A Fast Rejuvenation Technique for Server Consolidation with Virtual Machines

Kenichi Kourai
Tokyo Institute of Technology
2-12-1 Ookayama, Meguro-ku, Tokyo
152-8552, Japan
kourai@is.titech.ac.jp

Shigeru Chiba
Tokyo Institute of Technology
2-12-1 Ookayama, Meguro-ku, Tokyo
152-8552, Japan
chiba@is.titech.ac.jp

Abstract

As server consolidation using virtual machines (VMs) is carried out, software aging *of virtual machine monitors (VMMs) is becoming critical. Performance degradation or crash failure of a VMM affects all VMs on it. To counteract such software aging, a proactive technique called* software rejuvenation *has been proposed. A typical 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 accessing the memory images. To achieve this, we have developed two mechanisms:* on-memory suspend/resume *of VMs and* quick reload *of VMMs. The warm-VM reboot reduces the downtime and prevents the performance degradation due to cache misses after the reboot.*

1. Introduction

The phenomenon that the state of software degrades with time is known as *software aging* [16]. The causes of this degradation are the exhaustion of system resources and data corruption. This often leads to performance degradation of the software or crash failure. Recently, software aging of virtual machine monitors (VMMs) is becoming critical as server consolidation using virtual machines (VMs) is being widely carried out. Many VMs run on top of a VMM in one machine consolidating multiple servers and aging of the VMM directly affects all the VMs.

To counteract such software aging, a proactive technique called *software rejuvenation* has been proposed [16]. Software rejuvenation occasionally stops a running VMM, cleans its internal state, and restarts it. A typical example of rejuvenation is to reboot a VMM. 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 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 is 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.

In this paper, we propose 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 VMMs. Using the on-memory suspend/resume mechanism, a 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 using the quick reload mechanism without a hardware reset. 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 operating systems.

To achieve this fast rejuvenation, we have developed *RootHammer* based on Xen [9]. From our experimental results, the warm-VM reboot reduced the downtime due to rebooting the VMM by 83 % at maximum. For comparison, when we simply used the suspend/resume mechanism of the original Xen, the downtime was increased by 173 %. After the warm-VM reboot, the throughput of a web server was not degraded at all. When we did not use the warm-VM reboot, the throughput was degraded by 69 % just after the reboot of the VMM.

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 estimates the downtime reduced by it. Section 4 explains our implementation based on Xen and Section 5 shows our experimental results. Section 6 discusses the advantage 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, it is affected by software aging more largely than the other components. For example, a VMM may leak its memory by failing to release a part of memory. In Xen [9], 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 [19] or when some error paths were executed [11]. Out-of-memory errors can lead performance degradation or crash failure of the VMM. Such problems of the VMM directly affect all the VMs.

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 [26] to help the VMM for VM management and/or I/O processing of all VMs. 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 [13]. In privileged VMs, memory exhaustion easily occurs because the typical size of the memory allocated to them is not so large. Since privileged VMs do not run large servers, they do not need a large amount of memory. For example, Xen had a bug of memory leaks in its daemon named xenstored running on a privileged VM [15]. 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.

To counteract such software aging, a proactive technique called *software rejuvenation* has been proposed [16].

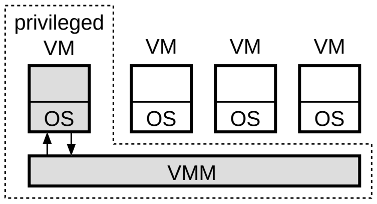

Figure 1. An assumed VM architecture.

Software rejuvenation occasionally stops a running VMM, cleans its internal state, and restarts it. A typical example of rejuvenation is to reboot a VMM. 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.

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. Unfortunately, the number of VMs that can run simultaneously is increasing due to processor support of virtualization such as Intel VT [17] and AMD Virtualization [3] and multi-core processors. In addition, recent servers tend to provide heavy-weight services such as the JBoss application server [18] 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.

In addition, 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 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 simply wasteful.

3.1. Warm-VM Reboot

To minimize the influences of the rejuvenation of VMMs, we propose a new technique for fast rejuvenation called the *warm-VM reboot*. The basic idea is that a VMM preserves the memory images of all the 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 for VMs and the *quick reload* mechanism for VMMs. A 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.

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 using quick reload. This 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 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 ACPI S3 state (Suspend To RAM) [2] 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 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 when the VMM initializes itself.

Although many VMMs provide suspend/resume mechanisms, they are not suitable to use for rejuvenation of VMMs because they have to use disks as persistent storage to save memory images. These traditional suspend/resume mechanisms are analogous to 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 disks before the reboot of a VMM. Our quick reload mechanism allows the VMM to reuse the memory images on volatile main memory by preserving them during the reboot.

3.2. Downtime Estimation

To estimate the downtime reduced by using the warm-VM reboot, let us consider the usage model of software rejuvenation. Usually the rejuvenation of a VMM (VMM rejuvenation) is used with the rejuvenation of operating systems (OS rejuvenation). In general, the OS rejuvenation is performed more frequently than the VMM rejuvenation. 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* [12]. When the warm-VM reboot is used, the VMM rejuvenation can be performed independently of the OS rejuvenation as shown in Figure 2 (a). This is because the warm-VM reboot does not involve the OS rejuvenation. On the other hand, when a VMM is rejuvenated by a normal reboot, which we call the *cold-VM reboot* in contrast to the warm-VM reboot,

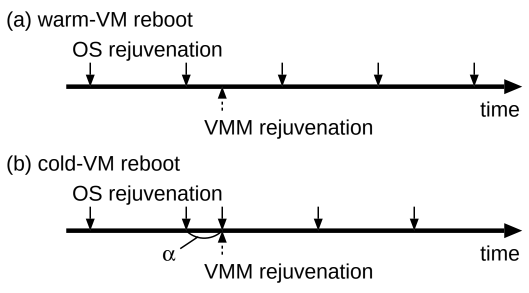

Figure 2. The timing of two kinds of rejuvenation. The rejuvenation of all but one operating system is omitted.

the VMM rejuvenation affects the timing of the OS rejuvenation as shown in Figure 2 (b) because the VMM rejuvenation involves the OS rejuvenation. The OS rejuvenation after the VMM rejuvenation will be performed at fixed intervals again.

When the warm-VM reboot is used, the downtime due to the VMM rejuvenation is caused by suspending all VMs, rebooting the VMM, and resuming all VMs. The increase of the downtime is:

$$d_w(n) = reboot_{vmm}(n) + resume(n)$$

where n is the number of VMs, $reboot_{vmm}(n)$ is the time needed to reboot a VMM when n VMs are suspended and resumed, and $resume(n)$ is the time needed to perform on-memory suspend and resume of n VMs in parallel.

On the other hand, when the cold-VM reboot is used, the downtime due to the VMM rejuvenation is caused by shutting down all operating systems, resetting hardware, rebooting a VMM, and booting all operating systems. The increase of the downtime is:

$$d_c(n) = reset_{hw} + reboot_{vmm}(0) + reboot_{os}(n) - reboot_{os}(1) \times \alpha$$

where $reset_{hw}$ is the time needed for a hardware reset, $reboot_{os}(n)$ is the time needed to shut down and boot n operating systems in parallel, and α is a ratio of the time elapsed until the VMM rejuvenation since the last OS rejuvenation to an interval between the OS rejuvenation ($0 < \alpha \le 1$). Since the OS rejuvenation is rescheduled after the VMM rejuvenation, the number of the OS rejuvenation is decreased by α in total although extra OS rejuvenation is added by the VMM rejuvenation.

The downtime reduced by using the warm-VM reboot is calculated by $d_c(n) - d_w(n)$:

$$r(n) = reset_{hw} + reboot_{vmm}(0) - reboot_{vmm}(n) + reboot_{os}(n) - reboot_{os}(1) \times \alpha - resume(n)$$

4. Implementation

To achieve the warm-VM reboot, we have developed *RootHammer* based on Xen 3.0.0. Like 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 P2M-mapping table is a table that records mapping from PFN to MFN for each domain. The size of our P2M-mapping table is 2 MB for 1 GB of pseudo-physical memory. A new entry is added to this table when a new machine page frame is allocated to a domain while an existing entry is removed when a machine page frame is deallocated. These entries are preserved after domains are suspended. Even when the total size of pseudo-physical memory is larger than that of machine memory due to using a ballooning technique [27], this table can maintain the mapping properly.

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 Figure 3. To suspend domain Us, the VMM sends a suspend event to each domain U. In the original Xen, domain 0 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 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. In the handler, the kernel detaches all devices. We used the handler implemented in the Linux kernel modified for Xen.

After the operating system in a domain U executes the suspend handler, it issues the suspend hypercall to the VMM, which is like a system call to the operating system. In the hypercall, the VMM freezes the memory image of

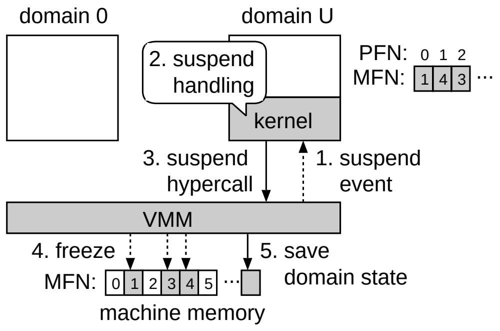

Figure 3. On-memory suspend of a domain U.

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 devices that were detached on suspend. Finally, the execution of the kernel 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 [21] 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 hypercall in the VMM.

When the xexec system call is issued in domain 0, the kernel issues the xexec hypercall to the VMM. This hypercall 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 hypercall 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. The latest Xen 3.0.4 also supports the kexec facility for its VMM, but it does not have any support to preserve 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, 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 the 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.

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 a ssh server in each VM as a service provided to the outside. 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. Figure 4 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

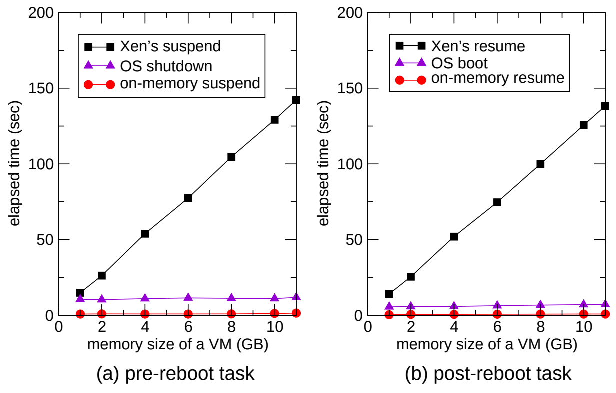

Figure 4. The time for pre- and post-reboot tasks when the memory size of a VM is changed.

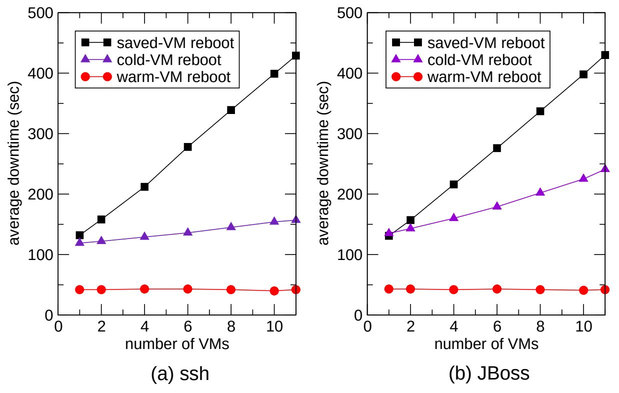

Figure 6. The downtime of ssh and JBoss when the number of VMs is changed.

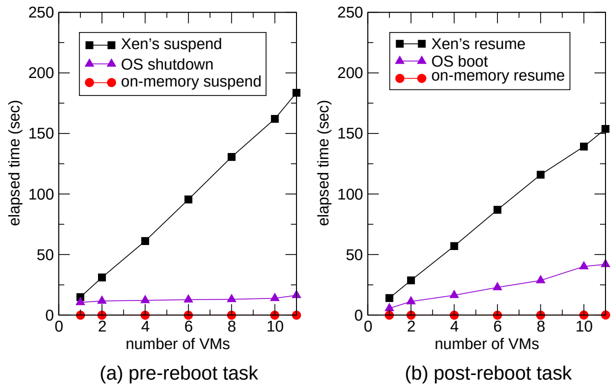

Figure 5. The time for pre- and post-reboot tasks when the number of VMs is changed.

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 0.9 second for resume. These are only 0.06 % and 0.7 % 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. Figure 5 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 4.2 seconds for resume. These were 0.02 % and 2.7 % 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 repeated sending packets from a client host to the VMs in a server host. We measured the time from when a networked service in each VM was down and until it was up again after the VMM was rebooted. We performed this experiment for (1) the warm-VM reboot, (2) the reboot using Xen's suspend/resume (*saved-VM reboot*), and (3) the reboot by shutdown/boot (*cold-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 a ssh server in each VM and measured its downtime during the reboot of the VMM. Figure 6 (a) shows the downtime. The downtime by the saved-VM reboot highly depended on the number of VMs. When the number was 11, the downtime was 429 seconds in average. At the same number of VMs, the downtime by the warm-VM reboot was 42 seconds and only 9.8 % of the saved-VM reboot. In addition, the downtime by the warm-VM reboot hardly depended on the number of VMs. On the other hand, the downtime by the cold-VM reboot was 157 seconds when the number of VMs was 11. This was 3.7 times longer than 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 TCP retransmission, 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 a JBoss application server [18] and measured its downtime during the reboot of a VMM. JBoss is a large server and it takes more time to start than a ssh server. We used the default configuration of JBoss. Figure 6 (b) shows the downtime. The downtime by 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 by 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 was 11, the downtime was 241 seconds. 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.

Let us consider the availability of the JBoss server when the number of VMs is 11. As an example, we assume that the OS rejuvenation is performed every week and the VMM rejuvenation is performed once per four weeks. According to our experiment, the downtime due to the OS rejuvenation was 33.6 seconds. For the cold-VM reboot, we assume that the expected value of α in Section 3.2 is 0.5. Under these assumptions, the availability is 99.993 %, 99.985 %, and 99.977 % for the warm-VM reboot, the cold-VM reboot, and the saved-VM reboot, respectively. The warm-VM reboot achieves four 9s although the others achieve three 9s. This improvement of availability is important for critical servers.

5.4. Downtime Analysis

To examine which factors reduce 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 [4] running on a VM in a server host by using the httperf benchmark tool [20]. We created 11 VMs and allocated 1 GB of memory to each VM. We rebooted the VMM and recorded the changes of the average throughput of 50 requests. We performed this experiment for the warm-VM reboot and the cold-VM reboot. Figure 7 shows the results. We executed the reboot command in domain 0 at time 20 seconds in this figure. We superimposed the time needed for each operation during the reboot onto Figure 7.

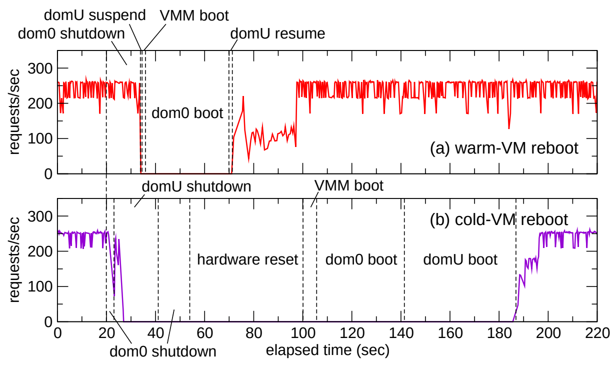

Figure 7. The breakdown of the downtime due to the VMM rejuvenation.

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. The throughput in the cold-VM reboot was degraded during 8 seconds. This was due to misses of the file cache. We examine this performance degradation in detail in the next section. The throughput in the warm-VM reboot was also degraded during 25 seconds after the reboot. This is not due to cache misses but an implementation problem of Xen. When Xen created new VMs simultaneously, the network performance was degraded for a while.

5.5. 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 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. 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. Figure 8 (a) shows the result. When we used the

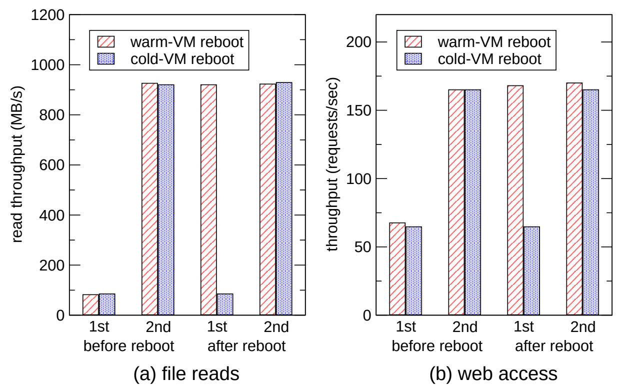

Figure 8. The throughput of file reads and web accesses before and after the reboot.

cold-VM reboot, the throughput just after the reboot was degraded by 91 %, compared with that just before the reboot. On the other hand, when we used the warm-VM reboot, the throughput just after the reboot was not degraded. This improvement was achieved by no miss in the file cache even when a file was accessed at the first time after the reboot.

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. Figure 8 (b) 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 69 %.

5.6. Applying to Our Model

From our experimental results when we ran 11 VMs, we can get the functions used in our model in Section 3.2:

$$reboot_{vmm}(n) = -0.55n + 43$$
$$resume(n) = 0.43n - 0.07$$
$$reboot_{os}(n) = 3.8n + 13$$
$$boot(n) = 3.4n + 2.8$$
$$reset_{hw} = 47$$

Using these functions, we can get the function of the downtime reduced by using the warm-VM reboot:

$$r(n) = 3.9n + 60 - 17\alpha$$

Since $r(n)$ is always positive under $\alpha \leq 1$, the warm-VM reboot can always reduce the downtime in our configuration.

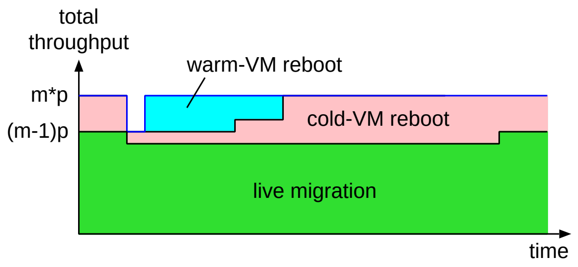

Figure 9. The total throughput in a cluster environment. m is the number of hosts and p is the throughput of each host.

6. Cluster Environment

Software rejuvenation is naturally fit with a cluster environment as described in the literature [7, 25]. 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 the service is degraded while some hosts are rebooted. The warm-VM reboot can mitigate the performance degradation by reducing the downtime of rebooted hosts.

Migration of VMs can be also used in a cluster environment to reduce the total cost. Unlike the warm-VM reboot, live migration [8] in Xen and VMotion in VMware [26] achieve negligible service downtime by using two hosts when a VMM is rejuvenated. Before the VMM is rebooted, it transfers the memory images of all VMs running on it to a destination host without stopping the VMs. After that, the VMM repeats transferring the changes in the memory images from the previous transmission until the changes become small. Finally, the VMM stops the VMs and transfers the changes and the execution state of the VMs. If we use live migration in a cluster environment, that destination host for migration can be shared among the remaining hosts.

Let us consider a cluster environment that consists of m hosts to estimate the total throughput of the cluster. When we let p be the throughput of each host, the total throughput is $m \cdot p$ when all hosts are running. Figure 9 illustrates the changes of the total throughput with time, based on our experimental results. During the rejuvenation of a VMM in one host, the total throughput is decreased to $(m-1)p$ because the rejuvenated host cannot provide any services. When we use the warm-VM reboot, the degradation of the total throughput lasts only for a short period. The period is the same as the downtime in the rejuvenated host and it was 42 seconds in our experimental environment. The total throughput is restored to $m \cdot p$ 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 a longer period. In our experimental environment, the period was 241 seconds when we created 11 VMs and ran JBoss. In addition, the total throughput is degraded to $(m - \delta)p$ $(0 \le \delta \le 1)$ for a while after the rejuvenation due to cache misses. In our experiment of Section 5.5, δ was 0.69.

On the other hand, when we use live 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 $\frac{m-1}{m}$ of the total throughput in a cluster environment where migration is not used. This is critical if m is not large enough. While one host performs live migration, the total throughput is $(m - 1.12)p$, which is led from the report that the degradation of the Apache web server was 12 % during live migration [8]. This degradation of the total throughput is estimated to last for 17 minutes when we run 11 VMs, each of which has 1 GB of memory. This is calculated from the report that the time needed for migration was 72 seconds when only one VM with 800 MB of memory was run [8]. This period of performance degradation is much longer than those in the warm-VM reboot and the cold-VM reboot. Although these reported values are not measured in our experimental environment, the trend would not be changed.

According to these analyses, the warm-VM reboot is more useful in a cluster environment than live migration. It can reduce performance degradation 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 reduce downtime by using alternative host as a spare.

7. Related Work

Microreboot [6] 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 [1] allow rebooting only its subsystems implemented as user processes. Nooks [24] 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 [23] 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 [26] 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 (Suspend To Disk). This can reduce disk accesses not only on hibernation but also on resume. These techniques are similar to incremental checkpointing [10] and fast compression of checkpoints [22]. 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 non-volatile RAM disks such as i-RAM [14]. Since most of the time for suspend and resume is spent to access slow disks, RAM disks can speed up the access. However, such non-volatile RAM disks are much more expensive than hard disks. Moreover, it takes time to copy the memory images from main memory to RAM disks on suspend and copy them from RAM disks to main memory on resume. The warm-VM reboot needs neither such a special device nor extra memory copy.

Recovery Box [5] 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.

To mitigate software aging of domain 0, Xen provides driver domains, which are domain Us that enable running device drivers. Device drivers are one of the most error-prone components. 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 in domain Us and rebooting the domains 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 cannot be suspended. Therefore, the existence of driver domains increases the downtime.

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 accessing the memory images. 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 83 % at maximum and kept the same throughput after the reboot.

One of our future directions is to empirically evaluate the reduction of performance degradation by using the warm-VM reboot in a cluster environment. Another direction is to enable privileged VMs to be rebooted without the reboot of the VMM and to be suspended.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of the USENIX 1986 Summer Conference*, pages 93–112, 1986.

[2] Advanced Configuration and Power Interface Specification. `http://www.acpi.info/`.

[3] AMD. *AMD64 Virtualization Codenamed "Pacifica" Technology: Secure Virtual Machine Architecture Reference Manual*, 2005.

[4] Apache Software Foundation. Apache HTTP Server Project. `http://httpd.apache.org/`.

[5] M. Baker and M. Sullivan. The Recovery Box: Using Fast Recovery to Provide High Availability in the UNIX Environment. In *Proceedings of the Summer USENIX Conference*, pages 31–44, 1992.

[6] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot – A Technique for Cheap Recovery. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 31–44, 2004.

[7] V. Castelli, R. Harper, P. Heidelberger, S. Hunter, K. Trivedi, K. Vaidyanathan, and W. Zeggert. Proactive Management of Software Aging. *IBM Journal of Research & Development*, 45(2):311–332, 2001.

[8] C. Clark, K. Fraser, S. Hand, J. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation*, pages 1–11, 2005.

[9] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the Art of Virtualization. In *Proceedings of the Symposium on Operating Systems Principles*, pages 164–177, 2003.

[10] S. Feldman and C. Brown. IGOR: A System for Program Debugging via Reversible Execution. In *Proceedings of the Workshop on Parallel and Distributed Debugging*, pages 112–123, 1989.

[11] K. Fraser. Xen changeset 11752. Xen Mercurial repositories.

[12] S. Garg, Y. Huang, C. Kintala, and K. Trivedi. Time and Load Based Software Rejuvenation: Policy, Evaluation and Optimality. In *Proceedings of the 1st Fault Tolerance Symposium*, pages 22–25, 1995.

[13] S. Garg, A. Moorsel, K. Vaidyanathan, and K. Trivedi. A Methodology for Detection and Estimation of Software Aging. In *Proceedings of the 9th International Symposium on Software Reliability Engineering*, pages 283–292, 1998.

[14] GIGABYTE Technology. i-RAM. `http://www.gigabyte.com.tw/`.

[15] V. Hanquez. Xen changeset 8640. Xen Mercurial repositories.

[16] Y. Huang, C. Kintala, N. Kolettis, and N. Fulton. Software Rejuvenation: Analysis, Module and Applications. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing*, pages 381–391, 1995.

[17] Intel Corporation. *Intel Virtualization Technology Specification for the IA-32 Intel Architecture*, 2005.

[18] JBoss Group. JBoss Application Server. `http://www.jboss.com/`.

[19] M. Kanno. Xen changeset 9392. Xen Mercurial repositories.

[20] D. Mosberger and T. Jin. httperf: A Tool for Measuring Web Server Performance. *Performance Evaluation Review*, 26(3):31–37, 1998.

[21] A. Pfiffer. Reducing System Reboot Time with kexec. `http://www.osdl.org/`.

[22] J. Plank, J. Xu, and R. Netzer. Compressed Differences: An Algorithm for Fast Incremental Checkpointing. Technical Report CS–95–302, University of Tennessee, 1995.

[23] B. Randell. System Structure for Software Fault Tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220–232, 1975.

[24] M. Swift, B. Bershad, and H. Levy. Improving the Reliability of Commodity Operating Systems. In *Proceedings of the 19th Symposium on Operating Systems Principles*, pages 207–222, 2003.

[25] K. Vaidyanathan, R. Harper, S. Hunter, and K. Trivedi. Analysis and Implementation of Software Rejuvenation in Cluster Systems. In *Proceedings of the 2001 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 62–71, 2001.

[26] VMware Inc. VMware. `http://www.vmware.com/`.

[27] C. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 181–194, 2002.