Ghost Process: a Sound Basis to Implement Process Duplication, Migration and Checkpoint/Restart in Linux Clusters

Geoffroy Vallée[1]
ORNL/INRIA/EDF
Computer Science and Mathematics Division
Oak Ridge National Laboratory
Oak Ridge, TN 37831, USA
valleegr@ornl.gov
Fax: 865-576-5491

Renaud Lottiaux David Margery
Christine Morin
IRISA/INRIA, PARIS project-team
Campus Universitaire de Beaulieu
35042 Rennes, Cedex, France
{rlottiau, dmargery, cmorin}@irisa.fr
Fax: +33 2 99 84 71 71

Jean-Yves Berthou
EDF R&D
1 avenue de Général de Gaulle
BP408, 92141 Clamart, France
jyberthou@edf.fr
Fax: +33 1 47 65 34 99

Abstract

Process management mechanisms (process duplication, migration and checkpoint/restart) are very useful for high performance and high availability in clustering systems.

The single system image approach aims at providing a global process management service with mechanisms for process checkpoint, process migration and process duplication. In this context, a common mechanism for process virtualization is highly desirable but traditional operating systems do not provide such a mecahnism.

This paper presents a kernel service for process virtualization called ghost process, extending the Linux kernel. The ghost process mechanism has been implemented in the Kerrighed single system image based on Linux.

Keywords: Linux cluster, distributed system, operating system, single system image, process virtualization.

1 Introduction

Today, clusters are more and more widely used to execute numerical applications. Mechanisms are needed to ease cluster use and to take advantage of the cluster's distributed resources. Process management mechanisms are very useful in this respect. A process duplication mechanism allows the deployment of processes of a parallel application on several cluster nodes. Such a mechanism extends for a cluster the traditional *fork* mechanism provided by the Linux system for process creation. However, to take advantage of the complete cluster resources during the execution of a given workload, it may not be sufficient to correctly place the processes on the cluster nodes when they are created. Dynamic load balancing strategies are advantageous to keep the load balanced in a cluster. Such strategies assume that an efficient process migration mechanism[8] is implemented to move a process from one node to another one. Finally, some cluster nodes may fail during the execution of an application. Fault tolerance mechanisms are needed to tolerate node failures during the execution of a long-running application. Checkpointing is a traditional fault tolerance technique well-suited to numerical applications. Checkpointing a process consists in periodically saving in stable storage the process state during failure-free execution. In the event of a failure, the process is restarted from its last checkpoint. All these mechanisms need a mechanism for process virtualization.

For the sake of efficiency and ease of use, a kernel level implementation of the mechanisms used for global process management in a cluster (process duplication, migration, checkpoint/restart) is highly desirable. However, the Linux

[1] The submitted manuscript has been authored by a contractor of the U.S. Government under Contract No. DE-AC05-00OR22725. Accordingly, the U.S. Government retains a non-exclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes

0-7695-2434-6/05 $20.00 © 2005 IEEE

kernel does not provide system services and interfaces for process virtualization.

In this paper, we present the *ghost process* mechanism for process virtualization based on the Linux kernel, which provides a system service and an API to virtualize processes. This mechanism eases the implementation of mechanisms for global process management. As example, we present the implementation of two mechanisms for process management at the cluster scale: a mechanism of process migration and a mechanism of process checkpoint/restart.

The *ghost process* mechanism has been implemented in the KERRIGHED Single System Image (SSI) cluster operating system[9, 10] based on Linux. It has been used to efficiently and easily implement process duplication, migration and checkpoint/restart in Kerrighed. Associated with global process identification mechanisms and with other global resource management mechanisms (*e.g.* global signal management) which are out of the scope of this paper, it allows global management of processes cluster wide.

The remainder of this paper is organized as follows: Section 2 presents related works on global process management mechanisms in clusters. Section 3 describes the *ghost process* concept we propose for process virtualization. In Section 4 we show how to simply implement process migration and checkpoint/restart relying on the ghost process concept. Section 5 provides a performance evaluation of the ghost process mechanism in the framework of its use for process checkpoint/restart. Finally, Section 6 concludes.

2 Background

A lot of cluster systems or toolkits allow application deployment. Tools like rsh or ssh allow users to manually deploy applications. Users individually choose a node for process creation so there is no global resource management. Other systems, like batch systems (*e.g.* PBS[5]), the BProc[6] system or programming environments (*e.g.* MPI, PVM) allow automatic and efficient deployment of applications. With such systems, users do no longer need to manually deploy processes.

Some systems, like Epckpt[11], condor[2] or BLCR[3] offer checkpoint/restart mechanisms. With all these systems, a checkpoint mechanism is implemented by the extraction of process information which is stored on a resource. Depending of the checkpoint mechanism, different ways to store processes can be implemented. Traditionally, resources used to store process checkpoint are (i) memory for efficiency or (ii) disks for storage stability. Using these images, a process can be restarted creating a clone process from a checkpoint.

Other systems allow to dynamically migrate processes between cluster nodes. A lot of academic studies have been made to implement process migration[8]. Two major approaches allow implementation of process migration. First, process migration can be implemented through a checkpoint/restart mechanism: the process is checkpointed on a shared file system (*e.g.* using NFS) and is restarted from the file system on a remote node. This is the approach of systems like Condor. The second approach is to migrate processes directly through the network for efficiency. In such a system, the process is extracted from a node, directly sent through the network to a remote node, and a new running process is created. System like OpenSSI[13] or Mosix[1] implement this approach.

Currently, no systems provide the complete set of mechanisms in an efficient way, excepted Genesis[4]. But Genesis is based on a specific micro-kernel and does not provide a complete Unix like interface.

3 The Ghost Process Concept for Process Virtualization

The Linux kernel provides some simple interfaces to manage processes on a single machine. Unfortunately, there is no interface to extract a process from the system to create an "image" of the process which is independent of the local machine.

To have a convenient process virtualization, we need a service to extract any process from the system and a simple interface to manage extracted processes. The different mechanisms of process management should not be related to the hardware access associated. For example, for process migration, the process is extracted and then transfered through the network on a remote node, whereas for process checkpoint, the extracted process is saved on a stable device. Therefore, the interface of the process virtualization services should integrate the definition of the resource used to manipulate processes. This definition is very important to guarantee a simple implementation of new mechanisms for global process management. The capability is also interesting to develop mechanism of global process management adapted to specific hardware. For example, with a simple definition of resource access, process migration can be ported on a new network technology that uses a specific programming interface.

Section 3.1 presents the service of process virtualization, and Section 3.2 presents interfaces to virtualize processes and to define resources access to use with the virtualization mechanism.

3.1 Implementation of Process Virtualization

In a system, a process is represented by two kinds of information: process information that is available in the kernel (*e.g.* open files, memory segments, pending signal, pro-

Proceedings of the 4th International Symposium on Parallel and Distributed Computing (ISPDC'05)
0-7695-2434-6/05 $20.00 © 2005 IEEE

cess identifier) and the register values associated to the process execution.

With this data, it is possible to execute a process anywhere in a cluster. The ghost process mechanism allows the extraction of all this information. The extraction of the process's context and address space is not an issue if kernel information is accessible (if the extraction mechanism can access the operating system data structures).

The extraction of the register values is more difficult. With most processors and modern systems, these values are only available in particular system states. For example, in Linux using x86 processors, register values are only available in the kernel during return of exception, system calls or interruptions. Therefore, to create a ghost process in the Linux kernel, the system has to be in a state where register values are available. It is should also be a state that enables the activation of a process extraction at anytime, even if the process is not active in the system (not running on a processor). Such a state is very similar to the signal treatment state. Therefore, we have created in the Linux kernel a new state similar to the kernel signal state (see Figure 1). This state is accessible after the treatment of pending signals, so less information (*a priori* no signals) needs to be managed. The creation of this state requires a kernel modification of about ten lines. This modification allows to mark a process as waiting for an extraction (in a way similar to the way a process is marked as having a pending signal). This new state is active during the return of exceptions, system calls or interruptions. Therefore, register values are available and process extraction can be activated at any time.

3.2 Process Virtualization Interfaces

3.2.1 Virtualization Interfaces

To ease the use of process virtualization, an API is available to deal with ghost processes. This API is based on the concepts of *process exportation* and *process importation*. The *process exportation* (see Listing 1) allows to virtualize a process from the local system and provides an object representing the ghost process. This object has the complete set of information to manage a process independently to its location.

Listing 1. Ghost Process API

```
/* Export a process */
int export_process(ghost, process);
/* Import a process */
int import_process(ghost);
/* Plug a resource object to a ghost process */
int plug_resource_object(class_id, ghost);
```

The *process importation* (see Listing 1) allows to transfer the ghost process in the local memory (if the ghost process is not locally available) and to create a clone process from a ghost process.

The system programmer needs to give a ghost process object provided by an exportation and a new process is created (forking) and inserted in the local system (the process is active and can run in the system). Therefore, after the process importation a new process is ready to run on the local node, independently to the source node.

3.2.2 Interfaces for Resource Access Definition

The mechanisms for ghost process management are always associated to a resource object representing the device used to back the ghost process. For example, process migration is a combination of a resource object representing the network and the generic ghost management mechanisms whereas process checkpoint/restart is the result of the combination of a resource object associated to a stable storage device (*e.g.* hard drive) with those mechanisms. Therefore, an interface is needed to specify the resource object used with a ghost process. This specification defines a new mechanism available for global process management, but is also used to adapt the mechanisms to new network technologies which provides their own network interface. We say a resource access object is plugged in the mechanism of process virtualization.

The interface between resource access objects and the mechanism of process virtualization are based on read/write methods. The write method is automatically called during the process exportation, whereas the read method is called during the process importation.

Some resource access objects need extra parameters. For example, to access to the network, we need to know the remote node address before sending the data. These parameters can be specified during the initialization step of the resource access object. For example, we can specify a memory buffer if memory access is plugged in a ghost process. Then, the ghost process importation reads ghost process information from the memory. A finalization step is also implemented for resource access objects that need finalization (like files).

The current implementation allows to read from and write to memory, read from or write to disk and to receive from and send to the network. Each resource access object type is identified by a unique identifier cluster wide. To plug such a resource object in a ghost process, a simple interface is available for system programmers (see Listing 1). To set a unique identifier to each resource access method, the system programmer must update an internal function of the ghost process mechanism. This function also set the function to initialize/finalize the resource access method, to read from/write in the resource.

A set of interface has been developed for file access,

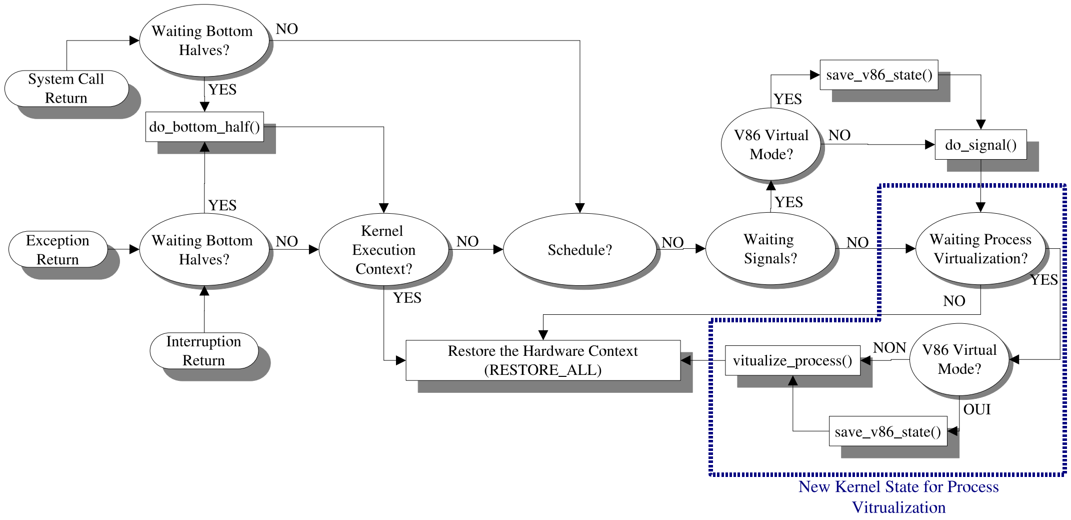

Figure 1. Kernel state for process virtualization

memory access and network access.

Memory Access The interface to manage buffers in memory is shown in Listing 2.

Listing 2. Kerrighed API for memory access

```
/* Initialization function */
buff_t create_new_buffer(buffer_size);
/* Read function */
int buff_read(destination, buffer, size);
/* Write function */
int buff_write(buffer, source, size);
/* Function to plug the memory access method
in an instance of a ghost process */
int associate_buffer_to_ghost(buffer, ghost);
```

This interface is integrated in the ghost process management and it is possible to associate a buffer to a ghost using the interface shown in Listing 2.

File Access KERRIGHED also provides an interface to manage file read/write operations (see Listing 3).

Listing 3. Kerrighed API for file access

```
/* Initialization function */
file* file_open(pathname);
/* Read function */
int file_read(file, destination, size);
/* Write function */
int file_write(file, source, size);
/* Finalization function */
int file_close(file);
/* Function to plug the file access method
in an instance of a ghost process */
int associate_file_to_ghost(file, ghost);
```

This interface is integrated in the ghost process management and it is possible to associate a file to a ghost using the interface shown in Listing 3.

Network Access KERRIGHED also provides an interface to send/receive data through the network (see Listing 4).

Listing 4. Kerrighed API for network access

```
/* Read function */
int network_read(remote_node_id, variable, size);
/* Write function */
int network_write(remote_node_id, variable, size);
/* Function to plug the network access method
in an instance of a ghost process */
int associate_network_to_ghost(remote_node_id,
channel, port, ghost);
```

This interface is integrated in the ghost process management and it is possible to associate a file to a ghost using the interface shown in Listing 4.

3.3 Summary

The ghost process mechanism is a mechanism for process virtualization which allows the implementation, in an easy way, of mechanisms for global process management such as process duplication, migration and checkpoint/restart.

To illustrate the use of the ghost process mechanism, the next section details the implementation of two global process management mechanisms: (i) process migration and (ii) process checkpoint/restart.

4 Using Ghost Processes to Create Mechanisms for Global Process Management

The API described in Section 3.2 is used to create new mechanisms to globally manage processes. For each mechanism (e.g. process migration or duplication), after the creation of a ghost process, system programmers have to manage the process from which the ghost process has been created and to manage the new active process created from the ghost process. We illustrate this approach on the examples of process migration in Section 4.1 and of process checkpoint/restart in Section 4.2.

4.1 Process Migration

For example, Listing 5 shows a simple approach to implement a process migration mechanism is to extract the process in a ghost process in the memory of the source node, then to send the ghost process through the network to a remote destination node. On the destination node, the ghost process needs to be received and then, the ghost process needs to be imported to create a new active process. When the new process is created on the destination node, an acknowledgement is sent to the source node. When the acknowledgement is received on the source node, the initial process is destroyed.

Listing 5. Process migration

```
/* Process migration: algorithm executed on
the source node */
void migrate_process (pid)
{
  /* find a process to migrate */
  process = find_task_by_pid(pid);
  /* create a new instance of ghost process */
  ghost = create_new_ghost();
  /* plug the ghost process into the memory
  access interface */
  plug_resource_interface(MEMORY_WRITE, ghost);
  /* export the process, the ghost process
  is created in memory */
  export_process(ghost, process);
  /* send the ghost process to the remote node */
  send_ghost_process(remote_node_id, ghost);
  /* wait for the acknowledgement sent by the
  remote node after the remote process creation */
  distant_pid = wait_ack(remote_node_id);
  /* when the remote process is created, we */
  /* destroy the process on the source  node */
  destroy_process(process);

/* Process migration: algorithm executed on
the destination node */
void receive_migrated_process (pid)
{
  /* receive the ghost process from the
  source node the ghost process is saved in
  a buffer */
  memory_buffer = receive_ghost(original_node_id);
  /* create a new instance of a ghost process */
  ghost = create_new_ghost();
  /* associate the memory buffer with the ghost
  process */
  associate_buffer_to_ghost(buffer, ghost);
  /* plug the ghost process into the  memory in
  read mode interface */
  plug_resource_interface(MEMORY_READ, ghost);
  /* the ghost process is loaded, a new process is
  created */
  new_local_pid = import_process(ghost, process);
  /* if the process is successfully created, an */
  /* acknowledgement is sent to the source node */
  if (process_creation_succeed) then
    send_ack(original_node_id, new_local_pid);
}
```

4.2 Process Checkpoint/Restart

A process checkpoint mechanism is quite simple. A process image needs to be extracted and stored in a stable storage device. The process restart mechanism is also simple. An active process is created from the process image saved in stable storage device. In KERRIGHED, checkpoints can be saved either in memory or on disk.

This section details the implementation of the process checkpoint/restart mechanism based on the ghost process mechanism. We present in Section 4.3 the implementation of a process checkpoint/restart mechanism in which checkpoints are saved in the local memory of the checkpointed process execution node. We present in Section 4.4 the implementation of a process checkpoint/restart mechanism in which checkpoints are saved in the local disk of the checkpointed process execution node.

4.3 Implementation of a Memory Checkpoint/Restart Mechanism

The algorithm to create a memory checkpoint is shown in Listing 6.

Listing 6. Memory checkpoint/restart

```
buff_t checkpoint_process_in_memory
       (process_identifier)
{
  buffer = create_new_buffer();
  ghost = create_new_ghost();
  /* associate memory access method to the
  ghost process */
  plug_resource_interface(MEMORY_WRITE, ghost);
  associate_buffer_to_ghost(buffer, ghost)
  process = find_task_by_pid(process_identifier);
  export_process(ghost, process);
  return buffer;
}

void memory_restart (memory_checkpoint_id)
{
  if (process_state == running) then
```

```
    destroy_process();
  ghost = create_new_ghost();
  /* associate memory access to ghost process
  mechanism */
  plug_resource_interface(MEMORY_READ, ghost);
  buffer = find_memory_checkpoint
  (memory_checkpoint_id);
  associate_buffer_to_ghost(buffer, ghost);
  import_process(ghost);
}
```

We have seen that the ghost process mechanism is composed of two parts: the definition of the ghost process data and the resource access method. For a memory checkpoint, the ghost process mechanism has to use the local memory, using a resource access method defined in KERRIGHED (see Figure 2). The memory access method is based on the KERRIGHED memory access interface presented in Listing 2.

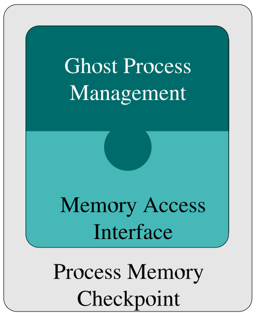

Figure 2: Process checkpoint on memory based on ghost process

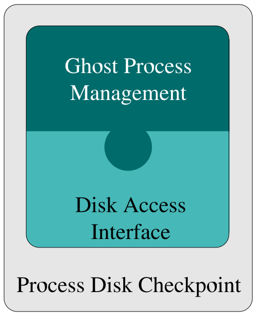

Figure 3: Process checkpoint on disk based on ghost process

The restart mechanism is quite simple. The original process (if it is still running) needs to be stopped. Then, a new process is created from a memory checkpoint (see Listing 6). To restart a process from a memory checkpoint, the ghost process mechanism needs to be associated to a method to access the local memory.

4.4 Implementation of a Disk Checkpoint/Restart Mechanism

The algorithm used to create a disk checkpoint is shown in Listing 7.

Listing 7. Disk checkpoint/restart

```
ghost_t disk_checkpoint (process_identifier)
{
  file = file_open(pathname);
  /* associate disk access method to the ghost
  process */
  plug_resource_interface(FILE_WRITE, ghost);
  associate_file_to_ghost(file, ghost)
  process = find_task_by_pid(process_identifier);
  export_process(ghost, process);
  /* associate the disk access method to */
  /* the ghost process mechanism */
  flush(file);
  return ghost;
}

void disk_restart (disk_checkpt_id)
{
  if (process_state == running) then
    destroy_process;
  ghost = create_new_ghost();
  /* associate disk access to ghost process
  mechanism */
  plug_resource_interface(FILE_READ, ghost);
  file = find_file_checkpoint(disk_checkpt_id);
  associate_buffer_to_ghost(file, ghost);
  import_process(ghost);
}
```

Matrix size	Ghost size (KBytes)
500x500	4 229
750x750	9 354
1000x1000	12 429
1250x1250	20 629
1500x1500	24 729
1750x1750	28 833
2000x2000	32 933

Table 1. Ghost process size for the MGS application according to the matrix size

For a disk checkpoint, the ghost process mechanism needs to use the local file system, using a resource access method defined in KERRIGHED (see Figure 3). The disk access method is based on the KERRIGHED disk access interface presented in Listing 3.

The restart mechanism is quite simple and very similar to the memory restart. The original process (if it is still running) needs to be stopped. A new process is created from a disk checkpoint (see Listing 7). To restart a process from a disk checkpoint, the ghost process mechanism needs to be associated to a method to access to local file system.

5 Evaluation

In this section, we present the results of an experimental evaluation of the ghost process mechanism through its use in the process checkpoint/restart mechanism. The evaluations have been done using the implementation of these mechanisms in the Kerrighed cluster operating system based on Linux 2.4.24. Our results are for a cluster in which each node is a PC containing a 1 GHz PIII processor, a 512 MB RAM, a 100Mps Ethernet NIC and a local hard disk. All the evaluations have been performed with a process executing a sequential version of the Modified Gram-Schmidt (*mgs*) application. *mgs* produces from a set of vectors an orthonormal basis of the space generated by these vectors.

Table 1 shows the size of the ghost process for a range of matrix sizes. Typically, the ghost process's size is proportional to the size of the application memory space. The bigger the memory space is, the bigger the ghost process is. For the *mgs* application, only the memory size impacts on the ghost size because the other process information has a constant size (the application always accesses the same files for example).

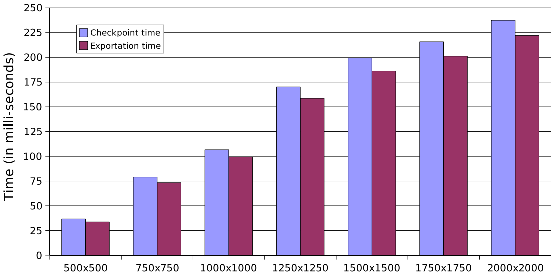

Figure 4. Cost of the creation of a memory process checkpoint

We evaluated the cost of the process checkpoint/restart mechanism, regarding the cost of the exportation/importation mechanism. So we have computed the total time for creating a checkpoint (*i.e.* the time between the beginning of the process checkpoint and the end of memory write of the ghost process) and the exportation time (ghost process extraction time). Note that the exportation time is a part of the total checkpoint creation time.

5.1 Evaluation of the Checkpoint/Restart Mechanism Using the Local Memory

Figure 4 shows the time to checkpoint a process in local memory, presenting both the total checkpoint creation time and the ghost process exporation time. The ghost process exportation (and so memory accesses, data being saved in memory during the process extraction) time is a major part of the total checkpoint time. So, the efficiency of the checkpoint mechanism, using the local memory, depends on the efficiency of the exportation mechanism.

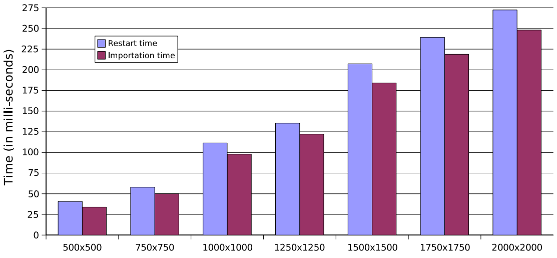

Figure 5. Restart cost from a memory checkpoint

Figure 5 shows the time needed to restart a process from a memory checkpoint. The cost of a process restart from a memory checkpoint is quite similar to the cost of process checkpoint in memory. The efficiency of the restart mechanism depends on the efficiency of the ghost process mechanism. The ghost process importation time is the major part of the restart time.

5.2 Evaluation of the Checkpoint/Restart Mechanism Using the Local Disk

The file access API of Kerrighed uses the local file system. When a checkpoint file is created to store a ghost process, this file is also stored in the local system file cache (with a flush of all file buffers to be sure that data is physically stored on disk). Moreover, to be sure that no cache effects improve artificially performances, all restart are done after a machine reboot.

Figure 6 shows the time needed to checkpoint a process on the local disk for different matrix sizes. As for memory checkpoints, the file access time (through the exportation phase) is a major part of the total checkpoint time.

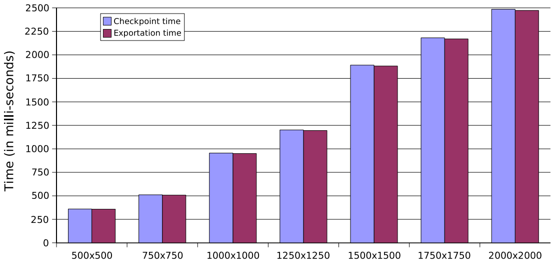

Figure 6. Cost of a process disk checkpoint

Figure 7 shows the cost for restarting a process from a checkpoint stored on disk assuming that the checkpoint file is not cached. The importation time is a major part of the restart time. The restart cost is due to the disk accesses. We observed that in our configuration the disk checkpoint cost is about 10 times higher than the cost of a memory checkpoint.

5.3 Summary

The performance evaluation shows that for both checkpoint/restart in memory and on disk, performance depends directly on the performance of the ghost process management. So, improving the efficiency of the ghost process

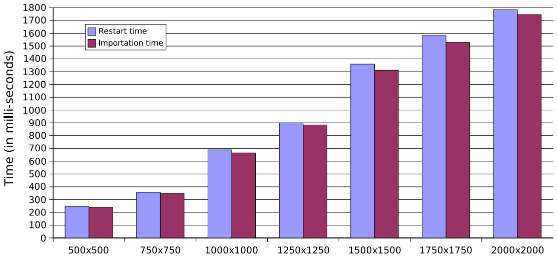

Figure 7. Restart cost from a disk checkpoint

mechanism, and in particular the efficiency of resource accesses, the efficiency of mechanisms for global process management will be improved.

6 Conclusion

In this paper, we have presented the ghost process mechanism for process virtualization and how it can be used to ease the implementation of various mechanisms for global process management. The ghost process mechanism provides a full system service for process virtualization (thanks to the exportation/importation mechanism) and a set of interfaces to plug in ghost process instances various methods to access resources. Thanks to ghost processes, traditional mechanisms of process management can be extended at the cluster scale. This approach guarantees the ease of maintenance and development of new mechanisms for global process management. The second benefit is that improving the efficiency of the ghost process mechanism, all global process management mechanisms become more efficient.

The ghost process mechanism has been implemented in the KERRIGHED SSI cluster operating system. The KERRIGHED's global process scheduler uses the process duplication, migration and checkpoint restart mechanisms based on the ghost processes[12].

Kerrighed provides different distributed services that are in charge of global resource management. Combining global memory management, ghost processes (used to implement process duplication) and a cluster wide process synchronization service, a complete support of POSIX threads has been implemented[7] in Kerrighed.

Systems that provides some mechanisms for process virtualization (*e.g.* process migration, process checkpoint/restart) at the kernel level are all based on a specific kernel patch. Nevertheless, all these patches are similar. Therefore, a common patch and the integration of this patch in the kernel should be very usefull and should simplify the development of such a virtualization mechanism for clustering systems. The ghost process mechanism can easily be ported on such a common patch.

References

[1] A. Barak and O. La'adan. The MOSIX multicomputer operating system for high performance cluster computing. *Future Generation Computer Systems*, 13(4–5):361–372, 1998.

[2] J. Basney and M. Livny. Deploying a high throughput computing cluster. In R. Buyya, editor, *High Performance Cluster Computing: Architectures and Systems, Volume 1*. Prentice Hall PTR, 1999.

[3] J. Duell, P. Hargrove, and E. Roman. The design and implementation of berkeley lab's linux checkpoint/restart. Technical report, Berkeley Lab, 2003.

[4] A. Goscinski, M. Hobbs, and J. Silcock. Genesis: an efficient, transparent and easy to use cluster operating system. *Parallel Comput.*, 28(4):557–606, 2002.

[5] Henderson and L. Robert. Job scheduling under the portable batch system. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 279–294. Springer-Verlag, 1995. Lecture Notes in Computer Science vol. 949.

[6] E. Hendriks. BProc: the Beowulf distributed process space. In *Institute for Crustal Studies (ICS) 2002*, pages 129–136, New York City, USA, June 2002.

[7] D. Margery, G. Vallée, R. Lottiaux, C. Morin, and J.-Y. Berthou. Kerrighed: a SSI cluster OS running OpenMP. In *Proc. 5th European Workshop on OpenMP (EWOMP '03)*, Sept. 2003.

[8] D. S. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou. Process migration. *ACM Computing Surveys (CSUR)*, 32(3):241–299, 2000.

[9] C. Morin, P. Gallard, R. Lottiaux, and G. Vallée. Towards an efficient Single System Image cluster operating system. *Future Generation Computer Systems*, 20(2), Jan. 2004.

[10] C. Morin, R. Lottiaux, G. Vallée, P. Gallard, G. Utard, R. Badrinath, and L. Rilling. Kerrighed: a single system image cluster operating system for high performance computing. In *Proc. of Europar 2003: Parallel Processing*, volume 2790 of *Lect. Notes in Comp. Science*, pages 1291–1294. Springer Verlag, Aug. 2003.

[11] E. Pinheiro. Truly-transparent checkpointing of parallel applications.

[12] G. Vallée, C. Morin, J.-Y. Berthou, and L. Rilling. A new approach to configurable dynamic scheduling in clusters based on single system image technologies. In *Industrial Track of the International Parallel an Distributed Processing Symposium*, Nice, France, Apr. 2003.

[13] B. J. Walker. Open single system image (openssi) linux cluster project. Technical report, Hewlett-Packard, 2000.