INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

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

Geoffroy Vallée , Renaud Lottiaux , David Margery , Christine Morin , Jean-Yves Berthou

N°5510

Mars 2005

Systèmes communicants

ISSN 0249-6399 ISRN INRIA/RR--5510--FR+ENG

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

Geoffroy Vallée , Renaud Lottiaux , David Margery , Christine Morin , Jean-Yves Berthou

Systèmes communicants
Projet PARIS

Rapport de recherche n ° 5510 — Mars 2005 — 16 pages

Abstract: Today, clusters are widely used to execute numerical applications. Mechanisms are needed to ease cluster use and to take advantage of the cluster distributed resources. Process management mechanisms are very useful in this respect. Process duplication is needed for parallel application deployment, dynamic load balancing relies on process migration and process checkpoint/restart is needed to tolerate node failures during the execution of long-running applications. A kernel level approach allows to efficiently implement all these mechanisms. However, they are very complex to implement and to maintain. Nevertheless they are all based on a common concept: process virtualization which provides a mean to extract the state of a process from the operating system executed on each cluster node. This paper presents a process virtualization mechanism called *ghost process*, implemented at kernel level which can be used by system programmers to easily implement various process management mechanisms. The ghost process mechanism has been implemented in Kerrighed single system image cluster operating system based on Linux. It has been used to efficiently and easily implement process duplication, migration and checkpoint/restart in Kerrighed.

Key-words: cluster, distributed system, operating system, single system image, process migration, checkpointing, process virtualization, Linux

(Résumé : tsvp)

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Process fantôme: mécanisme de base pour la mise en œuvre de mécanismes de duplication, migration et de création/reprise de points de reprise de processus

Résumé : Aujourd'hui, les grappes de calculateurs sont largement utilisées pour l'exécution d'applications numériques. Des mécanismes sont nécessaires pour simplifier l'utilisation des grappes de calculateurs et pour tirer profit des ressources distribuées de la grappe. Les mécanismes de gestion des processus sont très utiles dans ce cadre. La duplication de processus est nécessaire pour le déploiement des applications parallèles, l'équilibrage dynamique de la charge de la grappe s'appuie sur un mécanisme de migration de processus et la création/reprise de points de reprise est nécessaire pour tolérer des défaillances de nœuds durant l'exécution d'application au temps d'exécution important. Une approche noyau permet de mettre en œuvre efficacement tous ces mécanismes. Néanmoins, ils sont très difficiles à mettre en œuvre et à maintenir. De plus, ils sont tous fondés sur un concept commun : la virtualisation de processus qui permet d'extraire l'état d'un processus du système d'exploitation exécuté sur chaque nœuds. Ce papier présente un mécanisme de virtualisation de processus appelé *processus fantôme*, mis en œuvre au sein du noyau et qui peut être utilisé par les programmeurs système pour simplement mettre en œuvre divers mécanismes de gestion de processus. Le mécanisme de processus fantôme a été mis en œuvre au sein du système d'exploitation pour un système à image unique pour grappe de calculateurs Kerrighed, fondé sur Linux. Il a été simplement et efficacement utilisé pour mettre en œuvre des mécanismes de duplication, de migration et de création/reprise de points de reprise de processus au sein de Kerrighed.

Mots-clé : grappe de calculateurs, système distribué, système d'exploitation, système à image unique, migration de processus, création de points de reprise, virtualisation de processus, Linux

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 distributed resources. Process management mechanisms are very useful in this respect. A process duplication mechanism allows to deploy the 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 whole 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 [7] is implemented to move a process from one node to another one. Finally, some cluster nodes may fail during the execution of a application. Fault tolerance mechanisms are needed to tolerate node failures during the execution of a long-running applications. 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.

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 implementation of such mechanisms at kernel level is complex and their maintenance difficult. Nevertheless, all these mechanisms rely on a common concept: process virtualization which provides a mean to extract the state of a process from the operating system executed on each cluster node.

In this paper, we present the *ghost process* mechanism for process virtualization and we show how it eases the implementation of various global process management mechanisms such as process duplication, migration and checkpoint/restart in a cluster.

The *ghost process* mechanism has been implemented in the KERRIGHED Single System Image (SSI) cluster operating system [8, 9] 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 the scope of this paper, it allows to globally manage 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]) or programming environment (*e.g.* MPI, PVM) allow to deploy automatically and efficiently applications. With such systems, traditionally based on mechanisms like rsh or ssh, with a placement policy, users do not need any more to manually deploy processes.

Some systems, like Epckpt[10], condor[2], crak[12] or BKCR[3] offer checkpoint/restart mechanisms. With all these systems, a checkpoint is implemented by the extraction of process information which is stored on a resource, but implement different way to store processes in order to improve performance. 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 process on cluster nodes. A lot of academic studies has made to implement process migration[7]. Two major approach allow to implement 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 on a remote node, restarting the process from the disk. This is the approach of systems like Condor. The second approach is to migrate processes directly through the network for efficiency. In a such 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 Mosix[1] implement this approach.

Currently, no system provide the complete set of mechanisms in an efficient way, accepted 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 main difficulty to implement global process management mechanisms is to extract from the system, during the execution of a process, the data representing its current state. With a process extraction mechanism allowing to virtualize a process and to create a clone process anywhere in the cluster, the implementation of mechanisms such as process migration or process checkpoint/restart becomes much easier. For example, for process migration, the system first extracts a process and then sends the process's image through the network. For the mechanism of process checkpoint, the system first extracts a process and then saves the process's image in stable storage.

Process virtualization is composed of two parts: a part to define the set of process's information to extract to be able to manage the process independently from its location, and a part to define the resource used in conjunction with the process extraction mechanism. The core of the ghost process mechanism defines the set of system information needed to

virtualize the process, and a resource access method is plugged into this set of information (see Figure 1).

Figure 1: Ghost process architecture

The resource access method plugged into the ghost process allows to specify what to do with extracted data. If the ghost process is plugged into a *write in memory* interface, all extracted data will automatically be saved in memory during the process extraction. The different resource access interfaces which can be plugged into a ghost process are defined in the ghost process mechanism. The current implementation allows to read from and write to memory, read from or write to disk and to send to and receive from the network. Each resource access interface is identified by a unique identifier cluster wide. To plug such an interface into a ghost process, a simple interface is available for system programmers (see Listing 1).

Listing 1: API to plug a resource access interface into a ghost process

```
int plug_resource_interface (api_id, ghost);
```

We can also define some parameters for each resource access interface. For example, it is possible to define a memory buffer if a ghost process is plugged into a *memory in read mode* interface, to be able to import a process from a ghost process stored in memory. The interface to manage buffers in memory is shown in Listing 2.

Listing 2: Kerrighed API for memory access

```
buff_t create_new_buffer (buffer_size) ;
int buff_read (destination, buffer, size ) ;
int buff_write (buffer, source, size ) ;
```

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 3.

Listing 3: API to associate a buffer to a ghost process

```
int associate_buffer_to_ghost (buffer, ghost ) ;
```

KERRIGHED also provides an interface to manage file read/write operations (see Listing 4).

Listing 4: Kerrighed API for file access

```
file * file_open (pathname) ;
int file_close ( file ) ;
int file_read ( file , destination , size ) ;
int file_write ( file , source, size );
```

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 5.

Listing 5: API to associate a file to a ghost process

```
int associate_file_to_ghost (file , ghost ) ;
```

The interface to specify parameters for a given resource access method is resource specific and needs to be defined during the implementation of the resource interface in the ghost process mechanism. To simplify the programing of new process management mechanisms and the use of the ghost process mechanism, resource access interfaces available with the ghost process mechanism allow to specify parameters through interfaces. With these interfaces, system programmers can use the traditional resource access methods for memory, disk and network.

System programmers do not have *a priori* to implement such interfaces to create new global process management mechanisms. We do not detail interfaces to access resources nor to specify parameters in this paper.

3.1 Creation of a Ghost Process

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, process 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 in which register values are available. It is should also be a state which 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. 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 System API for the Implementation of Global Process Management Mechanisms

To ease the implementation of global process management mechanisms by system programmers, 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 6) allows to virtualize a process from the local system and provides an object representing the ghost process.

Listing 6: Process exportation API

```
int export_process (ghost, process) ;
```

The *process importation* (see Listing 7) 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.

Listing 7: Process importation API

```
int import_process (ghost);
```

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).

3.3 Summary

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

4 Using Ghost Processes to Create Global Process Management Mechanisms

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, 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 (see Listing 8), then to send the ghost process through the network to a remote destination node (see Listing 9). 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 8: Process migration: algorithm executed on the source node

```
void migrate_process (pid)
{
        /* we find a process to migrate */
        process = find_task_by_pid (pid) ;
        /* we create a new instance of ghost process */
        ghost = create_new_ghost () ;
        /* we plug the ghost process into  the memory access interface
        */
        plug_resource_interface (MEMORY_WRITE, ghost) ;
        /* we export the process, the ghost process is created */
        /* in memory */
        export_process (ghost, process) ;
        /* we send the ghost process to the destination node */
        send_ghost_process (remote_node_id, ghost) ;
        /* we wait for the acknowledgement sent by the destination 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) ;
}
```

Listing 9: Process migration: algorithm executed on the destination node

```
void receive_migrated_process (pid)
{
        /* we receive the ghost process from the source node */
        /* the ghost process is saved in a buffer */
        memory_buffer = receive_ghost (original_node_id) ;
        /* we create a new instance of a ghost process */
        ghost = create_new_ghost () ;
        /* we associate the memory buffer with the ghost process */
        associate_buffer_to_ghost (buffer, ghost) ;
        /* we 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

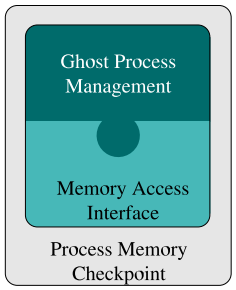

Figure 2: Process checkpoint stored in memory

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 10.

Listing 10: Creation of a memory process checkpoint

```
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 ;
}
```

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 and in Listing 3.

Figure 3: Process checkpoint stored in disk

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 Algorithm 11).

Listing 11: Restarting a process from a memory checkpoint

```
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) ;
}
```

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 12.

Listing 12: Creation of a disk process checkpoint

```
ghost_t disk_checkpoint (process_identifier)
{
        file = file_open (pathname) ;
        /* associate disk access method to the ghost process */
```

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

```
        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;
}
```

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 4 and in Listing 5.

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 Algorithm 13).

Listing 13: Restarting a process from a disk checkpoint

```
void memory_restart (disk_checkpoint_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_checkpoint_id);
        associate_buffer_to_ghost (file, ghost ) ;
        import_process (ghost) ;
}
```

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).

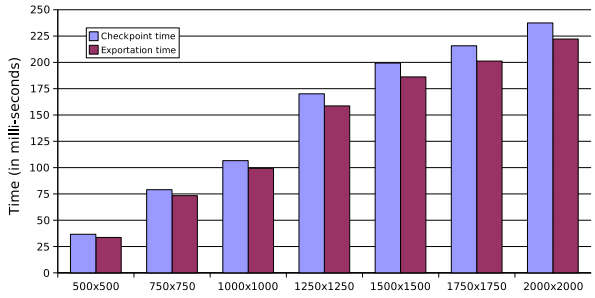

Figure 4: Cost of the creation of a memory process checkpoint

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.

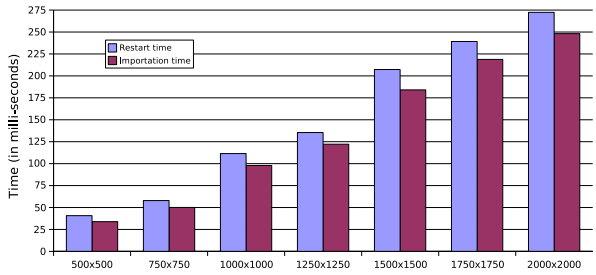

Figure 5: Cost of a process restart from a memory checkpoint

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

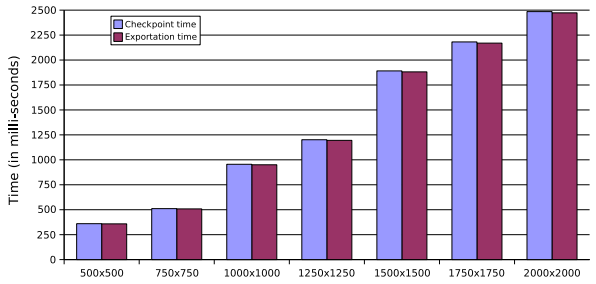

Figure 6: Cost of a Process Disk Checkpoint

phase) is a major part of the total checkpoint time.

Figure 7 shows the cost for restarting a process from a checkpoint stored on disk assuming that the checkpoint file is not cached. The exportation time is a major part of the checkpoint time. The checkpoint 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.

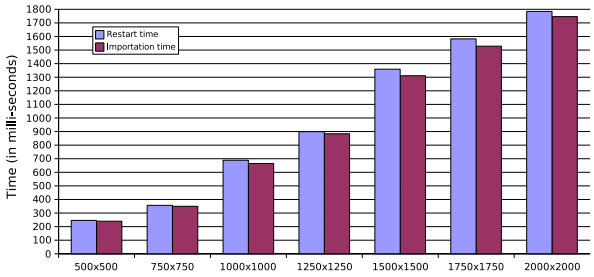

Figure 7: Cost of a Process Restart from a Disk Checkpoint

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 mechanism, and in particular the efficiency of resource accesses, the efficiency of the global process management mechanisms 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 global process management mechanisms process, taking the examples of process migration and process checkpoint/restart. The ghost process mechanism provides an exportation/importation interface and a set of interfaces (to access various resources) to be plugged into ghost process instances. An advantage of our approach is that improving the efficiency of the ghost process mechanism will benefit to several all global process management mechanisms.

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[11].

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 [6] in Kerrighed.

The ghost process mechanism cannot currently extract memory pages shared at the cluster scale. Ongoing works extend the ghost process mechanism to be able to implement checkpoint/restart mechanisms for shared memory applications.

Moreover, for process checkpoint/restart, performances are limited by the ghost process mechanisms but for other mechanisms of global process management, the method to access resources is the limitation. Some ongoing works implements efficient method of resource access; for example methods adapted to process migration are studied.

References

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

[2] Jim Basney and Miron Livny. Deploying a high throughput computing cluster. In Rajkumar 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 Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 279–294. Springer-Verlag, 1995. Lecture Notes in Computer Science vol. 949.

[6] David Margery, Geoffroy Vallée, Renaud Lottiaux, Christine Morin, and Jean-Yves Berthou. Kerrighed: a SSI cluster OS running OpenMP. In *Proc. 5th European Workshop on OpenMP (EWOMP '03)*, September 2003.

[7] Dejan S. Milojicic, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. Process migration. *ACM Computing Surveys (CSUR)*, 32(3):241–299, 2000.

[8] Christine Morin, Pascal Gallard, Renaud Lottiaux, and Geoffroy Vallée. Towards an efficient Single System Image cluster operating system. *Future Generation Computer Systems*, 20(2), January 2004.

[9] Christine Morin, Renaud Lottiaux, Geoffroy Vallée, Pascal Gallard, Gaël Utard, Ramamurthy Badrinath, and Louis 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, August 2003.

[10] Eduardo Pinheiro. Truly-transparent checkpointing of parallel applications.

[11] Geoffroy Vallée, Christine Morin, Jean-Yves Berthou, and Louis 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, April 2003.

[12] Hua Zhong and Jason Nieh. Crak: Linux checkpoint/restart as a kernel module. Technical Report CUCS-014-01, Department of Computer Science, Columbia University, November 2001.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399