2009 Fourth ChinaGrid Annual Conference

Optimize Performance of Virtual Machine Checkpointing via Memory Exclusion

Haikun Liu, Hai Jin, Xiaofei Liao
Services Computing Technology and System Lab
Cluster and Grid Computing Lab
School of Computer Science and Technology
Huazhong University of Science and Technology, Wuhan, 430074, China
hjin@hust.edu.cn

***Abstract—Virtual Machine* (VM) level checkpoints bring several advantages which process-level checkpoint implementation can hardly provide: compatibility, transparence, flexibility, and simplicity. However, the size of VM-level checkpoint may be very large and even in the order of gigabytes. This disadvantage causes the VM checkpointing and restart time become very long. To reduce the size of VM checkpoint, this paper proposes a memory exclusion scheme using ballooning mechanism, which omits saving unnecessary free pages in the VM. We implement our prototype in Xen environment. Experimental measurements show our approach can significantly reduce the size of VM checkpoint with minimal runtime overhead, thereby greatly improve the checkpoint performance.**

Keywords – virtual machine; checkpoint; restart; memory exclusion; ballooning

I. INTRODUCTION

System level virtualization technologies allow for an easy abstraction of the physic hardware [1], [15]. They encapsulate the entire running environment including the state of virtual hardware resources, as well as an operating system and all its applications, inside a software container called *virtual machine* (VM). This encapsulation allows for full-system checkpoint and restart therefore allows us to explore a number of exciting ideas for system mobility and fault tolerance. A VM-based checkpoint typically saves transient state of the system on any standard data storage medium. VM can later be quickly restored from the point in time when the checkpoint was created by reloading its state from persistent storage.

The encapsulation of virtual machine inherently brings VM-level checkpoints several advantages which the process-level checkpoint implementation can hardly provide: 1) Compatibility, VM-level checkpoints are completely compatible with all operating systems and relevant applications, no matter whether they provide checkpointing/restart mechanism; 2) Transparence, all applications as well as the under operation system kernel need not to be modified and recompiled, or re-linked against a checkpoint library; 3) Flexibility, the checkpoint interval can be customized in a more sophisticate scheme at the VMM level, but not automatically initiated in regular fixed intervals at process-level; 4) Simplicity, some applications can be consolidated in a VM container if they cooperate to supply services, this scheme can avoid considering the complicated internal process communications when cooperative checkpoints are taken.

VM-level Checkpoint/Restart has been widely studied in recent years for high availability [3] and fault-tolerant system [10], VM migration [4], playback debuggers [8], et al. Many VMMs (*Virtual Machine Monitor*) support checkpointing the state of a running virtual machine [1], [15]. However, the disadvantage of VM-level checkpoint is that the checkpoint size may be very large and even in the order of gigabytes. Because a VM is usually configured to use plentiful physical memory to guarantee *quality of service* (QoS), while VM-level checkpointing schemes are implemented to take core-dump style snapshots of the computational state. Dumping such a large amount of transient memory image to persistent disk is a time consuming process and may disrupt other active services residing in the same host through resource contention (e.g., CPU, I/O bandwidth). On the other hand, restoring the VM from the dumps also cost more overhead if the checkpoint size is larger. Moreover, if the system is configured to place more (i.e., dense) checkpoints within short intervals for high availability, the local storage or network file system can be used up quickly by continuous saving the heavy checkpoint files.

The Xen hypervisor currently provides mechanisms that allow users to save a VM's execution state, and restart the VM at another time or location. But Xen checkpoint implementation simply save the VM's whole machine page frames which are relevant to pseudo-physical memory address mapping. Nevertheless, it is unnecessary to save the unallocated free pages within the guest OS when the checkpoint is taken.

To reduce the checkpoint size of virtual machine, we propose a memory exclusion scheme using ballooning mechanism, which prevents the checkpoint daemon from saving unnecessary free pages in the VM. Our approach can significantly reduce the checkpoint size even the VM is running heavy workloads, and also speed up the VM checkpoint time and restart time. We design and implement the approach in the Xen 3.2 environment [1], and experimental measurements show our scheme can greatly improve the checkpoint performance when compared against the traditional Xen save

& restore approach by each of the following metrics: checkpoint size, checkpoint time, and restart time.

The rest of the paper is organized as follows. In section II we discuss the related work. In section III we give a brief introduction about the groundwork – Xen VM save & restore. In section IV we show the design details of memory exclusion scheme using ballooning mechanism. In section V we present the experiments undertaken and results obtained. Finally, in section VI we conclude our works.

II. Related Work

There are mainly two approaches proposed to reduce the checkpoint size in process-level checkpointing. An important concept in size reduction is memory exclusion [11],[12], with which the checkpointing operation does not save the unnecessary regions of a process's memory (read-only or dead pages). Incremental checkpointing is a well-know optimization wherein only the regions of the checkpoint that has been changed since the most recent checkpoint need to be saved. The unmodified potions are restored form the pervious checkpoints (so-called read-only memory). The virtual memory protection mechanism is commonly used to keep track of the page-granularity modifications. This technique has been evaluated and the experimental results show that the reduction in the size of the checkpoint data depends strongly on the application [13]. To the best of our knowledge, incremental checkpointing has seen very few implementations in Linux at process-level, and never before at the operating system level.

Another important approach is excluding unused memory regions of the process state. Those portions of memory can be excluded from a checkpoint file because they are dead, meaning the values of the memory will never be read or written, and thus their values are not necessary for the successful completion of the program. So those portions of memory do not need to be written to disk. Both Libckpt [11] and CLIP [5] significantly reduce the size of checkpoints using those strategies. Paul Stodghill et al [8] proposed an application-level self-checkpointing mechanism in the C3 system. It utilizes application knowledge to minimize the amount of information saved in the checkpoint. But this approach needs the compile-level assistance and violates the program integrality.

However, memory exclusion mechanism can only be used for use-level checkpointing implementation as programmers must explicitly declare the excludable areas of memory or the application-specific knowledge must be known by the checkpointing daemon. The semantic gap between the VMM and virtual machine make application-level knowledge VMM-agnostic. It is hard to apply this technique for VM-level checkpointing.

The ballooning mechanism has been achieved as a manner to manage memory in virtualization environment, such as VMWare [15] and Xen [1]. Ballooning performs the act of changing the view of physical memory (and pseudo-physical memory) mapping so that a guest OS can either give up or request more memory from the hypervisor's memory pool. To optimize the performance of VM migration, Rosenblum et al [14] employ ballooning mechanism to zero out the less useful data in a VM for compression. It reduces the size of the compressed memory state and thus reduces the data flying across network. Michael R. Hines and Kartik Gopalan [7] improved a Dynamic Self-Ballooning (DSB) technique, in which the guest OS actively balloons down its memory footprint without human intervention. It avoids transferring free pages for both pre-copy and post-copy migration schemes. To the best of our knowledge, ballooning mechanism has not been adopted systematically for VM checkpointing in the latest stable Xen release. Our work addresses this deficiency by introducing a memory exclusion strategy with ballooning mechanism. It significantly improves the performance of VM checkpointing with minimal runtime overhead.

III. Xen Save & Restore

Xen hypervisor currently provides several fault tolerance mechanisms for both paravirtualization and full virtualization solutions: 1) VM save & restore; 2) VM migration. The VM save and restore functions are the background of VM migration. The hypervisor, in conjunction with the host OS (also called domain 0), provides those management utilities to manage the VMs. The implementation of Xen checkpointing uses the following algorithm:

1） The VM first disconnects itself from virtual devices and disables all the interrupts;
2） The guest configuration information, include the *xen_start_info* structure, with the address mapping of MFN (*machine page frame number*) to PFN (guest *physical page frame number*) are serialized so that the VM can be recreated when the VM is restored from the checkpoint file;
3） A suspend hypercall is issued, and then domain0 waits for the Xen hypervisor to announce that the domain has suspended;
4） The guest memory pages are mapped and written out with a header listing the PFNs in batches;
5） Each VCPU state is written out with MFN to PFN fixups (e.g. CR3);
6） Flash I/Os;
7） Resume the stopped VM to continuous.

From the above steps, we can find out the VM memory pages are entirely copied to the checkpoint file. It makes the checkpoint size proportional to the amount of RAM configured during the VM booting. The following formula shows this case:

$$V_{ckpt} = (1+\theta)V_{mem} + V_{other}$$

where V_{ckpt} presents the checkpoint size, V_{mem} presents the configured memory size of the VM, V_{other} presents the space needed to save the VCPU state and device state. θ is a very small number used to calculate the space of MFN to PFN address mapping.

Table I presents the average checkpoint size, checkpoint time, and restart time for four VM configurations with 128MB, 256MB, 512MB, and 1024MB of memory. All the experiments are conducted to use the local disk. As we observed that the checkpoint sizes are directly determined by

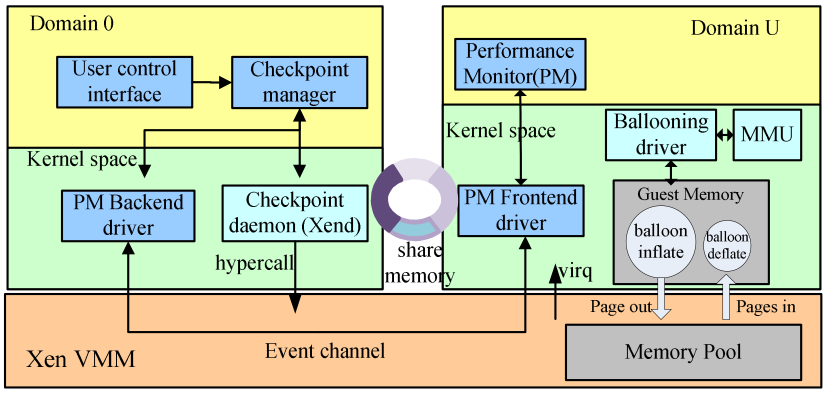

Figure 1. System architecture

TABLE I. CHECKPOINT PERFORMANCE OF XEN VIRTUAL MACHINE

VM RAM Configuration(MB)	Checkpoint size (MB)	Checkpointing time (sec)	Restart time (sec)
128	129	4.380	4.483
256	257	8.205	7.132
512	513	15.551	13.682
1024	1025	28.209	23.478

the VM memory configuration and do not depend on the applications scenario, checkpointing and restart times are directly correlated with the checkpoint size, so we only report the common results with different RAM configurations instead of different applications. We also observe all the VM's configured RAM are not fully used when the checkpoint is taken, so memory exclusion techniques may be employed to reduce the checkpoint size, thereby optimize the checkpointing performance.

IV. MEMORY EXCLUSION FOR CHECKPOINTING

This section shows how we use ballooning mechanism to achieve our VM-ME (VM memory exclusion) strategy for Xen checkpointing. We implement our prototype on top of Xen 3.2. We use the para-virtualized version of Linux 2.6.18.8 as our base.

The basic function of ballooning is to adjust a domain's memory usage by passing memory pages back and forth between the hypervisor and the virtual machine page allocator. As shown in Fig. 1, the balloon module is loaded into the guest OS as a pseudo-device driver or kernel service but is controlled by the hypervisor. The driver makes the memory adjustment by using the existing OS functions and avoids modifying the native memory management algorithms. When the hypervisor wants to reclaim memory, it instructs the driver to *inflate* the balloon by allocating pinned physical pages within the VM. Similarly, the hypervisor may *deflate* the balloon by instructing it to return previously allocated pages. When the balloon inflates, it tricks the guest OS to reserve physical memory and create memory pressure in the VM, the *memory management unit* (MMU) of the guest OS must reclaim space to satisfy the driver allocation request. The reclaimed pages are passed down to the hypervisor which in turn makes the physical memory available to other VMs. When the balloon deflates, the memory is made available again for general use by this VM.

A. *Detect memory pressure*

In current Xen implementation, ballooning mechanism is only used during guest OS boot time when it is first initialized and created. If the VM can not reserve enough memory from the Xen hypervisor, it steals some from the other VMs on the same host by inflating the balloon in other VMs. For our checkpointing scheme, to page out the unallocated free pages in the VM, we should actively perform ballooning once right before the checkpoint is taken. But how many pages should be reclaimed from the VM, the hypervisor or itself does not know it.

To bridge the semantic gap between the hypervisor and the above user domains, a performance monitor module is designed and placed in the domain U as shown in Fig. 1. The PM module collects the profile data of each VM and is directed by the domain 0. In our prototype, we mainly concern the memory usage pattern in the guest OS. To avoid violating the VM runtime state and degrading the application running performance by swapping some allocated pages to the block device, target memory of the VM is determined solely by the *Committed_AS* line in */proc/meminfo*. With the memory pressure information, we can explicitly fix the mount of memory the balloon should page out.

There are mainly two approaches to propagate the collected data to the management domain. A simple approach is using the virtual NIC with traditional socket programming interface. It is easy to implement, but with inefficient communication performance. Another approach is using interdomain shared memory in Xen. This inter-domain communication mechanism can greatly improve data transferring performance. For resource monitoring application, it is necessary to make the communication latency short enough between the monitor and decision-maker. As the virtual machines we monitored are resident on the same physical machine, inter-domain shared memory mechanisms can greatly increase the data transferring bandwidth, and also decrease the communication latency. In our prototype, we implement the communication between domain 0 and domain U with IDTS, which provides a suit of programming APIs for high-efficient inter-domain communication in our earlier work [6]. There are two kinds of device drivers as shown in Fig. 1: one is called backend driver, which runs in domain 0, and the other is called frontend driver, which runs in domain U. Device driver is lying in kernel space of domains, and provides convenient interfaces for domains to communicate with each other. Before communication tunnel is set up, Xenstore is used to set up shared memory regions and event channels for use with the split device drivers. The shared memory ring is a classic consumer-producer circular buffer, and process data with FIFO mechanism.

B. *Checkpointing implementation*

As shown in Fig. 1, our prototype mainly comprises four parts: directed ballooning, which excludes the free memory form the VM; performance monitor, which collects the memory usage data of the observed VM; checkpoint daemon, which takes VM checkpoint with Xen existing function; checkpoint manager, which cooperates with the other parts to perform our checkpointing scheme. We implement our checkpointing mechanism using the following algorithm:

1） A checkpointing request is issued from the user control interface in domain 0.

2） An event channel is set up for inter-domain communications by binding the PM backend driver in domain 0 and the PM frontend driver in domain U to the same port. Then the checkpoint manager informs the PM backend driver to collect the memory usage information of the running VM (domain U).

3） The PM frontend return the collected information (include the free pages, scrubable pages, total pages and so on) to domain 0.

4） With the memory pressure information, checkpoint manager calculates the total memory pages that can be reclaimed by the ballooning driver in domain U. Then it informs the Xen hypervisor to perform memory reservation by write a target value in the XenStore and wait for return message.

5） The balloon driver will watch this value, which indicates the *target* memory usage of the domain U. Then the ballooning process allocates the specified mount of free memory and hands those pages over to the hypervisor.

6） When the checkpoint manager receives the return message from Xend, it calls the native Xen checkpointing daemon to save the running domain. Along with the checkpointing file, we record the initial memory configuration of the VM in a profile file. When the VM is restored at another time, the hypervisor can give back the allocated free pages to the restarted VM by deflating the balloon with the configuration data.

V. Performance Evaluation

We evaluate the VM checkpointing performance with our VM-ME (VM memory exclusion) scheme on a wide range of real applications. To demonstrate the effectiveness of our approach, we make a quantitatively comparison with the native Xen checkpointing mechanism.

A. *Test environments*

Our measurements are conducted on two Intel blade servers, each blade with dual Quad-core Intel Xeon 1.6GHz processors, 4GB RAM, 160GB local SATA disk, and dual Full-duplex Intel Pro/1000 Gbit/s NIC. The blades are interconnected with a Gigabit Ethernet switch and use NFS file system to access a shared disk array. The host machines are running Fedora Core 8 distribution and the hypervisor is Xen 3.2 with Linux 2.6.18.8-Xen kernel. The guest OSes are also running Fedora Core 8 with Linux 2.6.18.8 kernel. The VMs are configured to use one VCPU and a suit of RAM configuration (128MB, 256MB, 512MB and 1024MB) respectively. The experiments use the following VM workloads:

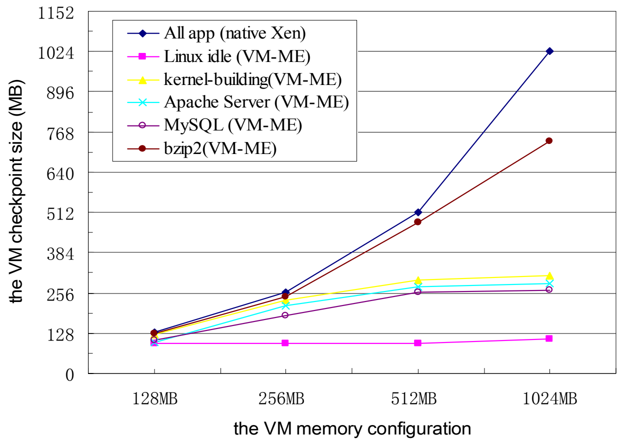

Figure 2. The VM checkpoint size with different memory configurations.

1) Linux idle: an idle Linux OS for daily use.

2) kernel-building: compilation of the complete Linux 2.6.18 kernel with GCC version 4.1.2 and make version 3.81. We configure the compiler using only one thread. This is a balanced workload that tests CPU, memory and disk performance.

3) Apache server: we use the Apache 2.0.63 to measure static-content web server performance. Both clients are configured with 50 simultaneous connections and repetitively downloading a 256KB file from the web server.

4) myslq: we use the MySQL 5.0.45 to test sql loads with standard sql-bench.

5) bzip2: we use bzip2 version 1.04 to compress a 512MB archive file , which produce severe CPU and memory pressure.

B. *Checkpoint size*

We measure the size of each VM checkpoint with different memory configurations for the above applications. In Fig. 2, the uppermost curve displays the checkpoint size of all the 5 applications using native Xen checkpointing mechanism. As we mentioned in the section III, the checkpoint size depends on the VM memory configuration, not the applications scenario, so the checkpoint sizes are the same. The measured data shows that the size of an idle Linux checkpoint can be reduced to approximate 100MB, no matter how much RAM the VM is configured to use. As bzip2 is a memory hungry application, it almost consumes all the allocatable memory while running in a VM configured with 128MB, 256MB, and 512MB RAM. So the reduction of the checkpointing size is not intense compared to the other applications. Fig. 2 demonstrates the VM checkpoint size can be significantly reduced with our VM-ME scheme, especially when the VM is configured using a large mount of memory to guarantee application running performance.

From the perspective of resource management, Fig. 2 implies the optimum memory configuration of the virtual machine for each applications scenario. In our test environment, to get better performance, the VM should be configured to use at lest 128MB, 310MB, 285MB, 264MB, and 740MB physical memory for Linux idle, kernel-building, apache server, mysql, and bzip2, respectively.

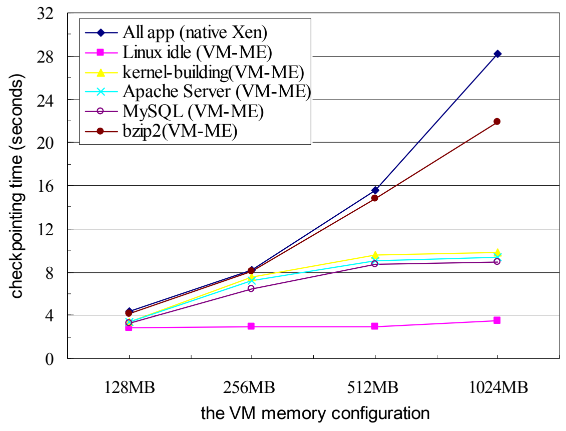

Figure 3. The VM checkpointing time with different memory configurations.

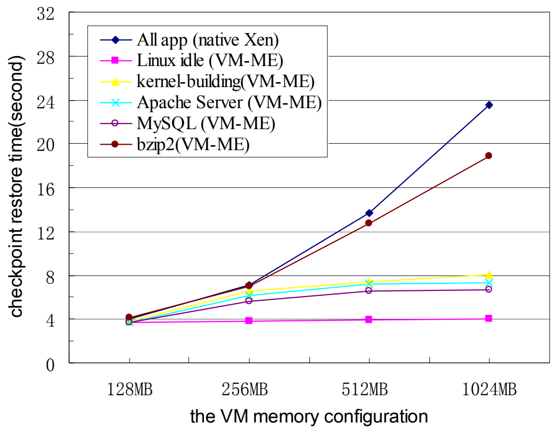

Figure 4. The VM restart time with different memory configurations.

C. *Checkpointing time and restart time*

Fig. 3 and 4 show both the VM checkpointing and restart time are significantly reduced with our VM-ME scheme for the above applications. We find that Fig. 3 and 4 are extremely similar. This demonstrates that the checkpointing and restart times are directly correlated with the checkpoint size. Because most of the cost of checkpointing and restart is to save or load the VM memory image in the local disk, Fig. 3 and 4 also illustrate that checkpointing a running VM costs a little more time than restarting a VM. The reason is not hard to explain, because disk reading has better performance than disk writing.

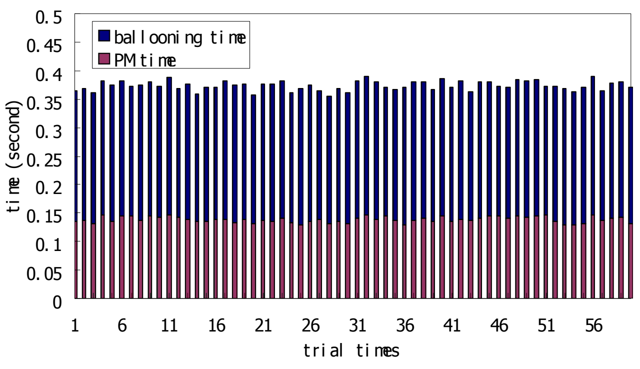

Figure 5. VM memory exclusion overheads.

D. *Overhead of memory exclusion*

VM-ME greatly improves the checkpointing performance, but also introduces a little time overhead undoubtedly. It mainly comprises two parts: the time to collect the memory usage data which related to PM module, and the time to perform memory exclusion which related to ballooning driver. To clearly show how the overheads are generated, we take 60 times experiments for the above application with different VM memory configurations (in each case we repeat the trial 3 times). As shown in Fig. 5, all the VM-ME operation cost less than 0.4 second with a little difference no matter what types of application the VM is running.

VI. CONCLUSIONS

This paper proposes a memory exclusion scheme to optimize the performance of VM checkpointing. By avoiding saving unnecessary free pages in the VM, we can greatly reduce the size of VM checkpoint, and thus reduce the VM checkpointing time and restart time. We implement our prototype in Xen environment. Experimental measurement shows our scheme can greatly improve the checkpoint performance compared against the traditional Xen save & restore approach.

ACKNOWLEDGEMENT

The paper is supported by National 973 Basic Research Program of China under grant No.2007CB310900.

REFERENCES

[1] http://www.xen.org/download/index_3.2.0.html

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization", *Proceedings of the nineteenth ACM symposium on Operating Systems Principles (SOSP'03)*, ACM Press, October 19-22, 2003, Lake George, New York, USA, pp.164-177

[3] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield, “Remus: High Availability via Asynchronous Virtual Machine Replication”, *Proceedings of the 5th USENIX Symposium on Networked Systems Design & Implementation (NSDI'08)*, April 16–18, 2008, San Francisco, CA, USA

[4] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, “Live Migration of Virtual Machines”, *Proceedings of 2nd Symposium on Networked Systems Design and Implementation (NSDI’05)*, May 2-4, 2005, Boston, MA, USA, pp.273-286

[5] Y. Chen, J. S. Plank, and K. Li, “CLIP: A checkpointing tool for message-passing parallel programs”, *Proceedings of High Performance Networking and Computing (SC’97)*, San Jose, CA, November 1997.

[6] D. Li, H. Jin, Y. Shao, and X. Liao, “A High-efficient Inter-Domain Data Transferring System for Virtual Machines”, *Proceedings of The 3rd International Conference on Ubiquitous Information Management and Communication,* ACM Press, January 15-16, 2009, SKKU, Suwon, Korea

[7] M. Hines and K. Gopalan, “Post-Copy Based Live Virtual Machine Migration Using Adaptive Pre-Paging and Dynamic Self-Ballooning”, *Proceedings of the The 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE‘09)*, March 7-11, 2009, Washington, DC, USA

[8] S. T. King, G. W. Dunlap, and P. M. Chen. “Debugging Operating Systems with Time-Traveling Virtual Machines”, *Proceedings of the USENIX Annual Technical Conference (USENIX'05)*, April 10-15, 2005, Anaheim, CA, USA

[9] D. Marques, G. Bronevetsky, R. Fernandes, K. Pingali and P. Stodghill, “Optimizing Checkpoint Sizes in the C3 System”, *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS’05)*, April 3-8, 2005, Denver, Colorado, USA

[10] A. B. Nagarajan, F. Mueller, C. Engelmann, and S. L. Scott, “Proactive Fault Tolerance for HPC with Xen Virtualization”, *Proceedings of 21st ACM International Conference on Supercomputing (ICS’07)*, June 16-20, 2007, Seattle, WA, USA, pp.23-32

[11] J. S. Plank, M. Beck, G. Kingsley, and K. Li, “Libckpt: Transparent checkpointing under Unix”, *Proceedings of UsenixWinter 1995 Technical Conference*, January 1995, pp.213-223.

[12] J. S. Plank, Y. Chen, K. Li, M. Beck, and G. Kingsley, “Memory Exclusion: Optimizing the Performance of Checkpointing Systems”, *Software - Practice and Experience*, Vol.29, No.2, February 1999, pp.125-142

[13] J. C. Sancho, F. Petrini, K. Davis, R. Gioiosa, and S. Jiang, “Current Practice and a Direction Forward in Checkpoint/Restart Implementations for Fault Tolerance”, *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS’05)*, April 3-8, 2005, Denver, Colorado, USA

[14] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum, “Optimizing the Migration of Virtual Computers”, *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI’02)*, December 8-11, 2002, Boston, MA, USA

[15] C. A. Waldspurger, “Memory Resource Management in VMware ESX Server”, *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI’02)*, ACM Press, December 8-11, 2002, Boston, MA, USA, pp.181-194