Fast Software Rejuvenation of Virtual Machine Monitors

Kenichi Kourai, *Member, IEEE Computer Society,* and Shigeru Chiba

Abstract—As server consolidation using virtual machines (VMs) is carried out, *software aging* of virtual machine monitors (VMMs) is becoming critical. Since a VMM is fundamental software for running VMs, its performance degradation or crash failure affects all VMs running on top of it. To counteract such software aging, a proactive technique called *software rejuvenation* has been proposed. A simple example of rejuvenation is to reboot a VMM. However, simply rebooting a VMM is undesirable because that needs rebooting operating systems on all VMs. In this paper, we propose a new technique for fast rejuvenation of VMMs called the *warm-VM reboot*. The warm-VM reboot enables efficiently rebooting only a VMM by suspending and resuming VMs without saving the memory images to persistent storage. To achieve this, we have developed two mechanisms: *on-memory suspend/resume* of VMs and *quick reload* of a VMM. Compared with a normal reboot, the warm-VM reboot reduced the downtime by 74% at maximum. It also prevented the performance degradation due to cache misses after the reboot, which was 52% in case of a normal reboot. In a cluster environment, the warm-VM reboot achieved higher total throughput than the system using VM migration and a normal reboot.

Index Terms—Operating systems, checkpoint/restart, main memory, availability, performance.

1 Introduction

The phenomenon that the state of running software degrades with time is known as *software aging* [1]. The causes of this degradation are the exhaustion of system resources and data corruption. This often leads to performance degradation of running software or crash failure. Software aging has been studied in the UNIX operating system [2] and the Apache web server [3], [4]. Recently, software aging in virtual machine monitors (VMMs) is becoming critical as server consolidation using virtual machines (VMs) is being widely carried out. A VMM is fundamental software that multiplexes physical resources such as CPU and memory to the VMs running on top of it. Since many VMs run on one machine consolidating multiple servers, aging of the VMM directly affects all the VMs.

To counteract such software aging, a proactive technique called *software rejuvenation* has been proposed [1]. Software rejuvenation occasionally stops running software, cleans its internal state, and restarts it. A well-known and simple example of rejuvenation is a system reboot [5]. This technique can be applied to rejuvenate a VMM because a VMM is similar software to an operating system kernel, which can be rejuvenated by a system reboot [2]. However, operating systems running on the VMs built on top of a VMM also have to be rebooted when the VMM is rejuvenated. This increases the downtime of services provided by the operating systems. It takes a long time to reboot many operating systems in parallel when the VMM is rebooted. After the operating systems are rebooted with the VMM, their performance can be degraded due to cache misses. The file cache used by the operating systems is lost by the reboot. Such downtime and performance degradation are critical for servers.

This paper proposes a new technique for fast rejuvenation of VMMs called the *warm-VM reboot*. The basic idea is that a VMM preserves the memory images of all VMs through the reboot of the VMM and reuses those memory images after the reboot. The warm-VM reboot enables efficiently rebooting only a VMM by using the *on-memory suspend/resume* mechanism of VMs and the *quick reload* mechanism of a VMM. Using the on-memory suspend/resume mechanism, the VMM suspends VMs running on it before it is rebooted. At that time, the memory images of the VMs are preserved on main memory and they are not saved to any persistent storage. The suspended VMs are quickly resumed by directly using the preserved memory images after the reboot. To preserve the memory images during the reboot, the VMM is rebooted without a hardware reset using the quick reload mechanism. The warm-VM reboot can reduce the downtime of operating systems running on VMs and prevent performance degradation due to cache misses because it does not need to reboot the operating systems.

To achieve this fast rejuvenation, we have developed *RootHammer* based on Xen [6]. According to our experimental results, the warm-VM reboot reduced the downtime due to rejuvenating the VMM by 74% at maximum, compared with rebooting the VMM normally.

- *K. Kourai is with the Department of Creative Informatics, Kyushu Institute of Technology, 680-4 Kawazu, Iizuka, Fukuoka 820–8502, Japan. Email: kourai@ci.kyutech.ac.jp*
- *S. Chiba is with the Department of Mathematical and Computing Sciences, Tokyo Institute of Technology, 2-12-1 Ookayama, Meguro-ku, Tokyo 152–8552, Japan. Email: chiba@is.titech.ac.jp*

Manuscript received 15 Feb. 2008; revised 18 May 2009; accepted 25 Jan. 2010; published online xx 2010.
For information on obtaining reprints of this article, please send e-mail to: tdsc@computer.org, and reference IEEECS Log Number xxx.

After the warm-VM reboot, the throughput of a web server was not degraded because the file cache was restored completely. When we rebooted the VMM normally, the throughput of a web server was degraded by 52% just after the reboot. In addition, we estimated the availability, based on the time-based rejuvenation [7], when we considered not only the rejuvenation of the VMM but also that of operating systems in the VMs.

Moreover, we show the usefulness of our warm-VM reboot in a cluster environment. Since the service downtime becomes zero when multiple hosts in a cluster provide the same service, we consider the total throughput of a cluster instead of availability. The total throughput can include the impacts of both the downtime and performance degradation of VMs. We also compare our warm-VM reboot with the migration of VMs. The migration technique can move all VMs to another host before the rejuvenation of the VMM and reduce the downtime. From our experimental results, the warm-VM reboot achieved the highest total throughput.

This paper substantially extends our previous work [8] as follows:

- We define the availability for the systems using our warm-VM reboot and the traditional approach when both a VMM and operating systems running on VMs are rejuvenated. In our previous work, we defined only the downtime due to the rejuvenation of a VMM and did not define the availability explicitly.
- We define the total throughput in a cluster environment for the systems using the warm-VM reboot, the traditional approach, and VM migration. Moreover, we perform its quantitative analysis through experiments. Our previous work did not define it and performed only qualitative analysis.

The rest of this paper is organized as follows. Section 2 describes the problems of current software rejuvenation of VMMs. Section 3 presents a new technique for fast rejuvenation of VMMs and defines the availability when using it. Section 4 explains our implementation based on Xen and Section 5 shows our experimental results. Section 6 shows the usefulness of the warm-VM reboot in a cluster environment. Section 7 examines related work and Section 8 concludes the paper.

2 Software Rejuvenation of VMMs

As server consolidation using VMs is widely carried out, *software aging* of VMMs is becoming critical. Recently, multiple server machines are consolidated into one machine using VMs. In such a machine, many VMs are running on top of a VMM. Since a VMM is long-running software and is not rebooted frequently, the influences due to software aging accumulate more easily than the other components. For example, a VMM may leak its memory by failing to release a part of memory. In Xen [6], the size of the heap memory of the VMM is only 16 MB by default in spite of the size of physical memory. If the VMM leaks its heap memory, it would become out of memory easily. Xen had a bug that caused available heap memory to decrease whenever a VM was rebooted [9] or when some error paths were executed [10]. Out-of-memory errors can lead performance degradation or crash failure of the VMM. Such problems of the VMM directly affect all the VMs.

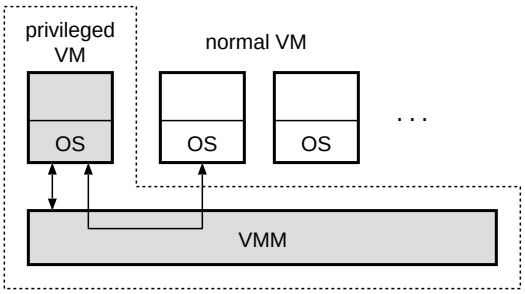

Fig. 1. The assumed VM architecture.

In addition to the aging of VMMs, that of privileged VMs can also affect the other VMs. Privileged VMs are used in some VM architectures such as Xen and VMware ESX server [11] to help the VMM for VM management and/or I/O processing of all VMs, as shown in Fig. 1. They run normal operating systems with some modifications. For operating systems, it has been reported that system resources such as kernel memory and swap spaces were exhausted with time [2] and the exhaustion also depended on system workload [12], [13]. In privileged VMs, memory exhaustion easily occurs because privileged VMs do not run large servers and therefore the typical size of the memory allocated to them is not so large. For example, Xen had a bug of memory leaks in its daemon named xenstored running on a privileged VM [14]. If I/O processing in the privileged VM slows down due to out of memory, the performance in the other VMs is also degraded. Since xenstored is not restartable, restoring from such memory leaks needs to reboot the privileged VM. Furthermore, the reboot of the privileged VM causes the VMM to be rebooted because the privileged VM strongly depends on the VMM. For this reason, we consider such privileged VMs as a part of a VMM and we do not count them as normal VMs.

Software rejuvenation [1] is a proactive technique to counteract such software aging. Since the state of long-running software such as VMMs degrades with time under aging conditions, preventive maintenance by software rejuvenation would decrease problems due to aging. Although the essence of rejuvenation is to clean the internal state of software, a system reboot is well-known as its simple example [5]. However, when a VMM is rejuvenated, operating systems on the VMs built on top of the VMM also have to be rebooted. Operating systems running on VMs have to be shut down to keep the integrity before the VMM terminates the VMs. Then, after the reboot of the VMM, newly created VMs have to boot the operating systems and restart all services again.

This increases the downtime of services provided by

operating systems. First of all, many operating systems are shut down and booted in parallel when the VMM is rebooted. The time for rebooting each operating system is proportional to the number of VMs because shutting down and booting multiple operating systems in parallel cause resource contention among them. The number of VMs that can run simultaneously is increasing due to processor support of virtualization such as Intel VT [15] and AMD-V [16] and multicore processors. In addition, recent servers tend to provide heavyweight services such as the JBoss application server [17] and the time for stopping and restarting services is increasing. Second, shutting down operating systems, rebooting the VMM, and booting operating systems are performed sequentially. The in-between reboot of the VMM increases the service downtime. The reboot of the VMM includes shutting down the VMM, resetting hardware, and booting the VMM. In particular, a hardware reset involves power-on self-test by the BIOS such as a time-consuming check of large amount of main memory and SCSI initialization.

Furthermore, the performance of operating systems on VMs is degraded after they are rebooted with the VMM. The primary cause is to lose the file cache. An operating system stores file contents in main memory as the file cache when it reads them from storage. An operating system speeds up file accesses by using the file cache on memory. When an operating system is rebooted, main memory is initialized and the file cache managed by the operating system is lost. Therefore, just after the reboot of the operating system, the execution performance of server processes running on top of it is degraded due to frequent cache misses. To fill the file cache after the reboot, an operating system needs to read necessary files from storage. Since modern operating systems use most of free memory as the file cache, it takes a long time to fill free memory with the file cache. The size of memory installable to one machine tends to increase due to 64-bit processors and cheaper memory modules. Consequently, more memory is allocated to each VM.

3 Fast Rejuvenation Technique

We claim that only a VMM should be rebooted when only the VMM needs rejuvenation. In other words, rebooting operating systems should be independent of rebooting an underlying VMM. Although an operating system may be rejuvenated occasionally as well as a VMM, the timing does not always the same as that of the rejuvenation of a VMM. If some operating systems do not need to be rejuvenated when the VMM is rejuvenated, rebooting these operating systems is just wasteful.

3.1 Warm-VM Reboot

To minimize the influences of the rejuvenation of VMMs, we propose a new technique for fast rejuvenation. The technique called the *warm-VM reboot* enables a VMM to preserve the memory images of all the VMs through its reboot and to reuse those memory images after the reboot. The warm-VM reboot consists of two mechanisms: *on-memory suspend/resume* of VMs and *quick reload* of a VMM. The VMM suspends all VMs using the on-memory suspend mechanism before it is rebooted, reboots itself by the quick reload mechanism, and resumes all VMs using the on-memory resume mechanism after the VMM is rebooted. Using these mechanisms, the warm-VM reboot enables rebooting only a VMM.

The on-memory suspend mechanism simply *freezes* the memory image used by a VM as it is. The memory image is preserved on memory through the reboot of the VMM until the VM is resumed. This mechanism needs neither to save the image to any persistent storage such as disks nor to copy it to non-volatile memory such as flash memory. This is very efficient because the time needed for suspend hardly depends on the size of memory allocated to the VM. Even if the total memory size of all VMs becomes larger, the on-memory suspend mechanism can scale. At the same time, this mechanism saves the execution state of the suspended VM to the memory area that is also preserved through the reboot of the VMM.

On the other hand, the on-memory resume mechanism *unfreezes* the frozen memory image to restore the suspended VM. The frozen memory image is preserved through the reboot of the VMM by the quick reload mechanism. The on-memory resume mechanism also needs neither to read the saved image from persistent storage nor to copy it from non-volatile memory. Since the memory image of the VM, including the file cache, is restored completely, performance degradation due to cache misses is prevented even just after the reboot. At the same time, the saved execution state of a VM is also restored. These mechanisms are analogous to the ACPI S3 state (Suspend To RAM) [18] in that they can suspend and resume a VM without touching its memory image on main memory.

The quick reload mechanism preserves the memory images of VMs through the reboot of a VMM and furthermore makes the reboot itself faster. Usually, rebooting a VMM needs a hardware reset to reload a VMM instance, but a hardware reset does not guarantee that memory contents are preserved during it. In addition, a hardware reset takes a long time as described in the previous section. The quick reload mechanism can bypass a hardware reset by loading a new VMM instance by software and start it by jumping to its entry point. Since the software mechanism can manage memory during the reboot, it is guaranteed that memory contents are preserved. Furthermore, the quick reload mechanism prevents the frozen memory images of VMs from being corrupted while the VMM initializes itself.

Although many VMMs provide suspend/resume mechanisms, they are not suitable for the rejuvenation of VMMs because they have to use disks as persistent storage to save memory images. These traditional suspend/resume mechanisms are analogous to the ACPI S4 state (Suspend To Disk), so-called *hibernation*. These

mechanisms need heavy disk accesses and they are too slow. On the other hand, our on-memory suspend/resume mechanism does not need to save the memory images to persistent storage. The memory images on volatile main memory can be reused.

3.2 Rejuvenation Model and Availability

We define the downtime due to the rejuvenation of a VMM. When the warm-VM reboot is used, the downtime is caused by suspending all VMs, rebooting the VMM, and resuming all VMs. The downtime $D_w(n)$ is the sum of the times needed for performing on-memory suspend and resume of n VMs in parallel and for rebooting a VMM. In the warm-VM reboot, rebooting a VMM depends on the number of VMs because the VMM does not need to initialize the memory area reserved for suspended VMs during its reboot.

For comparison, we consider the rejuvenation of a VMM by a normal reboot, which we described in Section 2. We call this the *cold-VM reboot* in contrast to the warm-VM reboot. When the cold-VM reboot is used, the downtime is caused by shutting down all operating systems, resetting hardware, rebooting a VMM, and booting all operating systems. The downtime $D_c(n)$ is the sum of the times needed for shutting down and booting n operating systems in parallel, for rebooting a VMM without preserving any VMs, and for a hardware reset.

To define the availability of both the warm-VM reboot and the cold-VM reboot, let us consider the rejuvenation model. Usually the rejuvenation of a VMM (VMM rejuvenation) is used with the rejuvenation of operating systems (OS rejuvenation) running on VMs. For simplicity, we assume that each operating system is rejuvenated by relying on the time elapsed since the last OS rejuvenation, which is called *time-based rejuvenation* [7]. We define the interval of the VMM rejuvenation as T_{vmm} and that of the OS rejuvenation as T_{os}.

When the warm-VM reboot is used, the VMM rejuvenation can be performed independently of the OS rejuvenation. This is because the warm-VM reboot does not involve the OS rejuvenation. Fig. 2 shows the rejuvenation timing when $T_{vmm} > T_{os}$ and $T_{vmm} \leq T_{os}$. On the other hand, when the cold-VM reboot is used, the VMM rejuvenation affects the timing of the OS rejuvenation because the VMM rejuvenation involves the OS rejuvenation. Fig. 3a shows the rejuvenation timing when $T_{vmm} > T_{os}$. Since the OS rejuvenation is performed at the same time of the VMM rejuvenation, the timer for the OS rejuvenation is reset and the OS rejuvenation is rescheduled. When $T_{vmm} \leq T_{os}$, the OS rejuvenation is always performed together with the VMM rejuvenation as shown in Fig. 3b. Before the timer for the OS rejuvenation reaches T_{os}, the VMM rejuvenation is performed and the timer is reset.

Using the above rejuvenation model, we define the availability. To focus on software rejuvenation, we ignore

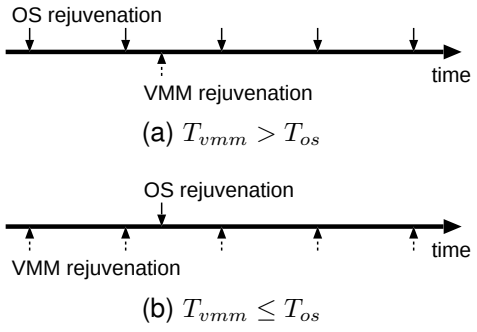

Fig. 2. The timing of two kinds of rejuvenation when using the warm-VM reboot.

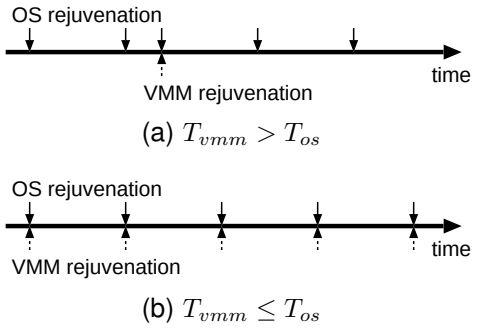

Fig. 3. The timing of two kinds of rejuvenation when using the cold-VM reboot.

software and hardware failures. Integrating these into our model is not difficult. First, let us consider the total downtime in T_{vmm}. In the system using the warm-VM reboot, the total downtime is $D_w(n)+N_w D_{os}$, where D_{os} is the time needed for rebooting one operating system for the OS rejuvenation. N_w is the average number of the OS rejuvenation performed in T_{vmm} and defined by $(T_{vmm}-D_w(n))/T_{os}$. We assume that the timer for the OS rejuvenation is stopped during the VMM rejuvenation. Therefore, the availability A_w is defined by

$$A_w = 1 - \frac{D_w(n) + N_w D_{os}}{T_{vmm}}. \quad (1)$$

In the system using the cold-VM reboot, on the other hand, the total downtime in T_{vmm} is $D_c(n) + \lfloor N_c \rfloor D_{os}$, where N_c is defined by $(T_{vmm} - D_c(n))/T_{os}$. $\lfloor N_c \rfloor$ is the exact number of the OS rejuvenation solely performed in T_{vmm}. Since the timer for the OS rejuvenation is reset by the VMM rejuvenation in the cold-VM reboot, the average number of the OS rejuvenation is an integer less than N_c. For simplicity, we assume that the VMM rejuvenation is not triggered during the OS rejuvenation. Note that the OS rejuvenation performed together with the VMM rejuvenation is not counted in N_c. Its downtime is included in $D_c(n)$. Therefore, the availability A_c

TABLE 1
The states rejuvenated by the warm-VM reboot.

state	warm-VM reboot	OS reboot
(VMM)		
memory allocation	x	
heap memory	x	
VM memory		x
VM execution state		x
VM configuration		x
P2M-mapping table		
(privileged VM)		
whole state	x	
(normal VM)		
whole state		x

is defined by

$$A_c = 1 - \frac{D_c(n) + \lfloor N_c \rfloor D_{os}}{T_{vmm}}. \quad (2)$$

3.3 Rejuvenated States

The warm-VM reboot does not rejuvenate several states in the whole system to achieve fast rejuvenation of VMMs. However, most of such states are rejuvenated by rebooting operating systems running on VMs, which is performed independently of the warm-VM reboot. In other words, the warm-VM reboot enables separating the OS rejuvenation from the VMM rejuvenation. Table 1 shows rejuvenated states by the warm-VM reboot and rebooting operating systems. The VM architecture we assume consists of a VMM, a privileged VM, and normal VMs, as illustrated in Fig. 1. Note that the cold-VM reboot can rejuvenate all the states by rebooting all these components in the system.

For VMs, the warm-VM reboot rejuvenates all the states inside the privileged VM. The operating systems running on the privileged VMs are rebooted normally and all services are restarted. On the other hand, all the states inside normal VMs are not rejuvenated. The warm-VM reboot suspends normal VMs as is and resumes them after rebooting the VMM. Those states are rejuvenated when the operating systems are rebooted.

For a VMM, the warm-VM reboot rejuvenates most of the states. For example, the data structures for memory allocation are initialized when the VMM is rebooted. The heap memory allocated in the VMM is reclaimed. However, the memory allocated to normal VMs is not reclaimed because it is preserved through the warm-VM reboot. The memory is reclaimed when the operating systems are rebooted. When the operating system running on a normal VM is rebooted, the VM is destroyed and a new VM is created, at least, in Xen. On the VM destruction, the whole memory allocated is reclaimed. Likewise, the execution state and configuration of VMs are discarded when the operating systems are rebooted.

The P2M-mapping table, on the other hand, is not rejuvenated by either the warm-VM reboot or the reboot of operating systems. It is a table for maintaining memory pages allocated to VMs. Its detail is described in Section 4.1. The table is preserved through both the warm-VM reboot and the reboot of operating systems and is not initialized after the first boot of the VMM. However, the table is a simple one-dimensional array of fixed size and is not allocated dynamically. Therefore, its memory is not likely to leak and the data structure is not likely to be corrupted.

4 Implementation

To achieve the warm-VM reboot, we have developed *RootHammer* based on Xen 3.0.0. In Xen, a VM is called a *domain*. In particular, the privileged VM that manages VMs and handles I/O is called *domain 0* and the other VMs are called *domain Us*.

4.1 Memory Management of the VMM

The VMM distinguishes machine memory and pseudo-physical memory to virtualize memory resource. Machine memory is physical memory installed in the machine and consists of a set of machine page frames. For each machine page frame, a machine frame number (MFN) is consecutively numbered from 0. Pseudo-physical memory is the memory allocated to domains and gives the illusion of contiguous physical memory to domains. For each physical page frame in each domain, a physical frame number (PFN) is consecutively numbered from 0.

The VMM creates the *P2M-mapping table* to enable domains to reuse its memory even after the reboot. The table is a one-dimensional array that records mapping from PFNs to MFNs. A new mapping is created in this table when a new machine page frame is allocated to a domain while an existing mapping is removed when a machine page frame is deallocated from a domain. These mappings are preserved after domains are suspended.

For each domain, the VMM allocates a contiguous area in the table. If the user repeats creating and destroying domains of various sizes, memory fragmentation may occur in the table. In this case, the user cannot create new domains any more even if there is enough machine memory unused. To avoid this situation, the VMM performs memory compaction when it cannot allocate a contiguous area in the table. Although Xen originally maintains a similar table, its format is the same as a page table. That format is more complicated than a simple array and error-prone.

4.2 On-memory Suspend/Resume Mechanism

When the operating system in domain 0 is shut down, the VMM suspends all domain Us as in Fig. 4. To suspend domain Us, the VMM sends a suspend event to each domain U. In the original Xen, domain 0, not the VMM, sends the event to each domain U. One advantage of suspending by the VMM is that suspending domain Us can be delayed until after the operating system in domain 0 is shut down. The original suspend by domain

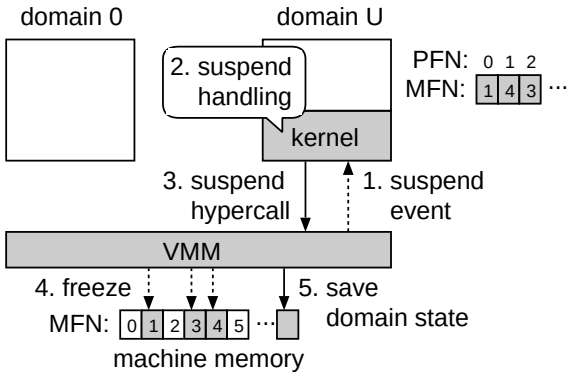

Fig. 4. On-memory suspend of a domain U.

0 has to be performed while domain 0 is shut down. This delay reduces the downtime of services running in a domain U. When a domain U receives the suspend event, the operating system kernel in the domain U executes its suspend handler and suspends all virtual devices. We used the handler implemented in the Linux kernel modified for Xen. In Xen, device drivers called front-end drivers in domain Us manage virtual devices. The front-end drivers communicate with device drivers called back-end drivers running on domain 0. Thanks to this splitting device-driver model, virtual devices can be easily detached from the operating systems in domain Us. On suspend, front-end drivers do not communicate with back-end drivers because the back-end drivers are stateless.

After the operating system in a domain U executes the suspend handler, it issues the suspend hypervisor call to the VMM, which is like a system call to the operating system. In the hypervisor call, the VMM freezes the memory image of the domain on memory by reserving it. The VMM does not release the memory pages allocated to the domain but it maintains them using the P2M-mapping table. This does not cause out-of-memory errors because the VMM is rebooted just after it suspends all domain Us. Next, the VMM saves the execution state of the domain to the memory pages that is preserved during the reboot of the VMM. The execution state of a domain includes execution context such as CPU registers and shared information such as the status of event channels. In addition, the VMM saves the configuration of the domain, such as devices. The memory space needed for saving those is 16 KB.

After the VMM finishes suspending all domain Us, the VMM is rebooted without losing the memory images of domain Us by using the quick reload mechanism, which is described in the next section. Then, after domain 0 is rebooted, it resumes all domain Us. First, domain 0 creates a new domain U, allocates the memory pages recorded in the P2M-mapping table to the domain U, and restores its memory image. Next, the VMM restores the state of the domain U from the saved state. The operating system kernel in the domain U executes the resume handler to re-establish the communication channels to the VMM and to attach the virtual devices that were detached on suspend. This re-attachment is to connect the front-end drivers with the corresponding back-end drivers. Finally, the execution of the domain U is restarted.

4.3 Quick Reload Mechanism

To preserve the memory images of domain Us during the reboot of a VMM, we have implemented the quick reload mechanism based on the kexec mechanism [19] provided in the Linux kernel. The kexec mechanism enables a new kernel to be started without a hardware reset. Like kexec, the quick reload mechanism enables a new VMM to be started without a hardware reset. To load a new VMM instance into the current VMM, we have implemented the xexec system call in the Linux kernel for domain 0 and the xexec hypervisor call in the VMM.

When the xexec system call is issued in domain 0, the kernel issues the xexec hypervisor call to the VMM. This hypervisor call loads a new executable image consisting of a VMM, a kernel for domain 0, and an initial RAM disk for domain 0 into memory. When the VMM is rebooted, the quick reload mechanism first passes the control to the CPU used at the boot time. Then, it copies the executable loaded by the xexec hypervisor call to the address where the executable image is loaded at normal boot time. Finally, the mechanism transfers the control to the new VMM.

When the new VMM is rebooted and initialized, it first reserves the memory for the P2M-mapping table. Based on the table, the VMM reserves the memory pages that have been allocated to domain Us. Next, the VMM reserves the memory pages where the execution state of domains is saved. To make the VMM reserve these memory pages, the xexec hypervisor call specifies a boot option to the new VMM. The latest Xen also supports the kexec mechanism for the VMM, but it does not have any support for preserving the memory images of domain Us while a new VMM is initialized.

5 Experiments

We performed experiments to show that our technique for fast rejuvenation is effective. For a server machine, we used a PC with two Dual-Core Opteron processors Model 280 (2.4 GHz), 12 GB of PC3200 DDR SDRAM memory, a 36.7 GB of 15,000 rpm SCSI disk (Ultra 320), and Gigabit Ethernet NICs. We used our RootHammer VMM and, for comparison, the original VMM of Xen 3.0.0. The operating systems running on top of the VMM were Linux 2.6.12 modified for Xen. One physical partition of the disk was used for a virtual disk of one VM. The size of the memory allocated to domain 0 was 512 MB. For a client machine, we used a PC with dual Xeon 3.06 GHz processors, 2 GB of memory, and Gigabit Ethernet NICs. The operating system was Linux 2.6.8. These two PCs were connected with a Gigabit Ethernet

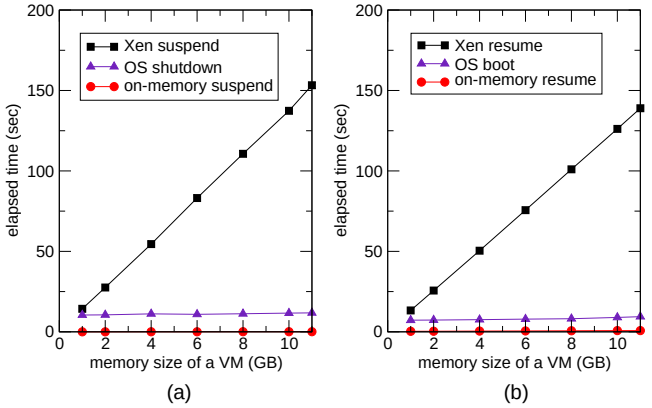

Fig. 5. The time for pre- and post-reboot tasks when the memory size of a VM is changed.

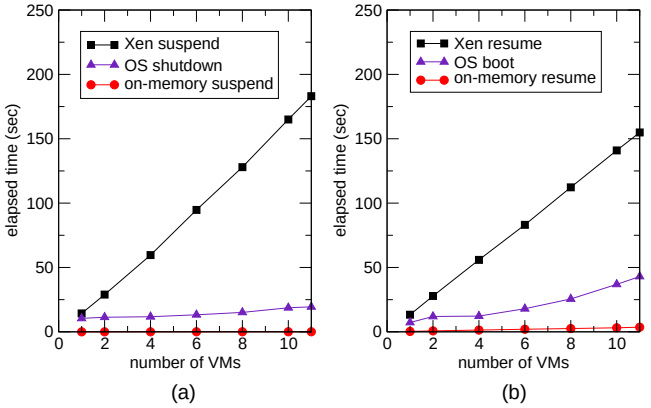

Fig. 6. The time for pre- and post-reboot tasks when the number of VMs is changed.

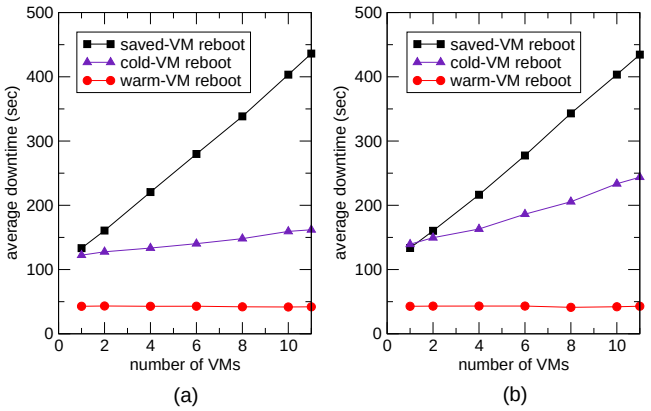

Fig. 7. The downtime of ssh and JBoss when the number of VMs is changed.

switch. We performed five runs for each experiment and show the average.

5.1 Performance of On-memory Suspend/Resume

We measured the time needed for tasks before and after the reboot of the VMM: suspend or shutdown, and resume or boot. We ran the ssh server in each VM as a service. We used OpenSSH version 4.0p1. We performed this experiment for (1) our on-memory suspend/resume, (2) Xen's suspend/resume, which uses a disk to save the memory images of VMs, and (3) simple shutdown and boot.

First, we changed the size of memory allocated to a single VM from 1 to 11 GB and measured the time needed for pre- and post-reboot tasks. Fig. 5 shows the results. Xen's suspend/resume depended on the memory size of a VM because this method must write the whole memory image of a VM to a disk and read it from the disk. On the other hand, our on-memory suspend/resume hardly depended on the memory size because this method does not touch the memory image of a VM. When the memory size was 11 GB, it took 0.08 seconds for suspend and 0.8 second for resume. These are only 0.05% and 0.6% of Xen's suspend and resume, respectively.

Next, we measured the time needed for pre- and post-reboot tasks when multiple VMs were running in parallel. We fixed the size of memory allocated to each VM to 1 GB and changed the number of VMs from 1 to 11. Domain 0 is not included in the number. Fig. 6 shows the results. All the three methods depended on the number of VMs. When the number of VMs was 11, on-memory suspend/resume needed only 0.04 seconds for suspend and 3.6 seconds for resume. These were 0.02% and 2.3% of Xen's suspend and resume, respectively. The result also shows that the time for the boot largely increases as the number of VMs increases.

5.2 Effect of Quick Reload

To examine how fast the VMM is rebooted by using the quick reload mechanism, we measured the time needed for rebooting the VMM. We recorded the time when the execution of a shutdown script completed and when the reboot of the VMM completed. The time between them was 11 seconds when we used quick reload whereas it was 59 seconds when we used a hardware reset. Thus, the quick reload mechanism speeded up the reboot of the VMM by 48 seconds.

5.3 Downtime of Networked Services

We measured the downtime of networked services when we rejuvenated the VMM. We rebooted the VMM while we periodically sent requests from a client host to the VMs on a server host. We measured the time from when a networked service in each VM was down until it was up again after the VMM was rebooted. We performed this experiment for (1) the warm-VM reboot, (2) the cold-VM reboot using a normal reboot, and (3) the reboot with Xen's suspend/resume (*saved-VM reboot*). We fixed the size of memory allocated to each VM to 1 GB and changed the number of VMs from 1 to 11.

First, we ran only the ssh server in each VM and measured its downtime during the reboot of the VMM. Fig. 7a shows the downtime. The downtime due to the saved-VM reboot highly depended on the number of

VMs. When the number was 11, the average downtime was 436 seconds and the 95% confidence interval was (433, 440). At the same number of VMs, the downtime due to the warm-VM reboot was 41.7 seconds and only 9.7% of the saved-VM reboot. The 95% confidence interval was (41.2, 42.1). In addition, the downtime due to the warm-VM reboot hardly depended on the number of VMs. On the other hand, the downtime due to the cold-VM reboot was 162 seconds and the 95% confidence interval was (160, 163) when the number of VMs was 11. This was 3.9 times longer than the downtime due to the warm-VM reboot.

After we rebooted the VMM using the warm-VM reboot or the saved-VM reboot, we could continue the session of ssh thanks to the TCP retransmission mechanism, even if a timeout was set in the ssh server. However, if a timeout was set to 60 seconds in the ssh client, the session was timed out during the saved-VM reboot. From this point of view, the downtime for one reboot should be short enough. When we used the cold-VM reboot, we could not continue the session because the ssh server was shut down.

Next, we ran the JBoss application server [17] and measured its downtime during the reboot of a VMM. JBoss is a large server and it takes a longer time to start than a ssh server. We used JBoss version 4.0.5.GA with the default configuration and Sun JDK version 1.5.0_09. Fig. 7b shows the downtime. The downtime due to the warm-VM reboot and the saved-VM reboot was almost the same as that of a ssh server because these reboot mechanisms resumed VMs and did not need to restart the JBoss server. On the other hand, the downtime due to the cold-VM reboot was larger than that of a ssh server because the cold-VM reboot needed to restart the JBoss server. When the number of VMs was 11, the downtime was 243 seconds and the 95% confidence interval was (240, 247). This was 1.5 times longer than that of a ssh server. This means that the cold-VM reboot increases the service downtime according to running services.

5.4 Downtime Analysis

To examine which factors reduce the downtime in the warm-VM reboot, we measured the time needed for each operation when we rebooted the VMM. At the same time, we measured the throughput of a web server running on a VM. We repeated sending requests from a client host to the Apache web server [20] running on a VM in a server host by using the httperf benchmark tool [21]. We used Apache version 2.0.54 and httperf version 0.8. We created 11 VMs and allocated 1 GB of memory to each VM. We rebooted the VMM and recorded the changes of the throughput. We performed this experiment for the warm-VM reboot and the cold-VM reboot five times and made sure that the results were almost the same. Fig. 8 shows the results at one of the runs. We superimposed the time needed for each operation during the reboot onto Fig. 8. We executed the reboot command in domain 0 at time 20 seconds in this figure.

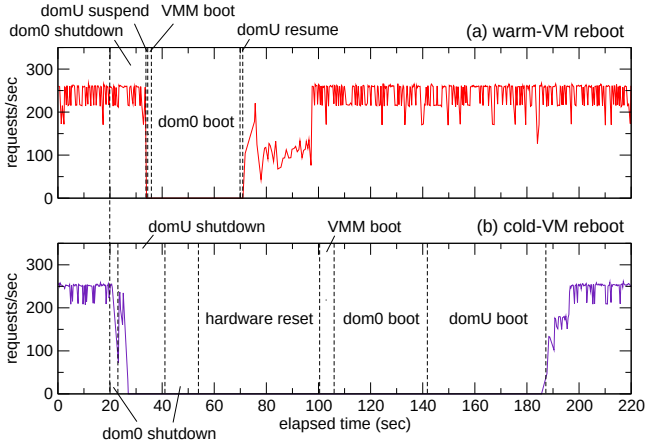

Fig. 8. The breakdown of the downtime due to the VMM rejuvenation.

As shown in the previous section, the on-memory suspend/resume mechanism provided by the warm-VM reboot reduced the downtime largely. The total time for on-memory suspend/resume was 4 seconds, but that for shutdown and boot in the cold-VM reboot was 63 seconds. In addition, the warm-VM reboot reduced the time for a hardware reset from 43 to 0 second. Also, the fact that the warm-VM reboot can continue to run a web server until just before the VMM is rebooted was effective for reducing downtime. A web server was stopped at time 34 seconds in the warm-VM reboot while it was stopped at time 27 seconds in the cold-VM reboot. This reduced downtime by 7 seconds. For the warm-VM reboot, the VMM is responsible for suspending VMs and it can do that task after domain 0 is shut down.

In both cases, the throughput was restored after the reboot of the VMM. However, the throughput in the cold-VM reboot was degraded during 8 seconds. This was most likely due to misses of the file cache. We examine this performance degradation in detail in Section 5.6. The throughput in the warm-VM reboot was also degraded during 25 seconds after the reboot. This is not due to cache misses but most likely due to resource contention between VMs. As described in Section 4, when a VM is resumed, the resume handler of the operating system is executed. It is highly possible that the execution of the resume handlers in resuming VMs interferes with handling network packets in already resumed VMs. In fact, there was no performance degradation when we used only one VM.

5.5 Availability

Let us consider the availability of the JBoss server when the number of VMs is 11. As well as A_w and A_c, we consider the availability A_s of the saved-VM reboot. It is $1-(D_s(n)+N_sD_{os})/T_{vmm}$, where $D_s(n)$ is the downtime due to the reboot with Xen's suspend/resume and N_s

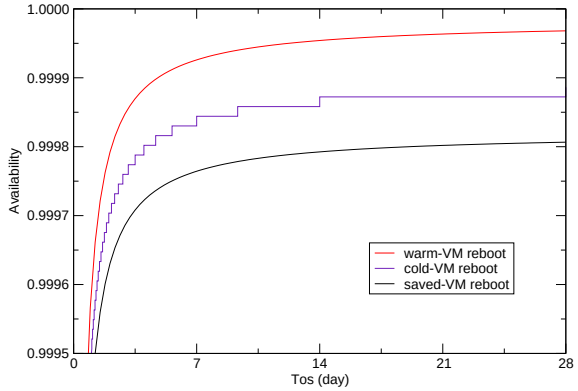

Fig. 9. The availabilities when T_{vmm} is 28 days.

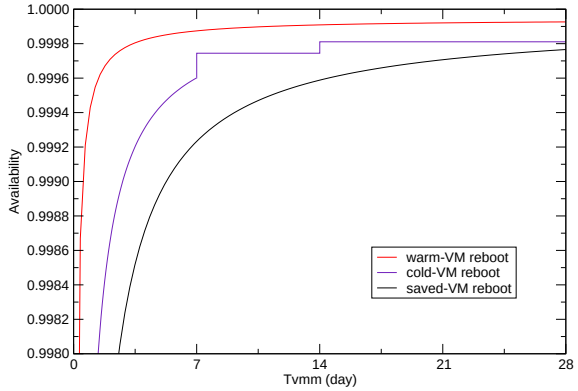

Fig. 10. The availabilities when T_{os} is 7 days.

is the average number of the OS rejuvenation in T_{vmm}. According to our experiment, $D_w(11) = 43$, $D_c(11) = 243$, $D_s(11) = 434$, and $D_{os} = 34$ for the system using JBoss.

Fig. 9 and Fig. 10 plot these availabilities for fixed T_{vmm} and T_{os}, respectively. Fig. 9 fixes T_{vmm} to 28 days and changes T_{os} from 0 to 28 days. The availability of the warm-VM reboot is always the highest and that of the saved-VM reboot is the lowest. On the other hand, Fig. 10 fixes T_{os} to 7 days and changes T_{vmm} from 0 to 28 days. The availability of the warm-VM reboot is the highest. Unlike Fig. 9, the differences between the availabilities are shrinking as T_{vmm} becomes large.

As an example, we consider that the OS rejuvenation is performed every week and the VMM rejuvenation is performed once per four weeks. The availabilities for the warm-VM reboot, the cold-VM reboot, and the saved-VM reboot are 0.99993, 0.99986, and 0.99977, respectively. The warm-VM reboot achieves four 9s although the others achieve three 9s. This improvement of availability is important for critical servers.

5.6 Performance Degradation

To examine performance degradation due to cache misses, we measured the throughput of operations with file accesses in a VM before and after the reboot of a VMM. To examine the effect of the file cache, we

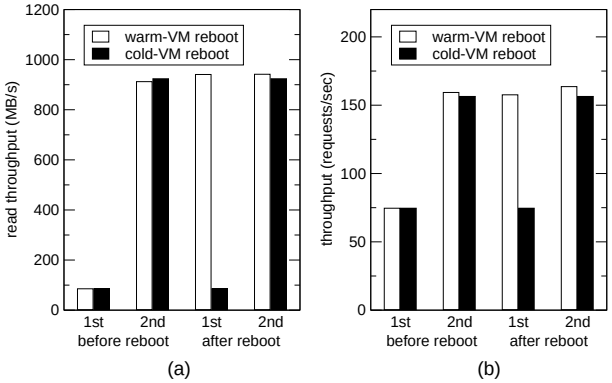

Fig. 11. The throughput of file reads and web accesses before and after the VMM rejuvenation.

measured the throughput of the first- and second-time accesses. We allocated 11 GB of memory to one VM. First, we measured the time needed to read a file of 512 MB and the number of file cache misses. In this experiment, all the file blocks were cached on memory. We performed this experiment for the warm-VM reboot and the cold-VM reboot. Fig. 11a shows the read throughput. When we used the warm-VM reboot, the throughput just after the reboot was not degraded. On the other hand, when we used the cold-VM reboot, the throughput just after the reboot was degraded by 91%, compared with that just before the reboot. This improvement was most likely achieved by no miss in the file cache. In fact, the number of file cache misses was zero.

Next, we measured the throughput of a web server before and after the reboot of a VMM. The Apache web server served 10,000 files of 512 KB, all of which were cached on memory. In this experiment, 10 httperf processes in a client host sent requests to the server in parallel. All files were requested only once. Fig. 11b shows the results. When we used the warm-VM reboot, the performance just after the reboot was not degraded, compared with that just before the reboot. When we used the cold-VM reboot, the throughput just after the reboot was degraded by 52%.

6 Cluster Environment

6.1 Total Throughput

Software rejuvenation is naturally fit with a cluster environment as described in the literature [22], [23]. In a cluster environment, multiple hosts provide the same service and a load balancer dispatches requests to one of these hosts. Even if some of the hosts are rebooted for the rejuvenation of the VMM, the service downtime is zero. However, the total throughput of a cluster is degraded while some hosts are rebooted. The warm-VM reboot can mitigate the performance degradation by reducing the downtime of rebooted hosts.

Let us consider a cluster environment that consists of m hosts to estimate the total throughput of a cluster. We let p denote the sum of the throughputs of all VMs

on one host, which we call *per-host throughput*. When all hosts are running in a cluster, the total throughput is mp, the sum of the per-host throughputs. During the rejuvenation of a VMM on one host, the total throughput is degraded to $(m-1)p$ because the host being rejuvenated cannot provide any services. When we use the warm-VM reboot, the degradation of the total throughput lasts only for short duration. The total throughput is restored to mp soon after the rejuvenation. However, when we use the cold-VM reboot, which is a normal reboot of a VMM, the degradation of the total throughput lasts for longer duration. In addition, the total throughput is degraded due to resource contention and cache misses.

6.1.1 Warm-VM Reboot

To calculate the total throughput, we need the availability and the performance degradation for each VM. The total throughput is the sum of the average throughput for each VM in all hosts providing services. We let ω denote the rate of the performance degradation after the warm-VM reboot ($0 \leq \omega \leq 1$) and R_{vmm} its duration. When suspended VMs are resumed at the warm-VM reboot, their performance is degraded most likely due to resource contention between VMs as described in Section 5.4. The average performance degradation due to the VMM rejuvenation is $\omega R_{vmm}/T_{vmm}$.

On the other hand, the OS rejuvenation can cause performance degradation both due to cache misses just after rebooting an operating system and due to resource contention while the other operating systems are being rebooted. We let R_{os} denote the sum of the duration of these two kinds of performance degradation and δ the average rate of the performance degradation for the duration ($0 \leq \delta \leq 1$). Since the OS rejuvenation is performed N_w times in T_{vmm}, the average performance degradation due to the OS rejuvenation is $N_w \delta R_{os}/T_{vmm}$.

Therefore, the total throughput in the system using the warm-VM reboot is defined by

$$P_w = \left(A_w - \frac{\omega R_{vmm} + N_w \delta R_{os}}{T_{vmm}}\right) mp. \quad (3)$$

6.1.2 Cold-VM Reboot

Like the warm-VM reboot, the performance degradation is also caused by the VMM rejuvenation and the OS rejuvenation. We let ω' denote the rate of the performance degradation due to the cold-VM reboot ($0 \leq \omega' \leq 1$) and R'_{vmm} its duration. ω' and R'_{vmm} are different from ω and R_{vmm}, respectively. Unlike the warm-VM reboot, the cold-VM reboot needs to reboot the operating systems as well as the VMM. At that time, all the operating systems in the VMs are rebooted in parallel and access a disk heavily due to cache misses after the reboots. This factor increases the rate and the duration of the performance degradation. Using these parameters, the average performance degradation due to the VMM rejuvenation is $\omega' R'_{vmm}/T_{vmm}$. On the other hand, the average performance degradation due to the OS rejuvenation is $\lfloor N_c \rfloor \delta R_{os}/T_{vmm}$. For simplicity, we assume that the VMM rejuvenation is not triggered while the performance is degraded just after the OS rejuvenation.

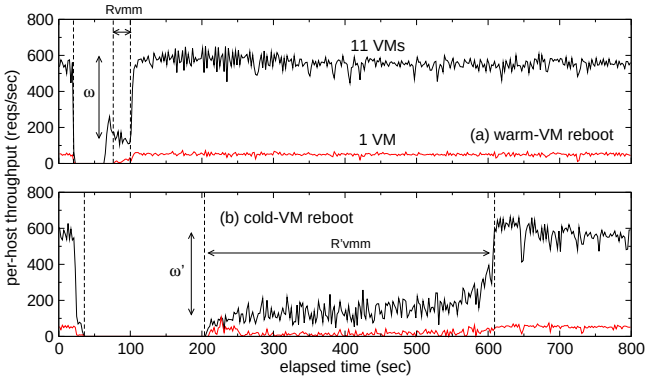

Fig. 12. The changes of the per-host throughputs during the VMM rejuvenation.

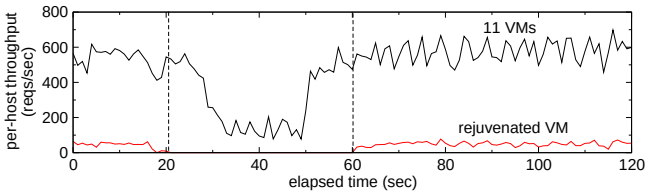

Fig. 13. The change of the per-host throughput during the OS rejuvenation.

Therefore, the total throughput in the system using the cold-VM reboot is defined by

$$P_c = \left(A_c - \frac{\omega' R'_{vmm} + \lfloor N_c \rfloor \delta R_{os}}{T_{vmm}}\right) mp. \quad (4)$$

6.1.3 Experimental Results

To estimate P_w and P_c, we examined the per-host throughput while we performed the VMM rejuvenation and the OS rejuvenation. We used the same experimental setup as that in Section 5, except that we changed the disk of the server machine with a 147 GB of SCSI disk. We created 11 VMs and allocated 1 GB of memory to each VM. The Apache web server ran in each VM and served 5,000 files of 128 KB, all of which were cached on memory of each VM. Note that this workload is different from that in Section 5.6 because the VM configuration is different. We repeated sending requests to each server by using 11 httperf processes. The request rate of each httperf was 50 requests/sec.

Fig. 12 shows the changes of the per-host throughputs when the warm-VM reboot and the cold-VM reboot were performed for the VMM rejuvenation. The per-host throughput p at a normal time was 560 requests/sec. This figure also shows the changes of the throughput of one VM. Fig. 13 shows the change of the per-host throughput when an operating system in a VM is rebooted for the OS rejuvenation.

For the warm-VM reboot, the average downtime $D_w(11)$ was 59 seconds. The per-host throughput was degraded until all the VMs were resumed. It took time for resuming all the VMs most likely due to resource contention between VMs. The rate of the performance degradation ω was 0.61 and its duration R_{vmm} was 25 seconds. For the cold-VM reboot, the average downtime $D_c(11)$ was 219 seconds. After all the operating systems were booted again, the per-host throughput was largely degraded for a long time because of cache misses in all the operating systems. The rate of the performance degradation ω' was 0.72 and its duration R'_{vmm} was 397 seconds.

For the OS rejuvenation, the downtime D_{os} was 41 seconds. The throughput of an operating system was largely degraded while the other operating systems were rebooted. The sum of the duration is $(n-1)D_{os}$. On the other hand, the throughput was not degraded after the reboot of an operating system because the request rate was not too large to degrade the throughput. From these results, the rate of the performance degradation δ was 0.35 and its duration R_{os} was 410 seconds.

6.2 VM Migration

Instead of suspending or shutting down VMs when the VMM rejuvenation is performed, we can migrate them to another host. Using this technique, we can move all the VMs on a source host to a destination host before the VMM rejuvenation is started and then reboot the VMM on the source host. The migrated VMs can continue to run on the destination host while the VMM is rebooted on the source host. The migration mechanism suspends a VM, transfers its memory image and execution state through the network, and resumes it on the destination host. The storage used by the VM is shared between these two hosts. These two hosts are located in the same network segment to deliver network packets after migration. Although one extra host is needed for a destination host, the host can be shared among the remaining hosts in a cluster.

In particular, live migration [24] in Xen and VMotion in VMware [11] achieve negligible downtime of services provided by a migrated VM. Non-live migration stops a VM when starting migration whereas live migration transfers the memory image of a VM without stopping the VM. After the VMM transfers the whole memory image, it repeats transferring the changes in the memory image from the previous transmission until the changes become small. Finally, the VMM stops the VM and transfers the remaining changes and the execution state of the VM. To minimize the impact on services using the network in the other VMs, live migration can also control the transmission rate adaptively.

However, when we use migration, the total throughput is $(m-1)p$ even when no hosts are being migrated because one host is reserved as a destination host for migration. This is $(m-1)/m$ of the total throughput in a cluster environment where migration is not used. This is critical if m is not large enough.

6.2.1 Non-live Migration

To define the total throughput in the system using non-live migration, we first define the availability. The availability is defined by

$$A_m = 1 - \frac{D_m + N_m D_{os}}{T_{vmm}}$$

where D_m is the downtime due to non-live migration. N_m is the average number of the OS rejuvenation performed in T_{vmm} and defined by $(T_{vmm} - D_m)/T_{os}$.

The performance of a VM is degraded by the migration of the other $n-1$ VMs because transferring the large memory image consumes CPU time and the network bandwidth and disturbs services in the other VMs. We let γ denote the rate of the performance degradation ($0 \le \gamma \le 1$). Its duration is $(n-1)D_m$ because the time needed to migrate one VM is D_m. Therefore, the average performance degradation due to non-live migration is $\gamma(n-1)D_m/T_{vmm}$. We assume that VMs are migrated one by one because migrating them in parallel increases the downtime of each VM. On the other hand, the average performance degradation due to the OS rejuvenation is $N_m \delta R_{os}/T_{vmm}$, which is similar to that in the warm-VM reboot.

Therefore, the total throughput is defined by

$$P_m = \left(A_m - \frac{\gamma(n-1)D_m + N_m \delta R_{os}}{T_{vmm}} \right) (m-1)p. \quad (5)$$

Note that we can simultaneously use only $m-1$ hosts for providing services when we use migration.

6.2.2 Live Migration

In the system using live migration, the availability is defined by

$$A_l = 1 - \frac{N_l D_{os}}{T_{vmm}}$$

where N_l is the average number of the OS rejuvenation and defined by T_{vmm}/T_{os}. When using live migration, the downtime is caused only by the OS rejuvenation.

The average performance degradation due to live migration is $\gamma' M_l/T_{vmm}$. γ' is the rate of the performance degradation ($0 \le \gamma' \le 1$) and M_l is the whole migration time. The duration of the performance degradation is equal to M_l. On the other hand, the average performance degradation due to the OS rejuvenation is $N_l \delta R_{os}/T_{vmm}$.

Therefore, the total throughput is defined by

$$P_l = \left(A_l - \frac{\gamma' M_l + N_l \delta R_{os}}{T_{vmm}} \right) (m-1)p. \quad (6)$$

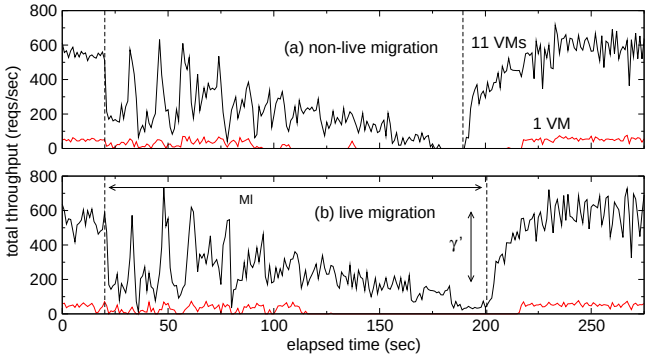

Fig. 14. The changes of the total throughputs during non-live and live migration.

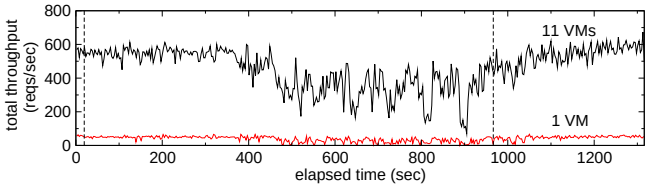

Fig. 15. The change of the total throughput during live migration with adaptive rate control.

6.2.3 Experimental Results

To estimate P_m and P_l, we examined the sum of the per-host throughputs of both the source and destination hosts while we performed Xen's migration. This means the total throughput when $m = 2$. For the source host, we used the same PC as used in Section 6.1. For the destination host, we used a PC with two Dual-Core Xeon 3.0 GHz processors, 12 GB of PC2-5300 DDR2 SDRAM memory, a 147 GB of 15,000 rpm SAS disk, and Gigabit Ethernet NICs. Since we could not install Xen 3.0.0, which we have modified to develop RootHammer, into this machine, we used Xen 3.1.0. To successfully migrate VMs, we also used the same version of Xen on the source host. We used the NFS server as a storage shared between these two hosts. For the NFS server machine, we used a PC with a Core 2 Duo E6700 processor, 2 GB of PC2-6400 DDR2 SDRAM memory, a 250 GB of SATA disk, and a Gigabit Ethernet NIC. These three PCs were connected with a Gigabit Ethernet switch.

Fig. 14 shows the changes of the total throughputs for non-live migration and live migration. Fig. 15 shows that for live migration with adaptive rate control. In Xen's adaptive rate control, the transmission rate starts from 100 Mbps and increases up to 500 Mbps according to the rate at which the memory contents of the VM being migrated are changed.

For non-live migration, the total throughput was largely degraded while each VM was being migrated. It was restored at the end of the migration of each VM because the system finished to transfer the large memory image and resumed the migrated VM. However, the throughput of the migrated VM became zero in most of the period. One of the possible reasons is that the destination host uses most of CPU time for handling the successive migration. In fact, the CPU utilization of the destination host was very high. Consequently, as the number of VMs running on the source host was decreasing by migration, the total throughput was also decreasing. The average downtime D_m of each VM was 15 seconds.

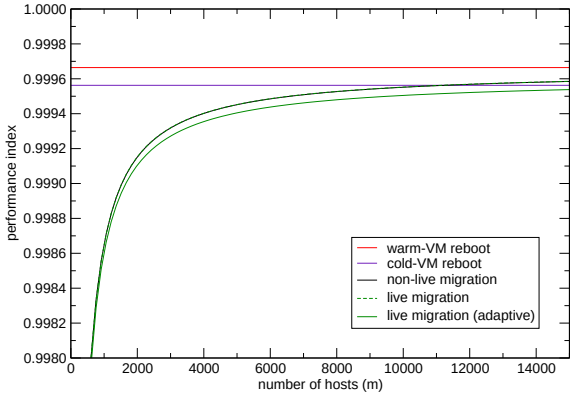

Fig. 16. The comparison of the total throughputs of five techniques.

The change of the total throughput in live migration without adaptive rate control is very similar to that in non-live migration. The difference is that a VM is running during its migration. Due to this overhead, it took slightly longer time than non-live migration. The whole migration time M_l was 181 seconds. On the other hand, when adaptive rate control was used in live migration, the change of the total throughput was largely different. The whole migration time was much longer. There was no performance degradation due to migration during 350 seconds after starting migration. After that, the performance was largely degraded. Due to the rate control, the whole migration time M_l increased to 950 seconds.

The rate of the performance degradation due to non-live migration γ was 0.60. For live migration, γ' was 0.57 when we did not use adaptive rate control. The value was 0.23 for live migration with adaptive rate control.

6.3 Comparison

We compare the total throughputs of the systems using various techniques described above, in terms of the number of hosts m in a cluster. As a performance index, we divide the total throughput estimated from our measurement by the maximum total throughput of m hosts. The maximum total throughput is the total throughput of a cluster under the assumption that there is no rejuvenation and all the hosts in a cluster are simultaneously used for providing services. In our case, that is mp. Fig. 16 plots the performance indexes for P_w, P_c, P_m, and P_l (non-adaptive and adaptive), which were obtained in the previous sections. We assumed that T_{vmm} was 28 days and T_{os} was 7 days.

The performance index in the system using the warm-VM reboot is always the highest and independent of the number of hosts. On the other hand, the indexes in the systems using migration depend on the number of hosts. When the number of hosts is larger than about 11000, the systems using non-live migration and live migration without adaptive rate control excel the system using the cold-VM reboot. In usual number of hosts ($m < 1000$), however, the total throughput of the system using the warm-VM reboot or the cold-VM reboot is much higher than that using VM migration.

According to this analysis, the warm-VM reboot is also useful in a cluster environment. It can keep high total throughput by reducing the downtime of rejuvenated hosts. On the other hand, for services that cannot be replicated to multiple hosts, live migration is still useful. It can prevent downtime by using an alternative host as a spare.

7 Related Work

Microreboot [25] enables rebooting fine-grained application components to recover from software failure. If rebooting a fine-grained component cannot solve problems, microreboot recursively attempts to reboot a coarser-grained component including that fine-grained component. If rebooting a finer-grained component can solve problems, the downtime of the application including that component can be reduced. Microreboot is a reactive technique, but proactively using it allows micro-rejuvenation. Likewise, microkernel operating systems [26] allow rebooting only its subsystems implemented as user processes. Nooks [27] enables restarting only device drivers in the operating system. Thus, microreboot and other previous proposals are fast reboot techniques for subcomponents. On the other hand, the warm-VM reboot is a fast reboot technique for a parent component while the state of subcomponents is preserved during the reboot.

In this paper, we have developed mechanisms to rejuvenate only a parent component when the parent component is a VMM and the subcomponents are VMs. Checkpointing and restart [28] of processes can be used to rejuvenate only an operating system. In this case, the parent component is an operating system and the subcomponents are its processes. This mechanism saves the state of processes to a disk before the reboot of the operating system and restores the state from the disk after the reboot. This is similar to suspend and resume of VMs, but suspending and resuming VMs are more challenging because they have to deal with a large amount of memory. As we showed in our experiments, simply saving and restoring the memory images of VMs to and from a disk are not realistic. The warm-VM reboot is a novel technique that hardly depends on the memory size by preserving the memory images.

To speed up suspend and resume using slow disks, several techniques are used. On suspend, VMware [11] incrementally saves only the modification of the memory image of a VM to a disk. This can reduce accesses to a slow disk although disk accesses on resume are not reduced. Windows XP saves compressed memory image to a disk on hibernation. This can reduce disk accesses not only on hibernation but also on resume. These techniques are similar to incremental checkpointing [29] and fast compression of checkpoints [30]. On the other hand, the warm-VM reboot does not need any disk accesses.

Instead of using slow hard disks for suspend and resume, it is possible to use faster disks such as solid state drives (SSDs) and non-volatile RAM disks [31]. Since most of the time for suspend and resume is spent to access slow disks, faster disks can speed up suspend and resume. However, such disks are much more expensive than hard disks. Moreover, it takes time to copy the memory images from main memory to disks on suspend and copy them from disks to main memory on resume. The warm-VM reboot needs neither such a special device nor extra memory copy.

Recovery Box [32] preserves only the state of an operating system and applications on non-volatile memory and restores them quickly after the operating system is rebooted. Recovery Box restores the partial state of a machine lost by a reboot while the warm-VM reboot restores the whole state of VMs lost by a reboot. In addition, Recovery Box speeds up a reboot by reusing the kernel text segment left on memory. This is different from our quick reload mechanism in that Recovery Box needs hardware support to preserve memory contents during a reboot.

For the file cache in operating systems, the Rio file cache [33] can preserve dirty file caches through the reboot of operating systems. Rio saves dirty file caches to disks when an operating system is rebooted and prevents any modification to files from being lost. Rio also improves the performance of file systems because it does not need to write back dirty file caches to disks periodically to prepare crashes. However, the performance is degraded after operating systems are rebooted because the file cache on memory is lost.

To mitigate software aging of domain 0, Xen provides driver domains, which are privileged VMs for running only device drivers for physical devices. Device drivers are one of the most error-prone components [34], [35]. In a normal configuration of Xen, device drivers are run in domain 0 and the rejuvenation of device drivers needs to reboot domain 0 and the VMM. Driver domains enable localizing the errors of device drivers inside them and rebooting them without rebooting the VMM. Thus, using driver domains reduces the frequency of the rejuvenation of the VMM. However, when the VMM is rebooted, driver domains as well as domain 0 are rebooted because driver domains deal with physical devices directly and cannot be suspended. Therefore, the existence of driver domains increases the number of domains to be rebooted and then increases the downtime due to the VMM rejuvenation.

8 Conclusion

In this paper, we proposed a new technique for fast rejuvenation of VMMs called the warm-VM reboot. This technique enables only a VMM to be rebooted by using the on-memory suspend/resume mechanism and the quick reload mechanism. The on-memory suspend/resume mechanism performs suspend and resume of VMs without saving the memory images to any persistent storage. The quick reload mechanism preserves the memory images during the reboot of a VMM. The warm-VM reboot can reduce the downtime and prevent the performance degradation just after the reboot. We have implemented this technique based on Xen and performed several experiments to show the effectiveness. The warm-VM reboot reduced the downtime by 74% at maximum and kept the same throughput after the reboot. Also, we showed that the warm-VM reboot also achieved the highest total throughput in a cluster environment.

Acknowledgments

This research was supported in part by JST, CREST.

References

[1] Y. Huang, C. Kintala, N. Kolettis, and N. Fulton, "Software Rejuvenation: Analysis, Module and Applications," in *Proc. Int'l Symp. Fault-Tolerant Computing*, 1995, pp. 381–391.
[2] S. Garg, A. van Moorsel, K. Vaidyanathan, and K. Trivedi, "A Methodology for Detection and Estimation of Software Aging," in *Proc. Int'l Symp. Software Reliability Engineering*, 1998, pp. 283–292.
[3] L. Li, K. Vaidyanathan, and K. Trivedi, "An Approach for Estimation of Software Aging in a Web Server," in *Proc. Intl. Symp. Empirical Software Engineering*, 2002, pp. 91–100.
[4] M. Grottke, L. Li, K. Vaidyanathan, and K. Trivedi, "Analysis of Software Aging in a Web Server," *IEEE Trans. Reliability*, vol. 55, no. 3, pp. 411–420, 2006.
[5] S. Garg, A. Puliafito, M. Telek, and K. Trivedi, "Analysis of Preventive Maintenance in Transactions Based Software Systems," *IEEE Trans. Computers*, 1998.
[6] 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," in *Proc. Symp. Operating Systems Principles*, 2003, pp. 164–177.
[7] S. Garg, Y. Huang, C. Kintala, and K. Trivedi, "Time and Load Based Software Rejuvenation: Policy, Evaluation and Optimality," in *Proc. Fault Tolerance Symposium*, 1995, pp. 22–25.
[8] K. Kourai and S. Chiba, "A Fast Rejuvenation Technique for Server Consolidation with Virtual Machines," in *Proc. Int'l Conf. Dependable Systems and Networks*, 2007, pp. 245–254.
[9] M. Kanno, "Xen changeset 9392," Xen Mercurial repositories.
[10] K. Fraser, "Xen changeset 11752," Xen Mercurial repositories.
[11] VMware Inc., "VMware," http://www.vmware.com/.
[12] K. Vaidyanathan and K. Trivedi, "A Measurement-Based Model for Estimation of Software Aging in Operational Software Systems," in *Proc. Int'l. Symp. Software Reliability Engineering*, 1999, pp. 84–93.
[13] ——, "A Comprehensive Model for Software Rejuvenation," *IEEE Trans. Dependable and Secure Computing*, vol. 2, no. 2, pp. 124–137, 2005.
[14] V. Hanquez, "Xen changeset 8640," Xen Mercurial repositories.
[15] Intel Corporation, *Intel Virtualization Technology Specification for the IA-32 Intel Architecture*, 2005.
[16] AMD, *AMD64 Virtualization Codenamed "Pacifica" Technology: Secure Virtual Machine Architecture Reference Manual*, 2005.
[17] JBoss Group, "JBoss Application Server," http://www.jboss.com/.
[18] Hewlett-Packard, Intel, Microsoft, Phoenix Technologies, and Toshiba, "Advanced Configuration and Power Interface Specification, Revision 3.0b," http://www.acpi.info/, 2006.
[19] A. Pfiffer, "Reducing System Reboot Time with kexec," http://www.osdl.org/.
[20] Apache Software Foundation, "Apache HTTP Server Project," http://httpd.apache.org/.
[21] D. Mosberger and T. Jin, "httperf: A Tool for Measuring Web Server Performance," *Performance Evaluation Review*, vol. 26, no. 3, pp. 31–37, 1998.
[22] V. Castelli, R. Harper, P. Heidelberger, S. Hunter, K. Trivedi, K. Vaidyanathan, and W. Zeggert, "Proactive Management of Software Aging," *IBM J. Research & Development*, vol. 45, no. 2, pp. 311–332, 2001.
[23] K. Vaidyanathan, R. Harper, S. Hunter, and K. Trivedi, "Analysis and Implementation of Software Rejuvenation in Cluster Systems," in *Proc. ACM SIGMETRICS Int'l Conf. Measurement and Modeling of Computer Systems*, 2001, pp. 62–71.
[24] C. Clark, K. Fraser, S. Hand, J. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live Migration of Virtual Machines," in *Proc. Symp. Networked Systems Design and Implementation*, 2005, pp. 1–11.
[25] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox, "Microreboot – A Technique for Cheap Recovery," in *Proc. Symp. Operating Systems Design and Implementation*, 2004, pp. 31–44.
[26] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A New Kernel Foundation for UNIX Development," in *Proc. USENIX Summer Conference*, 1986, pp. 93–112.
[27] M. Swift, B. Bershad, and H. Levy, "Improving the Reliability of Commodity Operating Systems," in *Proc. Symp. Operating Systems Principles*, 2003, pp. 207–222.
[28] B. Randell, "System Structure for Software Fault Tolerance," *IEEE Trans. on Softw. Eng.*, vol. SE-1, no. 2, pp. 220–232, 1975.
[29] S. Feldman and C. Brown, "IGOR: A System for Program Debugging via Reversible Execution," in *Proc. Workshop on Parallel and Distributed Debugging*, 1989, pp. 112–123.
[30] J. Plank, J. Xu, and R. Netzer, "Compressed Differences: An Algorithm for Fast Incremental Checkpointing," University of Tennessee, Tech. Rep. CS–95–302, 1995.
[31] GIGABYTE Technology, "i-RAM," http://www.gigabyte.com.tw/.
[32] M. Baker and M. Sullivan, "The Recovery Box: Using Fast Recovery to Provide High Availability in the UNIX Environment," in *Proc. USENIX Summer Conf.*, 1992, pp. 31–44.
[33] P. Chen, W. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell, "The Rio File Cache: Surviving Operating System Crashes," in *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, 1996, pp. 74–83.
[34] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An Empirical Study of Operating Systems Errors," in *Proc. Symp. Operating Systems Principles*, 2001, pp. 73–88.
[35] A. Ganapathi, V. Ganapathi, and D. Patterson, "Windows XP Kernel Crash Analysis," in *Proc. Large Installation System Administration Conf.*, 2006, pp. 149–159.

Kenichi Kourai received his PhD degree from the University of Tokyo in 2002. Since 2008, he is an associate professor in Department of Creative Informatics at Kyushu Institute of Technology, Japan. He has been working on operating systems. His current research interest is in dependable and secure systems using virtual machines. He is a member of the IEEE Computer Society and the ACM.

Shigeru Chiba received his PhD degree from the University of Tokyo in 1996. Since 2008, he is a professor in Department of Mathematical and Computing Sciences at Tokyo Institute of Technology, Japan. He has been working on programming language design and middleware/operating systems. His current research interest is in software composition from various aspects. Dr. Chiba has been also active as a program committee member of a number of conferences and journals, including ACM OOPSLA and ACM/IEEE MODELS.