

An Efficient Virtual Machine Checkpointing Mechanism for Hypervisor-based HPC Systems

K. Chanchio¹, C. Leangsuksun², H. Ong³, V. Ratanasamoot¹, and A. Shafi⁴

¹Thammasat University, Rangsit Campus, Patumtani 12121, Thailand
kasidit@cs.tu.ac.th

²Louisiana Tech University, Ruston, LA 71272, USA
box@latech.edu

³Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA
hongong@ornl.gov

⁴NUST Institute of Information Technology, Pakistan
aamir.shafi@niit.edu.pk

Abstract- This paper presents the design of the *Checkpointing-Enabled Virtual Machine (CEVM)* architecture. A unique feature of the CEVM is the provision of an efficient mechanism for checkpointing Virtual Machines on hypervisor-based High Performance Computing (HPC) systems. Our goals are (1) to enable implicit system-level fault tolerance without modifying existing operating systems, applications or hardware, and (2) to minimize the space and time overhead needed to execute software that cannot tolerate faults. We accomplish these goals by leveraging hypervisor technologies that yielded tremendous reliability and productivity improvements in the HPC environment. In this paper we discuss the two novel protocols used in the VM checkpointing mechanism, which emphasize achieving efficiency as well as ensuring correctness.

I. INTRODUCTION

Virtualization is a technology that uses a hypervisor or Virtual-Machine Monitor (VMM) to manage the resources of the underlying hardware and provides an abstraction for one or more virtual machines [1]. Each Virtual Machine (VM) can run the entire Operating Systems (OS) stack and its applications. Virtualization technology is popular on desktop and server-based markets due to its ability to encapsulate the state of a running system, mediate all interactions between the hardware and software and isolate concurrently running software components. More recently, virtualization technology also shows promising impact, particularly in the area of high availability and reliability, for the High Performance Computing (HPC) community.

Contemporary virtualization software often provides mechanisms to checkpoint-restart VMs. Unlike data checkpointing, the VM checkpointing typically involves saving the states of hardware devices, memory, and disk image of a VM to physical disks. This operation is known to be a time consuming process due to potentially large VMs memory and/or disk image size. Indeed making a copy of the whole disk image is sometime not practical as its size could be in the order of several gigabytes. Although solutions such as

using a stack file system exist, they may also take a long period of time to perform disk I/O operations. Meanwhile, all computations must be stopped until the operation completes. Thus, a VM can be “out of order” for a considerable time. Consequently, the turnaround time of applications running in a VM would increase significantly due to the delay in VM checkpointing. This is certainly unacceptable for most HPC applications.

This work addresses these issues by developing the *Checkpointing-Enabled Virtual Machine (CEVM)* system. The CEVM is a VM emulator able to perform checkpoint and restart operations efficiently in a multiprocessing environment. We expect the CEVM system (1) to enable implicit system-level fault tolerance without modifying existing operating systems, applications or hardware, and (2) to minimize the space and time overhead needed to execute software that cannot tolerate faults. In this paper, we discuss the design of the VM checkpointing mechanism in CEVM, emphasizing the efficiency of the checkpointing operations and minimizing the VM downtime.¹

Briefly, novel contributions of this paper include: (i) an efficient *VM checkpointing protocol*, and (ii) an efficient *disk image manipulation* protocol. The VM checkpointing protocol utilizes the VM process migration mechanism [2] to move the execution of a VM to a new processor while allowing the original processor to save the VM’s state to physical storage; this approach enables the checkpointing operation to execute without halting the VM. The protocol primary task is thus coordinating the VM checkpointing activities on two processors and hiding overheads from the VM execution. The disk image manipulation protocol, on the other hand, is responsible for the creation and manipulation of a temporary

¹ The discussion on the VM checkpointing mechanism will be limited to sequential processing while leaving parallel processing as future work.

overlay disk image layer to handle all disk updates during a checkpointing event. Currently, multiple VMs cannot access the same disk image simultaneously in the existing VM design. This is because multiple updates from different VMs could cause inconsistent states between the VM's OS and disk image contents. For example, imagine a scenario where two VMs share the same disk image. In this case it is possible that while the first VM is checkpointing, the second VM may update the shared disk and corrupt the checkpointed data. Our protocol assures that these operations are performed simultaneously and safely during a checkpointing operation. Additionally, the protocol must support multiple checkpointing operations without introducing additional disk accesses overheads.

This paper is organized as follows: Section II discusses a related work. The overview of our solution is given in Section III. Section IV describes our VM checkpointing and disk manipulation protocols. Section V discusses ongoing implementation. Section VI concludes and discusses future work.

II. RELATED WORK

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]. Proposed solutions to achieve this typically 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. Additionally, there are many details of a process's state, which are only known to the kernel or are otherwise difficult to re-create, such as the status of open files and signal handling, can be easily captured at kernel level. 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 contribution 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 building block to implement checkpoint/restart algorithms, and to virtualize the resources in a cluster. For example, VM can simplify cluster management by allowing the installation of customized OS with different levels of security, instrumentation, and services. In order to gain wide acceptance in the user community these solutions must incur very low overhead—both in fault-free and recovery mode—and be highly scalable.

III. CEVM ARCHITECTURE OVERVIEW

The design of CEVM architecture is to leverage live migration to support VM checkpointing mechanism. The central idea is to minimize the associated overhead by concurrently executing VM computation on two different processors. CEVM's checkpointing mechanisms can simply be described in three basic steps:

1. **Live migration:** We first migrate a VM from one processor to another. The VM on the original processor performs checkpointing, while the VM on the new processor continues to execute. Live migration could be conducted quickly across different cores or processors on a multi-core or SMP machine. It could also be done across two separate computers over a network.
2. **Checkpointing and Computation Overlap:** Usually, after the VM migrated, the VM on the original processor (or the original VM) is stopped. The new VM on the destination processor would continue its execution. In the CEVM design, the original VM would perform checkpointing instead of being stopped. Moreover in multi-core or SMP environments, we can assign the new VM and the original VM to any core or processor. Therefore, if users want to let the new VM continue running on the original processor and the checkpointing VM run on a new processor, they could readily do so by adjusting CPU scheduling parameters.
3. **Maintaining concurrent disk accesses:** The two VM's can simultaneously access to the same disk during a checkpointing operation. Access to disk is prohibited by most VM technologies, including Xen and QEMU. This is because the potential to produce data inconsistencies between guest OS and the disk image is high. In order to make that possible, CEVM creates a new temporary file to keep all disk updates generated by the new VM. This allows the old VM to checkpoint without interruption. After the checkpointing finishes, the new temporary file will be merged with the original temporary file.

Figure 1 illustrates the three basic steps in the CEVM architecture. Supposed a VM, namely *Old_VM*, originally executes on one processor. We first migrate the VM to another processor (See (1) in Figure 1). We then denote *New_VM* to be the new VM at the destination after the migration. At the end of the migration, *New_VM* resumes normal execution while *Old_VM* performs checkpointing by recording the VM’s state to a checkpoint file (See (2) and (3) in Figure 1). In our design, a VM can perform checkpointing if and only if the previous checkpointing operations have already finished.

The design of disk image in CEVM is based on that of QEMU. QEMU allows stackable storage in its virtual disk organization, where a disk image could operate under QEMU “snapshot” mode. The virtual disk consists of two parts: an updated disk image (labeled as “U” in Figure 1) and a base disk image. The updated disk image contains all changes made to the base disk image from the time the VM execution begins until the checkpointing starts. The base disk image is read-only. Information in the update image can be incorporated to the base disk image using QEMU’s “commit” command.

We define a VM checkpoint as a set of data that can be used together with the base disk image to restore an execution of a VM. This is equivalent to QEMU’s VM snapshot. The VM checkpoint, therefore, consists of a VM execution state (labeled as “VS” in Figure 1) and an update disk image at the moment when the snapshot was taken. The VM execution state consists of the state of all hardware devices including RAM’s contents. In QEMU, users can create a snapshot using the “savevm” command and restore VM execution from a snapshot using the “loadvm” command.

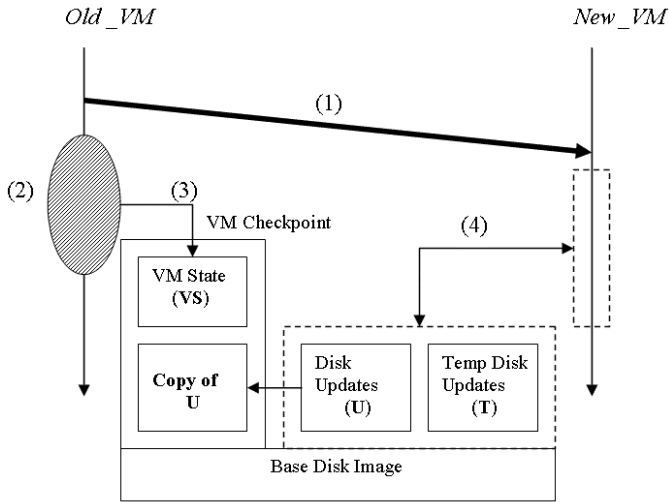


Figure 1. CEVM approach to implement efficient checkpointing mechanisms

Checkpointing is performed incorrectly if *Old_VM* and *New_VM* operate on “U” at the same time. In Figure 1, *Old_VM* creates a checkpoint immediately after the migration

finishes (See (2)). In doing so, *Old_VM* has to dump its “VS” and copy the current updated disk image (labeled as “Copy of U” in Figure 1) to a checkpoint file. Meanwhile, *New_VM* continues its computation and may want to read or write data to disk (See (4)). In the case of read accesses, *New_VM* can retrieve data from either “U” or the base disk image. On the other hand, if *New_VM* writes data to disk, problems may arise. Since *Old_VM* is copying “U” to a checkpoint file, any change made to “U” could corrupt the content of the checkpoint.

To overlap VM checkpointing and computation possible, we propose a novel mechanism to enable concurrent disk accesses to the same disk image using two VMs. In our solution, we create another layer to update disk image, namely a *temporary* disk update file, (labeled as “T” in Figure 1) on top of “U”. We use “T” to hold all disk updates that *New_VM* makes during checkpointing period. After the copying of “U” finishes, we let *Old_VM* integrate contents of “T” to “U”. During the integration, our mechanism would redirect all disk access requests from *New_VM* to *Old_VM*. This allows *Old_VM* to schedule the disk image integration and updates in a consistent manner. Since *Old_VM* is the only process that access “U”, “T”, and the base disk image, it knows exactly where to read and write data from the disk during the integration operation. After the integration completes, *New_VM* would be informed to stop the redirection and use “U” and the base disk image.

On the event of VM recovery, CEVM will load the VM execution state from a checkpoint file, and copy the update disk image in the checkpoint back to the update disk image the VM is currently using. As shown in Figure 1, the “copy of U” in the checkpoint will be copied back to “U”. The VM will resume execution afterwards.

IV. PROTOCOLS

This section describes the *VM checkpointing protocol* and the *disk image manipulation protocol*; these are the two key components of CEVM system. Briefly, the checkpointing protocol coordinates activities among all parties involved in a checkpointing operation, while the disk image manipulation protocol handles disk accesses and disk structure manipulation during checkpointing.

A. The VM Checkpointing Protocol

Figure 2 illustrates our checkpointing protocol. We let *CO* denotes as a process that coordinates activities on *Old_VM* and *New_VM*. When checkpointing starts, *CO* first sends a *VM_INIT* request to the destination processor to load a new VM to memory. The *New_VM* would wait for the VM execution state from *Old_VM*. Then, it sends a *VM_INIT_ACK* to *CO* to inform that the loading is successful. *CO*, in turn, sends a *CPT_START* message to instruct *Old_VM* to migrate its state to *New_VM*.

After *Old_VM* migrated, it creates a checkpoint file. At the same time, *New_VM* continues its normal execution. During the checkpointing period, the *Old_VM* and *New_VM* would have to access the same disk image. The key idea of our solution is the creation of a *temporary disk update file* to hold all updates to disk images made by *New_VM* during checkpointing. We will discuss this mechanism in more details in Section IV B. When *Old_VM* finishes checkpointing, it sends a CPT_DONE message to *CO*, which subsequently sends an INTEGRATE_STORAGE message to instruct *New_VM* to stop writing disk updates to the temporary file so that *Old_VM* can integrate the temporary file's contents to the original disk image.

In response, *New_VM* sends an INTERGRATE_START message to *Old_VM* to start the integration. All disk updates occur on *New_VM* beyond this point are redirected to *Old_VM*, which would schedule these updates to appropriate disk locations during the integration. At the end of the integration, three messages consisting of INTEGRATE_END, LAST_UPDATE, and UPDATE_DONE are exchanged between *Old_VM* and *New_VM* to make sure all disk update requests from the new VM have been served. Then, *New_VM* would discard the temporary file and send all disk updates to the original disk image (just like before checkpointing took place). More details on disk integration operations will be described in the next section. Finally, *New_VM* sends a RESUME_DISK_ACCESS message to tell *CO* that it is now using the original disk image for normal VM execution. The *CO* sends a TERMINATE_OLD_VM message to terminate *Old_VM*.

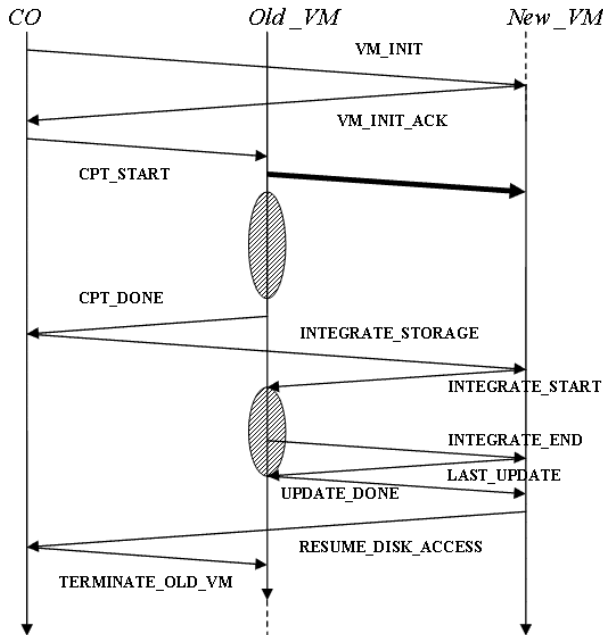


Figure 2. The illustration of CEVM checkpointing protocol

B. Disk Image Manipulation Protocol

Inconsistencies between the checkpointed VM and its disk image could arise if we use the original QEMU disk structures for our checkpointing mechanism. For example, in Figure 1, a checkpoint consists of the VM State (“VS” in Figure 1) and a copy of all disk updates to the base image (“copy of U” in Figure 1). The base disk image is assumed to be read-only throughout the VM’s computation. Traditionally, on a checkpointing event, e.g., using “savevm” command in QEMU or KVM, the VM is paused so that contents of the VM execution state and disk updates can be stored in a checkpoint. It is crucial that the checkpointing operation is atomic so that the VM execution state and the disk update contents are consistent to one another. Any changes to disk block contents unrecognized to the VM’s guest operating system could cause it to work incorrectly or become unbootable.

To address the problem described above, CEVM implements a two-step solution, focusing on both efficiency and correctness. The two-step solution is illustrated in Figure 1. The first step attains efficiency by enabling checkpointing and disk accesses to progress concurrently. In doing so, CEVM performs the following:

1. Before the migration, the *New_VM* has to create the temporary file, namely “T”, to hold all disk updates during checkpointing. “T” must also be the same copy-on-write disk format as “U”. After the migration, when *New_VM* operates, it writes all disk updates to “T”. On data retrieval, *New_VM* would search for data from “T” first. If it cannot find the data, then it searches “U” and then the base disk image.
2. When *Old_VM* receives CPT_START, it set “U” to be read-only and then migrates. Right after the migration finishes, *Old_VM* performs checkpointing by writing “VS” to a checkpoint file, and then copying “U” to it. Since “U” is read-only, it cannot be altered from any disk writes from *New_VM*. Finally, *Old_VM* sends CPT_DONE to inform *CO* that the checkpoint file has been successfully created.

The second step ensures correctness by requiring the integration of all changes occur during a checkpointing event to the original disk images so that VM execution can perform normally on “U” like before checkpointing. After *CO* receives CPT_DONE, it sends an INTEGRATE_STORAGE message to inform the *New_VM* that *Old_VM* is ready to perform disk image integration. Then, *New_VM* sends INTEGRATE_START to instruct *Old_VM* to integrate contents of “T” to that of “U”. From this point onward, *New_VM* stops writing any further disk updates to “T” but send the updates to *Old_VM* instead.

Based on the QCOW image format [23], we propose an algorithm, listed in Figure 3, to perform the integration. The *Old_VM* executes this algorithm after it receives the INTEGRATION_START message. The algorithm repeatedly integrates one unit of “T’s” content to “U” and checks for an incoming disk update request from *New_VM*. If there is such a

request arrives, it integrates contents of the request to either “U” or “T” based on the current “T” and “U” integration status. According to the QCOW format, an image file contains a set of data clusters. The cluster size is varied (e.g., 4K bytes) depending on disk image configuration. Each data cluster contains contents of a number of disk sectors. This algorithm considers a data cluster to be a basic unit of data to operate on. It first calls the nextCluster() function (line 3) to obtain the content of the first cluster stored in T and the real address of the base disk image associated with the content (Taddress). The nextCluster() function traverses the L1 and L2 lookup tables in T in a depth first search manner and retrieves a cluster’s content in order. Then, the algorithm calls function updateCluster() to write the retrieved contents to U at the location Taddress (line 4).

```

1. While  $\neg DONE$  do
2.   if  $\neg EOF\_on\_T$  then
3.     if nextCluster(T, &content, &Taddress)  $\neq EOF$  then
4.       updateCluster(U, content, Taddress)
5.     else
6.       send INTEGRATE_END to New_VM
7.        $EOF\_on\_T = true$ 
8.     end if
9.   end if
10.  nrecv m from New_VM
11.  if ( $m \neq LAST\_UPDATE \wedge m \neq NULL$ ) then
12.    accessDisk(m)
13.  else if ( $m \neq NULL$ ) then
14.    send UPDATE_DONE to New_VM
15.     $DONE = true$ 
16.  end if
17. end while

```

Figure 3. An algorithm to integrate T’s content to U

After integrating one data unit to U, the algorithm invokes the nrecv() function (line 10) to check if there are any incoming disk update request from New_VM. The nrecv() function is a non-blocking receive function. It would proceed to copy data from “T” to “U” if no message were received. On the other hand, if Old_VM receives a message m, it invokes accessDisk() function to process the request.

Figure 4 listed the accessDisk() function. If m is a disk update request, it integrates the content of m to either “T” or “U” depending on their respective disk address. Henceforth, we define m to contain a cluster of data, and denote m.content and m.address to represent the content and the real disk address of m, respectively. If m’s address is lesser than the last address of “T” plus the cluster’s size, the algorithm concludes that the integration of “T” to “U” has exceeded the m’s address bound. Therefore, the new content in m can be written to “U” directly. Otherwise, it would write the new content to “T”. As the integration of “T’s” contents to “U” proceeds, the new content will be copied to “U” eventually.

On the other hand, if m is a disk read request, the function accessDisk() would retrieve data from “T”, “U”, or base image considering the value of Taddress. If the retrieval disk address (m.address) is lesser than Taddress plus the cluster’s size (line 8 in Figure 4), it searches for the cluster using “U’s” m.address. If the cluster does not exist in “U”, it retrieves the cluster from the base image. Otherwise, it searches data from “T” first if “U’s” address is greater than the sum of “T’s” address and cluster size, If the search fails, it looks for the data in “U”. Eventually, it looks for the request data cluster from the base image if it does not exist in “T” and “U”.

```

AccessDisk(m)
1. if m is a disk update request then
2.   if ( $m.address < Taddress + cluster\_size$ )  $\vee EOF\_on\_T$  then
3.     updateCluster(U, m.content, m.address)
4.   else
5.     updateCluster(T, m.content, m.address)
6.   end if
7. else // m is a read request
8.   if ( $m.address < Taddress + cluster\_size$ )  $\vee EOF\_on\_T$  then
9.     read from U or base image, and send a cluster to New_VM
10.  else
11.    read from T or U or base image, and send a cluster
12.  end if
13. end if

```

Figure 4. A function to access disk image during integration

The algorithm (in Figure 3) also makes sure that there are no new update requests from New_VM left in transit when it finishes the integration. It achieves this using INTEGRATE_END, LAST_UPDATE, and UPDATE_DONE messages. The INTEGRATE_END would be transmitted to the New_VM when data retrieval on “T” reaches EOF. Upon the arrival of this message, New_VM can no longer send new disk update requests to Old_VM. At this stage, it would send the LAST_UPDATE message to Old_VM and store any new update requests to an internal buffer. The arrival of LAST_UPDATE to Old_VM means the messages sent before it has already been received and processed. The Old_VM will response with the UPDATE_DONE message. After receiving this message, New_VM will process all the update requests in its internal buffer and future updates on “U”.

V. IMPLEMENTATION AND ANALYSIS DISCUSSION

In multi-core or multiprocessor environments, we can execute Old_VM and New_VM on two different cores. We can use the “sched_setaffinity()” system call or Linux “taskset” command to assign VM process to a core or a processor. Since the two VM are on the same computer, the migration can be done quickly across processors or cores. Moreover, users can assign the Old_VM to a new processor while the New_VM to the original one as they wish.

Our protocols can be implemented into the existing systems, especially in KVM and QEMU. It is our goal to leverage

existing technology as much as possible to make our checkpointing mechanism highly portable and reduce implementation time. We are considering two implementation alternatives: thread-based and process-based. The former represents each VM with a thread while the latter represents each with a process. The thread base one would make VM migration quite efficient since both thread can share VM data structure; however, it requires more modification to the source code than the process-based solution.

To study the feasibility of our design, we have measured the VM migration and checkpointing performance to see potential performance improvement. Since the main goal of our mechanism is to use live migration to hide the checkpointing delay, we consider the migration time to be the lower bound time and the checkpointing time to be the upper bound time. Figure 5 shows our preliminary performance comparison between VM migration and checkpointing using KVM version 60. The measurements have been done on a notebook computer with Centrino Duo T7250 CPU and 2 Gigabytes of RAM. The host OS is the 64 bits Ubuntu Linux We have created two hard disk images for a Debian and Fedora 8 guest OSes, respectively. We run KVM every time with a virtual RAM size of 1 Gigabytes.

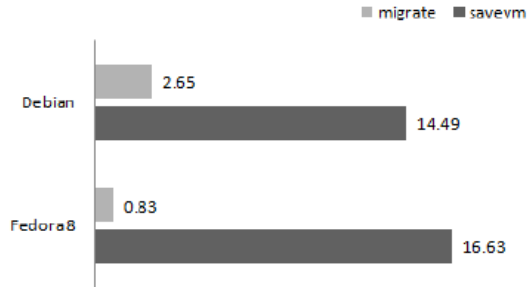


Figure 5 comparisons of migration and checkpointing time

In our experiments, we run a KVM on the notebook one at a time. We use Debian as the guest OS in the first experiment and let it migrates on the same machine and measure performance. Every performance report showed in Figure 5 is an average of 10 measurements. In the second set of experiments, we restart KVM with Debian and measure checkpointing performance (using KVM “savevm” command). We also do the same thing with Fedora 8 guest OS. From the preliminary results showed in Figure 5, the migration time is much faster than the checkpointing time. On Debian guest OS, the checkpointing cost is 5.4 times (14.49 seconds/2.65 seconds) higher than the migration cost, while the checkpointing cost is 20 times (16.63 seconds/0.83 seconds) higher than that of VM migration. We would want our mechanism to perform close to the lower bound time for better performance.

Thus, we can model the upper bound time, lower bound time, and estimated execution performance of total VM execution time as follows. Supposed a VM creates n

checkpoints throughout its lifetime, we can model its upper bound total computation time as

$$T_{vm} = T_{comp} + nT_{cpt},$$

where T_{vm} denotes the total execution time, T_{comp} denotes computation time, and T_{cpt} denotes an average checkpointing delay using the “savevm” command. On the other hand, we may express the lower bound time of our approach as

$$T_{vm} = T_{comp} + nT_{mg}$$

where T_{mg} denotes an average migration delay using “migrate” command. Finally, we can model the total VM execution time using our approach as

$$T_{vm} = T_{comp} + nT_{cevm} + T_o$$

where T_{cevm} denotes an average checkpointing delay using our approach and T_o denotes additional overheads due to checkpoint coordination and disk accesses during checkpointing. Thus, based on the expected lower bound time, upper bound time, and estimated delay mentioned earlier, we would expect the relationship among T_{cpt} , T_{mg} , and T_{cevm} to be

$$T_{mg} \leq T_{cevm} + T_o/n \leq T_{cpt}$$

Thus, the average checkpointing delay plus the average additional overheads during the checkpoint should lie between the upper bound and lower bound time.

VI. CONCLUSIONS AND FUTURE WORK

We have presented the design of CEVM system for hypervisor-based HPC environment. In particular, we discussed the two protocols, checkpointing and disk manipulations, used in the VM checkpointing mechanism for promoting checkpointing efficiency and accuracy. A preliminary performance measurement model for our protocol is also presented.

Our future work will consist of two main directions – firstly, we will prototype and evaluate the CEVM system. Secondly, we continue to investigate the scalability of the system.

REFERENCES

- [1] Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: a view from Berkeley," Electrical Engineering and Computer Sciences, University of California at Berkeley, Tech. Rep. UCB/EECS-2006-183, December 2006. <http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf>
- [2] 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.
- [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, Yaniv 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] M. McLoughlin, The QCOW image format, June 2006. Available from <http://www.gnome.org/~markmc/qcow-image-format.html>,