallel Systems Conference, Providence, RI, 1992.

[75] A. Wright and M. Felleisen, A syntactic approach to type soundness, *Inf. Computation*, 115(1), pp. 38–94, 1994.

[76] G. Winskel, *The Formal Semantics of Programming Languages: An Introduction*, Foundations of Computing Science, MIT Press, Cambridge, MA, 1993.

[77] M. Wand and D.P. Oliva, Proving the Correctness of Storage Representations, in *Proceedings of the 1992 ACM Conference on LISP and Functional Programming, New York*, 1992, pp. 151–160.

Index

Computer Science/Computer Engineering

The Compiler Design Handbook

Optimizations and Machine Code Generation

Internet security concerns and the widespread use of object-oriented languages are just the beginning. Add embedded systems, multiple memory banks, highly pipelined units operating in parallel, and a host of other advances and it becomes clear that current and future computer architectures pose immense challenges to compiler designers — challenges that already exceed the capabilities of traditional compilation techniques.

The Compiler Design Handbook: Optimizations and Machine Code Generation is designed to help you meet those challenges. Written by top researchers and designers from around the world, it presents detailed, up-to-date discussions on virtually all aspects of compiler optimizations and code generation. It covers a wide range of advanced topics, focusing on contemporary architectures such as VLIW, superscalar, multiprocessor and digital signal processing. It also includes detailed presentations that highlight the different techniques required for optimizing programs written in parallel and those written in object-oriented languages. Each chapter is self-contained, treats its topic in depth and includes a section on future research directions.

Compiler design has always been a highly specialized subject with a fine blend of intricate theory and difficult implementation. With its careful attention to the most researched, difficult and widely discussed topics in compiler design, ***The Compiler Design Handbook*** offers a unique opportunity for designers and researchers to update their knowledge, refine their skills and prepare for future innovations.

1240

ISBN 0-8493-1240-X

90000

9 780849 312403

CRC PRESS

www.crcpress.com