

# Enhancing Reliability in Grid Systems with Virtual Machine Checkpointing Mechanism

Kasidit Chanchio<sup>1</sup> Hong Ong<sup>2</sup> ChokChai Leangsuksun<sup>3</sup> Veerapong Ratanasamoot<sup>1</sup>

<sup>1</sup> Department of Computer Science, Thammasat University Rangsit Campus,  
Klong Luang, Phatumtani, Thailand 12121

<sup>2</sup> Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA

<sup>3</sup> Louisiana Tech University, Ruston, LA 71272, USA

e-mail : <sup>1</sup>kasiditchanchio@gmail.com, <sup>2</sup>[hongong@ornl.gov](mailto:hongong@ornl.gov), <sup>3</sup>box@latech.edu

## Abstract

Fault tolerant is crucial for long-running Grid applications. This paper presents a work in progress on the development of a novel Hypervisor-based checkpointing mechanism that utilizes multi-core computers to reduce checkpointing delay of a virtual machine. Our preliminary experiments indicate high potentials of our mechanism and can reduce the delay substantially.

## 1 Introduction

Over the last decade, Grid technology [1] has emerged as an important tool for solving large-scale scientific problems both in academia and industry. However, due to the diverse failures and error conditions in the grid environments, developing, deploying, and executing scientific applications over the grid remains a challenge.

This paper addresses the reliability issue for Grids on multi-core computers, equipped with virtualization technology, spanning the Internet. We have developed a novel framework that implements a policy-based virtual machine checkpointing mechanism. The framework enables implicit system-level fault tolerance without modifying existing operating systems, applications, or hardware. Unlike traditional checkpointing mechanisms, our mechanism utilizes a high-performance protocol that enables concurrent execution of checkpointing and computation. The proposed mechanism has been incorporated into modified KVM software [2], namely the Checkpointing-Enabled Virtual Machine (CEVM). Preliminary experiments have been conducted using the Linpack

benchmark. The performance evaluation results show that this framework is scalable, highly efficient, and low overhead. In particular, our mechanism reduces a typical checkpointing delay substantially to about one forth as compare to the traditional approach. We finally discuss performance impacts and future works.

## 2 Related Works

A number of checkpoint-restart solutions [4] [5] [6] [7] have been proposed to avoid the prohibitive cost of using hardware redundancy [3]. These past solutions include (a) taking advantage of the algorithmic properties of some scientific applications, which converge to the correct result even in the presence of system failures [8], (b) modifying the application's source code to provide explicit checkpoint/recovery [9], and (c) using the compiler to automatically insert the checkpoint code in a way that is nearly transparent to the application programmer [10].

An increasing number of research projects are putting emphasis on achieving user-transparent, automatic and efficient checkpoint and restart [11] [12] [13]. The solution to achieving user-transparent normally involves solving several problems, such as the virtualization of computation and communication [14], the identification a global recovery line to take a coordinated snapshot of the system, and the implementation of process migration algorithms [15].

Kernel-level checkpointing [16] [17] is highly responsive, with minimal overhead. The main advantage of this approach is that it is totally user-transparent and requires no changes to any application code. The downside is the increased complexity of working at kernel level,

with rapidly changing and often undocumented kernel versions, and the demanding constraint of porting the checkpoint/restart mechanisms to multiple processor and network architectures. An integrated solution to checkpoint/restart is presented in [18]. The proposed software infrastructure is based on LAM/MPI and BLCR (Berkeley Labs Checkpoint Restart).

The emphasis of this work is in the development of a VM checkpoint-restart system. Virtualization technologies, such as Qemu [19], KVM [20], VMware [21] and Xen [22], have recently gained popularity in the academic and industrial communities. A VM is an ideal computing platform for long-running Grid applications since most virtualization software has features to save and restore VM state without modifying guest OS or applications.

However, creating a checkpoint of a VM is non-trivial since it may cause too much delay and render VM checkpointing impractical in real life. Checkpointing typically involves saving the VM state and disk image to a checkpoint file. Since the VM must stop computation to perform checkpointing, the application response time would increase with checkpoint delays. Since the VM state and disk image are large in general, creating a checkpoint using the traditional VM save state feature and straightforwardly making a copy of disk image would be prohibitive.

### 3 Overview of CEVM

To achieve practical VM checkpointing, we propose a novel checkpointing protocol that exploits multi-cores parallelism by performing checkpointing operations and VM computation simultaneously on two different cores to reduce checkpointing delay. CEVM has the following highlights.

**Live migration:** Live migration is the key technique CEVM use to reduce checkpointing delay. From the Figure, we first migrate a VM from one processor core to another in order to let the VM on the original processor (Old\_VM) performs checkpointing while the VM on the new processor (New\_VM) continues execution (See (1) in Figure 1). We will discuss live migration more in the next section.

**Checkpointing and Computation Overlap:** After the migration, the protocol instructs the Old\_VM to save its state to disk (See (2) and (3) in Figure 1), while letting the New\_VM continues computation. The overlapping would hide most of disk access delays occurring during the creation of a checkpoint file (“VM Checkpoint” in the Figure).

In CEVM, all disk images, except the Base Disk Image, are copy-on-write (COW) images. They form another layer, namely the Disk Update layer (U), on top of the Base Disk. During VM computation, if the VM want to write data to disk image, CEVM would record all disk modifications to U. On the other hand, if the VM want to read data from disk, CEVM would look for the data in U first. If it cannot find the data there, CEVM will try to look for it in the Base Disk image. Normally, CEVM uses two layers of disk images i.e. U and Base Disk Image as mentioned earlier. However, during checkpointing, CEVM use three layers of disk images for the execution of New\_VM. As the result, the New\_VM would be able to continue read and write data to disk at the same time with the checkpointing operations.

To create a checkpoint file, the Old\_VM must replicate U to a new file (Copy of U), and then save the VM state (VS) to that file (or the VM checkpoint file). The computation of the New\_VM would not corrupt the replication of U because CEVM stores all changes New\_VM made to disks in T.

Moreover, in maintaining good disk operation performance, the checkpoint file (i.e.,

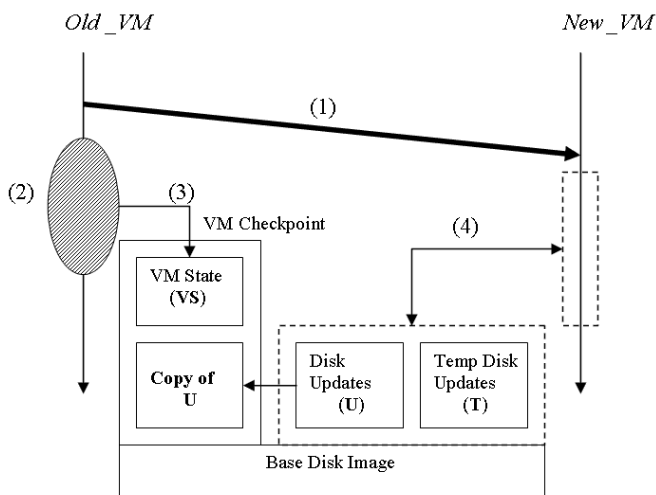


Figure 1. CEVM Architecture

Copy of U with VS) and all disk images the VM uses for computation (i.e., T, U, and Base Disk Image) should locate on different physical disks on a host computer to avoid disk contention problems when the Old\_VM creates a checkpoint file, and the New\_VM accesses disk images at the same time.

**Maintaining concurrent disk accesses:** While the Old\_VM is creating a checkpoint, the New\_VM must be able to read or write disk as usual. However, the New\_VM cannot write data directly to U because we have to keep it unchanged until its replication complete. Changing contents of U during the replication may make the copied disk image inconsistent with VS and, thus, corrupt the checkpoint file.

Therefore, our protocol uses a new temporary file (T), another layer of copy-on-write disk image on top of U and Base Disk Image, to keep all disk updates generated by New\_VM during the replication. From Figure 1, New\_VM would use three disk images i.e., T, U, and Base Disk image. The disk image T is a COW on top of U, and in turn U is a COW image on top of Base Disk image. When New\_VM writes data to disks, CEVM would record disk image modification on T, and thus, leaving the contents of U intact. On the other hand, when the New\_VM read disks, CEVM would look for the wanted data in T. If the data cannot be found, it will look for the data in U and Base Disk image, respectively.

After the Old\_VM finishes creating a checkpoint file, the New\_VM will merge contents of T into U (using QEMU's "commit" command), delete T, and then use U as its default disk image. The protocol finally terminates Old\_VM.

In our current implementation, CEVM suspends New\_VM during the committing operation. We have proposed a concurrent disk commitment protocol in [2] and plan to implement it in the near future.

## 4 Checkpointing Protocol

Our protocol implementation consists of two pieces of software: the checkpointing coordinator (CO) and CEVM. The coordinator is a program that controls order of activities between Old\_VM and New\_VM. The CEVM is the modified version of KVM, which

incorporates functions to perform VM activities for checkpointing.

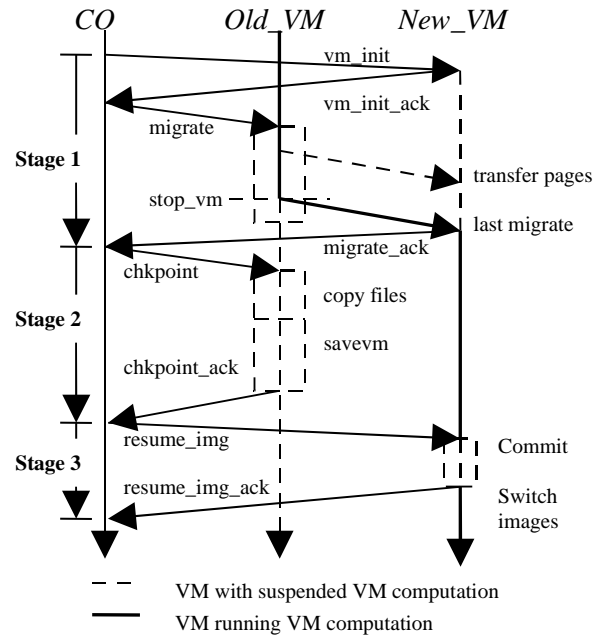


Figure 2. The CEVM protocol

First, the CO collaborates with CEVM to migrate Old\_VM to New\_VM (See Stage 1 in Figure 2). The CO creates new process to run New\_VM on a new processor core, which will inform CO (via `vm_init_ack` message) of its readiness and block waiting for migration information. The CO, in turn, instructs Old\_VM (via `migrate` message) to live migrate to New\_VM. Based on KVM live migration mechanism [23], KVM would interrupt Old\_VM's computation periodically to send dirty memory pages to New\_VM (transfer pages). Therefore, the computation on Old\_VM will progress simultaneously with VM's memory state transfer. Eventually, when the number of dirty pages is low, the Old\_VM stops VM computation (`stop_vm`) and transfer the rest of memory contents and state of every device (`last_migrate`). As the results, live migration would suspend VM computation only for a short period of time. After the migration finishes, the New\_VM send a (`migrate_ack`) message to inform CO of migration completion. Before starting the New\_VM, CEVM must create T and use it as a default disk image. As the result, after computation on New\_VM resumes, all disk updates would be stored in T.

In the second stage of CEVM protocol as illustrated in Figure 2, after CO receives the

migrate\_ack message, it collaborates with CEVM of Old\_VM to create a checkpoint file. Upon receiving the checkpoint message, the CEVM would copy U to “Copy of U” as mentioned earlier. Then, it would switch default disk image from U to “Copy of U” and save VM state there. To switch the disk image, we have to manipulate the “bdrv” data structures in KVM source code to maintain correct disk operations. Finally, the Old\_VM sends “checkpoint\_ack” message back to CO and then terminates. The file “Copy of U” is now a checkpoint file, which can be used to restore VM computation.

Finally, CO instructs New\_VM to suspend its VM computation and merge (using “commit” command) contents of “T” to “U”. Then, the New\_VM would discard T and set U to be the default disk image before resume VM computation.

## 5 Preliminary Results

We have conducted a set of preliminary experiments on the CEVM prototype. We ran our experiments on a PC with quad core 2.4 GHz Intel Q6600 CPU with 4 Gigabytes of memory and two 250GB SATA hard disks.

The processes involved in our experiments consists of a coordinator process (CO), a timer process, an original virtual machine, and a destination virtual machine. All virtual machines operate on top of our CEVM software. We configure these processes to run on different CPU cores. The guest OS we used for the virtual machine is Fedora core 8. The virtual machine is configured to run with 512 MB memory. The timer process is a process that waits to receive timing information from applications running inside the virtual machine and calculate application response time. Since the virtual machine will be stopped at a checkpointing event, we cannot rely on the virtual clock within the virtual machine to time application’s execution time.

We start our experiment by letting an original VM execute and starting a modified linpack benchmark [24] within the virtual machine. In this experiment, we have modified the linpack benchmark to execute for five rounds and to send “start timing” and “stop timing” messages to the timing server at the beginning and the end of program execution, respectively.

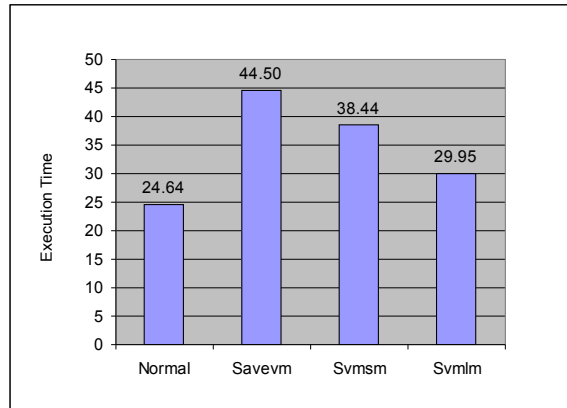


Figure 3. The response time running a modified Linpack benchmark on the CEVM prototype

Figure 3 reports the response time of the modified Linpack benchmark under various checkpointing situations. First, “Normal” denotes the execution time without checkpointing. We found that the “Normal” execution running on CEVM is similar to the execution time measured when running the same application on unmodified KVM demonstrating negligible execution overhead caused by additional functionalities of CEVM. Next, we denote “Savevm” to represent the response time with a checkpointing event using KVM’s “savevm” mechanism. The “Svmsm” represents the execution time with a checkpointing mechanism similarly to our proposed mechanism but using “Stop-and-Migrate” migration mechanism instead of live migration. That means we stop the virtual machine execution first and then migrate all of its memory to the destination virtual machine in one shot. Finally, we denote “Svmlm” for our approach, checkpointing using live migration. All reported data are the average of 10 measurements.

We found that our approach generate smallest amount of overheads comparing to others. By using KVM’s original “savevm”, there is an additional execution time of 19.85 seconds, while our approach cause only 5.30 seconds about one fourth of that of the original “savevm”. We also found that checkpointing using live migration perform much better than the “Stop-and-migrate” approach since live migration periodically sending memory pages to destination virtual machines during virtual machine execution. Thus, the virtual machine

computation can progress while migration, which involves transferring a large number of memory pages, takes place. From Figure 3, the “Svmsm” shows 13.79 seconds more time over “Normal” execution.

## 6 Conclusion and Future Woks

We have presented a novel virtual machine-level checkpointing mechanism that can substantially reduce checkpointing delays incurred to applications comparing with traditional approach. Implementing checkpointing mechanisms at the virtual machine-level allow high transparency to applications, guest operating systems, and hardware. We believe that this approach has high potential for providing fault tolerant for long-running Grid applications. Future works include the development of additional mechanisms to handle data communication, and to further reduce checkpointing delays.

## References

- [1] I. Foster and C. Kesselman (eds.) *The Grid: Blueprint for a new Computing Infrastructure* (Morgan Kaufmann Publishers, 1998) ISBN 1-55860-475-8.
- [2] K. Chanchio, C. Leangsuksun, H. Ong, V. Ratanasamoot, and A. Shafi, An Efficient Virtual Machine Checkpointing Mechanism for Hypervisor-based HPC systems, in Proc. of the High Availability and Performance Computing Workshop (HAPCW), Mar. 2008
- [3] M. Prvulovic, Z. Zhang, and J. Torrellas. Revive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors. In Proceedings of the International Symposium on Computer Architecture, May 25–29, 2002.
- [4] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent Checkpointing under Unix. In Proceedings of the Usenix Winter 1995 Technical Conference, January 16–20, 1995.
- [5] J. S. Plank, M. Beck, and G. Kingsley. Compiler-Assisted Memory Exclusion for Fast Checkpointing. IEEE Technical Committee on Operating Systems and Application Environments, 7(4):10–14, Winter 1995.
- [6] J. S. Plank, Y. Kim, and J. J. Dongarra. Diskless Checkpointing. IEEE Transactions on Parallel and Distributed Systems, 9(10):972–986, October 1998.
- [7] J. S. Plank and K. Li. ickp — A Consistent Checkpointer for Multicomputers. IEEE Parallel and Distributed Technologies, 2(2):62–67, Summer 1994.
- [8] A. Geist and C. Engelmann. Development of Naturally Fault Tolerant Algorithms for Computing on 100,000 Processors. Oak Ridge National Laboratory, 2002.
- [9] A. Beguelin, E. Seligman, and P. Stephan. Application Level Fault Tolerance in Heterogeneous Networks of Workstations. Journal of Parallel and Distributed Computing, 43(2):147–155, May 25, 1997.
- [10] C. J. Li and W. K. Fuch. CATCH - Compiler Assisted Techniques for Checkpointing. In Proceedings of the International Symposium on Fault Tolerant Computing, pages 74–81, June 1990.
- [11] F. Petrini, K. Davis, and J. C. Sancho. System-Level Fault-Tolerance in Large-Scale Parallel Machines with Buffered Coscheduling. In 9th IEEE Workshop on Fault-Tolerant Parallel, Distributed and Network-Centric Systems (FTPDS04), Santa Fe, NM, April 2004.
- [12] Q. Jiang and D. Manivannan. An Optimistic Checkpointing and Selective Message Logging Approach for Consistent Global Checkpointing Collection in Distributed Systems. In IEEE/ACM International Parallel and Distributed Processing Symposium (IPDPS 2007), Long Beach, CA, March 2007.
- [13] J. C. Sancho, F. Petrini, G. Johnson, J. Fernandez, and E. Frachtenberg. On the Feasibility of Incremental Checkpointing for Scientific Computing. In Proceedings of the International Parallel and Distributed Processing Symposium 2004 (IPDPS04), Santa Fe, NM, April 2004.
- [14] J. Ruscio, M. Heffner, and S. Varadarajan. DejaVu: Transparent User-Level Checkpointing, Migration and Recovery for Distributed Systems. In IEEE/ACM International Parallel and Distributed Processing Symposium (IPDPS 2007), Long Beach, CA, March 2007.
- [15] S. Chakravorty and L. Kal. A Fault Tolerance Protocol for Fast Fault Recovery. In IEEE/ACM International Parallel and Distributed Processing Symposium (IPDPS 2007), Long Beach, CA, March 2007.
- [16] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle. Fault Tolerance under UNIX. ACM Transactions on Computer Systems (TOCS), 7(1):1–24, February 1989.
- [17] R. Gioiosa, J. C. Sancho, S. Jiang, F. Petrini, and K. Davis. Transparent Incremental Checkpointing at Kernel Level: A Foundation for Fault Tolerance for Parallel Computers. In IEEE/ACM Conference on Supercomputing SC’05, Seattle, WA, November 2005.
- [18] C. Wang, F. Mueller, C. Engelmann, and S. Scott. A Job Pause Service under LAM/MPI+BLCR for Transparent Fault Tolerance. In IEEE/ACM International Parallel and Distributed Processing Symposium (IPDPS 2007), Long Beach, CA, March 2007.
- [19] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In Proceedings of the USENIX 2005 Annual Technical Conference, FREENIX Track, pages 41–46, 2005.
- [20] Avi Kiviti, Yanjv Kamay, Dor Laor, Uri Lublin, Anthony Linguori, kvm: the linux virtual machine monitor, In Proceedings of the Linux Symposium, 2007.
- [21] VMware, Inc. <http://www.vmware.com>.
- [22] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the Art of Virtualization. In Proceedings of the ACM Symposium on Operating Systems Principles, October 2003.
- [23] Qumranet, Kernel Based Virtual Machine <http://kvm.qumranet.com/kvmwiki>
- [24] <http://www.netlib.org/linpack/>