Zero-copy Migration for Lightweight Software Rejuvenation of Virtualized Systems

Kenichi Kourai
Kyushu Institute of Technology
kourai@ci.kyutech.ac.jp

Hiroki Ooba
Kyushu Institute of Technology
hiroki@ksl.ci.kyutech.ac.jp

Abstract

Virtualized systems tend to suffer from software aging, which is the phenomenon that the state of a running system degrades with time. Software aging is restored by a technique called software rejuvenation, e.g., a system reboot. To reduce the downtime due to software rejuvenation, all the virtual machines (VMs) on an aged virtualized system have to be migrated in advance. However, VM migration stresses the system and causes performance degradation. In this paper, we propose *VMBeam*, which enables lightweight software rejuvenation of virtualized systems using *zero-copy migration*. When rejuvenating an aged virtualized system, VMBeam starts a new virtualized system at the same host by using *nested virtualization*. Then it migrates all the VMs from the aged virtualized system to the clean one. At this time, VMBeam directly relocates the memory of the VMs on the aged virtualized system to the clean virtualized system without any copy. We have implemented VMBeam in Xen and confirmed the decreases of system loads.

1. Introduction

A virtualized system enables running many virtual machines (VMs) on one physical host. This reduces the number of physical hosts in data centers and eliminates the cost. However, virtualized systems tend to suffer from the phenomenon called software aging [7] because they are running for a long time and accumulates more errors. Software aging is caused by memory leakage, for example, and degrades the state of running software with time. It results in the performance degradation of virtualized systems and VMs running on top of it and can lead to unplanned system stops in the worst case.

APSys 2015, July 27–28, 2015, Tokyo, Japan.

http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/2797022.2797026

To counteract such software aging, a technique called software rejuvenation has been proposed [7]. Software rejuvenation returns the state of virtualized systems to that before software aging progresses. It can recover the system performance and prevent unexpected system down. A typical example of software rejuvenation is a system reboot. However, rebooting a virtualized system needs to stop all the VMs on top of it and to restart them after the reboot. Since it takes a long time to shut down and boot the system in each VM, the downtime cannot be ignored.

One possible approach for reducing this downtime is to use VM migration. VM migration enables a VM to be moved to another physical host without stopping it. Using VM migration, software rejuvenation can first migrate all the VMs with negligible downtime and then reboot a virtualized system without VMs. As a result, software rejuvenation does almost not interrupt services running in the VMs. However, VM migration largely stresses the involving hosts and network because it has to transfer large memory images via the network [12]. This can lead to service level objective (SLO) violations.

In this paper, we propose VMBeam, which makes software rejuvenation of virtualized systems more lightweight using *zero-copy migration*. On rejuvenating a running virtualized system, VMBeam starts another clean virtualized system at the same host by using *nested virtualization*. Then it migrates all the VMs from the aged virtualized system to the new one. At this time, it leverages the fact that these two virtualized systems run at the same host. Instead of transferring the memory images of VMs via the network, VMBeam directly relocates the memory of VMs between different virtualized systems without any copy. To enable a VM to continue to run during migration, it makes a VM newly created on the clean virtualized system temporarily share the memory of the VM on the aged virtualized system. Therefore zero-copy migration does not need to re-transfer the memory modified during migration.

We have implemented VMBeam in Xen 4.2.2 [1]. In the current implementation, two virtualized systems using Xen run on the Xen hypervisor. On VM migration, a clean virtualized system creates a new empty VM and both virtual-

ized systems pass only the memory information on the two VMs to the host hypervisor. Using that information, the host hypervisor makes both VMs share the memory by *inter-guest memory sharing*. According to our experimental results, VMBeam could suppress the CPU, memory, and network loads to 31%, nearly 0%, and nearly 0% of the traditional migration, respectively. In addition, it could perform VM migration 5.8 times faster than the traditional migration between physical hosts at maximum.

This paper is organized as follows. Section 2 describes issues of software rejuvenation of virtualized systems. Section 3 proposes lightweight software rejuvenation using zero-copy migration. Section 4 explains the implementation details and Section 5 shows the experimental results. Section 6 explains the related work and Section 7 concludes this paper.

2. Software Rejuvenation of Virtualized Systems

In virtualized systems, software aging [7] tends to progress because virtualized systems are long-running software, which accumulates more errors over time. A virtualized system mainly consists of the hypervisor and often the management VM. Since a virtualized system has to usually host many VMs, it is not easy to find appropriate timing for the stop of a virtualized system. Recently, software aging in Xen has been studied [15]. The study reported that the free memory of the management VM decreased by 80% after VM migration was performed 100 times. Also, Xen had a bug that caused the decrease of the free disk space by 185 MB after a VM was suspended and resumed. As software aging is progressing, the system performance can degrade.

Software rejuvenation [7] is a proactive technique for counteracting such software aging of virtualized systems. It can restore virtualized systems to the normal state before the progress of software aging and prevent the performance degradation. Its simplest but powerful method is to reboot a virtualized system. On software rejuvenation, however, a virtualized system has to stop all the VMs on top of it and restart them again after its reboot. It takes a long time to shut down and boot the system running in each VM. In addition, booting the system often causes heavy disk accesses and degrades the system performance. Worse, recently, more VMs are being consolidated in one virtualized system. Therefore, the downtime of the services provided by VMs tends to be longer. The proposed techniques for fast software rejuvenation of virtualized systems [11, 13] can avoid rebooting the systems in VMs and rejuvenate only a virtualized system. However, the time needed for rejuvenating a virtualized system still becomes the downtime.

To reduce such downtime caused by software rejuvenation, VM migration is often used. In particular, live migration [4] can achieve negligible downtime. First, a source virtualized system transfers the memory image of a VM to a destination system with the VM running and stores it into a newly created VM. After transferring the entire memory image, it transfers only memory areas modified during migration again until the size of such areas are small enough. At the final stage, the source system stops the VM and transfers the remaining memory areas modified, while the destination system resumes the cloned VM. Using this technique, a virtualized system can migrate all the VMs to other hosts at the first stage of software rejuvenation. Then it can reboot itself without VMs. As a result, software rejuvenation does almost not affect the downtime of VMs.

However, VM migration stresses involving hosts and network largely. Since all the VMs have to be migrated, the size of transferred memory images can be usually from several gigabytes to several hundreds of gigabytes in total. Such memory images are often encrypted to prevent eavesdropping and tampering during the transfer via the network. Therefore, VM migration can occupy CPUs and memory bandwidths at both source and destination hosts. In addition, the network performance can be also degraded when the dedicated network for VM migration is not used. These high loads in hosts and network degrade the performance of virtualized systems and VMs running in both hosts. In fact, it is reported that the web servers in migrating VMs cause performance degradation by 57% on average during the migration of 11 VMs [12]. Consequently, VM migration can violate SLOs such as response time even if the frequency of software rejuvenation is not so high.

3. VMBeam

In this paper, we propose VMBeam for enabling lightweight software rejuvenation of virtualized systems. When rejuvenating a virtualized system, VMBeam first starts a new virtualized system at the same host by using *nested virtualization*. After that, it migrates all the VMs running on top of the aged virtualized system onto the new one, using *zero-copy migration*. Zero-copy migration performs memory relocation across virtualized systems by leveraging the fact that VM migration is performed inside one host and efficiently transfers the memory images of VMs. Finally, VMBeam stops the aged virtualized system to complete software rejuvenation without rebooting it.

3.1 Using Nested Virtualization

VMBeam uses nested virtualization [2] to run two virtualized systems at the same host. Nested virtualization enables a virtualized system to run in a VM. Fig. 1 shows the system architecture of VMBeam while it is rejuvenating a virtualized system. In this paper, we call the traditionally used hypervisor and VMs the *guest* hypervisor and VMs, respectively. In contrast, we call added extra ones the *host* hypervisor and VMs, respectively. During software rejuvenation, two host VMs run on top of the host hypervisor and the guest hypervisor and guest VMs run inside each host VM.

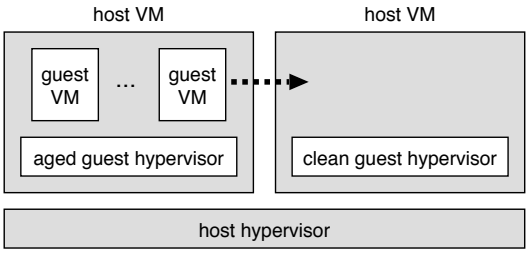

Figure 1. The system architecture of VMBeam.

When software rejuvenation is not in progress, devirtualization [3, 9, 10, 14, 17] can largely reduce the overhead of nested virtualization. This is a technique for temporarily disabling virtualization provided by the hypervisor. Using this technique, VMBeam can devirtualize the entire system at the end of software rejuvenation and revirtualize it at the beginning of the next rejuvenation. While the virtualization by the host hypervisor is disabled, we could gain performance similar to usual single-level virtualization. Although this technique has not been proposed for nested virtualization, it would be also applicable to the host hypervisor. One downside of using devirtualization is that the guest hypervisor could directly corrupt the hardware state, so that a hardware reset is required. Without devirtualization, the hardware state is protected by the host hypervisor. In this paper, we assume that software aging in the hypervisor does not corrupt the hardware state in an unrecoverable manner.

In addition, several mechanisms have been proposed to reduce the overhead of nested virtualization [2]. They can suppress the performance degradation to 6% to 8%. Special-purpose host hypervisors as used in CloudVisor [22] and TinyChecker [18] can improve the performance of nested virtualization more. Recently, hardware support for nested virtualization has been also added. For example, Intel VMCS Shadowing [8] can eliminate VM exits due to VMREAD and VMWRITE instructions for accessing VMCS. Note that reducing the overhead of nested virtualization is out of scope of this paper. This paper focuses on reducing the overhead of software rejuvenation.

VMBeam needs extra resources for the host hypervisor and, during software rejuvenation, for one more virtualized system. However, the total amount of resources consumed by guest VMs does not increase. Guest VMs run only in either host VM during a normal run and share the memory between two host VMs during VM migration. Therefore VMBeam needs resources only for the guest hypervisor (and the management VM) in the extra virtualized system. On the other hand, the amount of resources assigned to host VMs has to be dynamically adjusted according to the resource consumption of running guest VMs. To do this, CPU and memory overcommitment [20] in the host hypervisor can be used. It allows the host hypervisor to assign CPUs and memory that exceed the amount of physical resources to host VMs. In other words, host VMs can share physical resources. VMBeam can assign minimum resources to the extra host VM at first and increase the amount of resources as guest VMs are migrated to it.

3.2 Rejuvenation Target

The target of lightweight software rejuvenation in this paper is the *guest* hypervisor, not the host hypervisor. This is because software aging tends to progress in the guest hypervisor much faster. First, the guest hypervisor is much larger in size than the host hypervisor. This is because most users want feature-rich hypervisors as the guest hypervisor but the host hypervisor needs only minimum functionality. For example, Xen 4.2 is approximately 300K lines of code (LOC), while the host hypervisor named the security monitor in CloudVisor is only less than 6K LOC [22]. In addition, Xen requires the management VM, which runs the regular operating system and various services to help the hypervisor. In general, larger hypervisors suffer from software aging more.

Second, devirtualization during a normal run can suppress software aging of the host hypervisor because the host hypervisor does almost not work for virtualization. Third, as described in Section 2, the guest hypervisor frequently performs complex VM operations such as migration, suspension, and resumption and therefore requires periodic software rejuvenation. In contrast, the host hypervisor does not perform such operations basically.

Consequently, software rejuvenation of the guest hypervisor needs to be more frequent than that of the host hypervisor. This means that lightweight software rejuvenation of the guest hypervisor is more effective for the entire system. It is meaningful to rejuvenate only the guest hypervisor because the hypervisor and VMs are much more loosely coupled, compared with the operating system and applications. Note that rejuvenating guest VMs can be done separately. If software rejuvenation is required for the host hypervisor, it can be done as usual after migrating only one host VM to another host or can be done using fast rejuvenation techniques [11, 13].

3.3 Zero-copy Migration of Guest VMs

Some readers might think that even the traditional migration becomes faster in VMBeam. Unlike traditional systems migrating VMs between two physical hosts, VMBeam runs both the source and destination virtualized systems at the same host. It can transfer the memory image of a guest VM via the virtual network inside a host. However, the virtual network is often slower than the physical network due to the overhead of network virtualization. This leads to the increase of the migration time. In addition, system loads are doubled because one host plays two roles of the client and server in VM migration. The host has to read the memory image of a guest VM, send it to the virtual network, receive it, and write it to a newly created guest VM. A fast virtual network in nested virtualization has been proposed [21], but encryp-

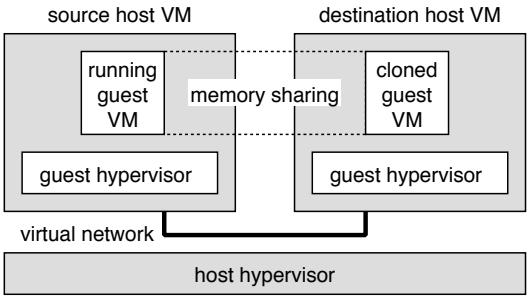

Figure 2. Guest VMs sharing memory during zero-copy migration.

tion of the memory image is still a bottleneck in migration. Such encryption is necessary even in the virtual network to prevent possible eavesdropping by the other guest VMs.

To reduce such overhead, VMBeam provides zero-copy migration of guest VMs between virtualized systems at the same host. Zero-copy migration just *relocates* the memory of a guest VM at a source host VM to a new one at a destination host VM without any copy. It consists of two steps to enable live migration. First, it makes a source guest VM *share* the memory with a destination guest VM so that the source guest VM can continue to run with its memory during VM migration. We call this mechanism *inter-guest memory sharing*. Fig. 2 illustrates a running guest VMs and a cloned guest VM after their memory is shared. Second, zero-copy migration releases the memory of the source guest VM and completes relocating it to the destination guest VM at the final stage of migration.

Although live migration needs multiple iterations for re-transferring modified memory areas, zero-copy migration is completed in only one iteration. Thanks to the memory sharing, modifications to the memory of a source guest VM are directly reflected to the destination guest VM. It is not necessary to transfer modified memory again. This can largely reduce the migration time especially for VMs that execute memory-intensive workloads. It also reduces the time for transferring the remaining memory at the final stage of migration. Since a guest VM is suspended at the final stage, this leads to the downtime reduction.

Since zero-copy migration does not use the virtual network for data transfers, it can decrease the network load. As a result, it does not suffer from the overhead of network virtualization and can also decrease the CPU and memory loads. In addition, it can reduce the copy overhead of a large memory image by simply relocating the memory. Thanks to no copy, the encryption of the memory image is not necessary because any data is not exposed to the outside of source and destination guest VMs. Furthermore, zero-copy migration does not need to detect memory modifications by VMs during migration.

4. Implementation

We have implemented VMBeam in Xen 4.2.2 [1] as a proof of concept. The host hypervisor runs host VMs as fully virtualized (HVM) guests. In a host VM, the guest hypervisor runs guest VMs as HVM guests. In Xen, a virtualized system consists of the hypervisor and the management VM called Dom0. We refer to Dom0s running on the host and guest hypervisors as the *host* and *guest Dom0s*, respectively.

4.1 Memory Management

The host hypervisor manages the physical memory called *machine memory* in the entire machine and allocates a part of it to each host VM. Only the allocated memory becomes the physical memory of a host VM and it is called *host physical memory*. The guest hypervisor in a host VM allocates a part of it to each guest VM. This memory is called *guest physical memory* and becomes the physical memory of a guest VM. For each memory, page frame numbers are consecutively assigned. Machine frame numbers (MFNs), host physical frame numbers (HPFNs), and guest physical frame numbers (GPFNs) are for machine, host, and guest physical memory, respectively. The mapping between MFN and HPFN and that between HPFN and GPFN are maintained in the host and guest hypervisors, respectively.

4.2 Zero-copy Migration

In VMBeam, guest Dom0 at a source host VM transfers the memory image of a guest VM to guest Dom0 at a destination host VM. Without zero-copy migration, a migration client in the source guest Dom0 first maps the memory pages allocated to a guest VM. Then it reads the memory contents of the guest VM and transfers it to a destination host VM via an SSH tunnel constructed on the virtual network [1]. In contrast, zero-copy migration can complete the transfer of the memory image only by passing the GPFNs assigned to the memory of the guest VM to the guest hypervisor. Memory transfer in zero-copy migration requires neither memory mapping, encryption by SSH, nor network transfers.

At the destination host VM, a migration server in the destination guest Dom0 creates a new empty guest VM and maps its memory pages in the traditional migration. Then it writes the memory contents received via an SSH tunnel to them. In contrast, zero-copy migration can complete the receipt of the memory image only by passing the GPFNs assigned to the memory of the new guest VM to the guest hypervisor. Real memory transfers are done by the host hypervisor, which is invoked by the guest hypervisor. The host hypervisor makes a destination guest VM share the memory pages of a source guest VM, using inter-guest memory sharing (See Section 4.3 for the detail). Therefore, memory receipt in zero-copy migration does not need to map memory pages or to execute the decryption by SSH.

[1] This is in the case of the `xl migrate` command. The traditional `xm migrate` command transfers a memory image via an SSL connection.

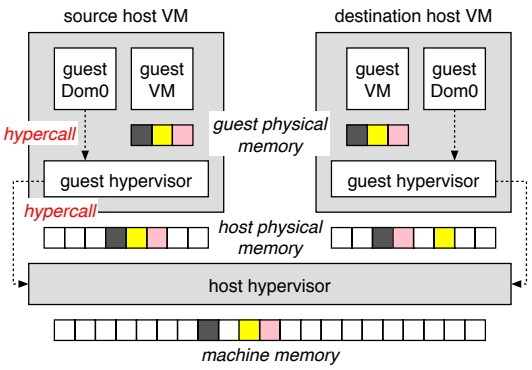

Figure 3. Inter-guest memory sharing across virtualized systems.

Zero-copy migration is completed after it transfers all the memory pages of a VM once. In the traditional migration, the source guest Dom0 runs a migrating source guest VM in the logdirty mode to detect modifications to memory pages by the VM. At the end of each iteration of memory transfers, it obtains a dirty bitmap from the guest hypervisor. A dirty bitmap maintains which page is modified since it is cleared. According to the dirty bitmap, the source guest Dom0 re-transfers modified memory pages. In contrast, zero-copy migration does not need to re-transfer memory pages or to run a guest VM in the logdirty mode, which cause performance degradation of the target guest VM. Nevertheless, all the modifications to the memory pages are reflected simultaneously to a destination guest VM.

4.3 Inter-guest Memory Sharing

For guest Dom0s, VMBeam provides a function for sharing memory pages between guest VMs across different host VMs. As illustrated in Fig. 3, the guest Dom0 in a source host VM invokes the guest hypervisor. At this time, it passes an ID of a target guest VM and an array of GPFNs corresponding to memory pages of the guest VM. Similarly, the guest Dom0 in a destination host VM invokes the guest hypervisor, specifying the same information.

Each guest hypervisor translates the passed array of GPFNs into that of HPFNs and then invokes the host hypervisor. The guest hypervisor performs this translation using virtual Extended Page Table (EPT), which maintains the mapping from GPFNs to HPFNs. Then the guest hypervisor invokes the host hypervisor using a hypercall page [21].

The host hypervisor makes the destination host VM share memory pages of the source host VM on the basis of the passed arrays of HPFNs. It first finds an EPT entry from an HPFN of the source host VM and obtains an MFN in the entry. Then it finds an EPT entry from an HPFN of the destination host VM and sets the obtained MFN to the entry. It releases the memory page corresponding to the original MFN in the entry. Consequently, both HPFNs of the source and destination host VMs are translated into the same MFN by EPT.

5. Experiments

We have conducted several experiments to show the effectiveness of zero-copy migration. We used two PCs with an Intel Xeon E5-2665 processor (8 cores, 2.40 GHz), 32 GB of memory, and gigabit Ethernet. These PCs were connected via a gigabit Ethernet switch.

5.1 Experimental Setup

To compare VMBeam with the existing systems, we conducted experiments for three systems: VMBeam, Xen-Phys, which uses two hosts running Xen without nested virtualization, and Xen-Blanket [21]. Xen-Blanket introduces a paravirtual network driver in guest Dom0s and enables direct communication with host Dom0 to improve the performance of the virtual network.

For VMBeam and Xen-Blanket, we used Xen 4.2.2 supporting nested virtualization as the host hypervisor and ran two host VMs on top of it. In each host VM, we used Xen 4.2.2 or Xen-Blanket 4.1.1 as the guest hypervisor. In addition, we ran one guest VM on one host VM. To conduct experiments in the same conditions as much as possible, we used an HVM guest in Xen-Blanket. The original Xen-Blanket used a PV guest because it assumed the host hypervisor without special support for nested virtualization. We ran Linux 3.2.0 in host Dom0 and Linux 3.5.0 in guest Dom0. We assigned three physical CPUs and 10 GB of memory to each host VM. Among the resources assigned to each host VM, we assigned one CPU and memory between 128 MB and 4 GB to a guest VM.

For Xen-Phys, we used Xen 4.2.2 as the hypervisor and ran one VM in one host. For brevity, we also call this VM a guest VM in the following. We assigned the same resources as the above guest VM to this VM.

5.2 System Loads

We examined system loads per host while we migrated a guest VM with 4 GB of memory. First, we measured the size of network data transferred in Dom0. Since VMBeam did not use the virtual network for transferring the memory image, the total network transfer was less than 0.003% of Xen-Phys, as shown in Fig. 4a. The reason why the total network transfer was not zero is that VMBeam still uses the virtual network to transfer several data such as VM configuration due to implementation issues.

Next, we measured the CPU utilization in Dom0. The maximum CPU utilization in VMBeam was twice as large as that in each host of Xen-Phys. However, as shown in Fig. 4b, the total CPU time in VMBeam was 29% of the source host and 31% of the destination host in Xen-Phys. This is because the migration time in VMBeam was much shorter than that in Xen-Phys.

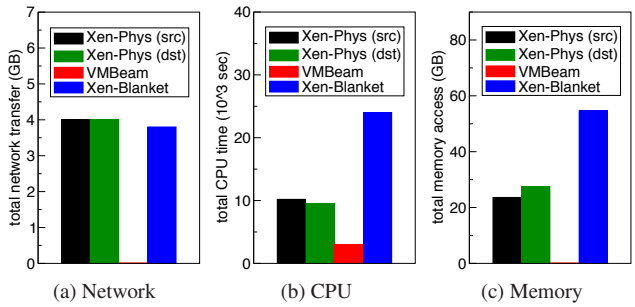

Figure 4. The system loads per host.

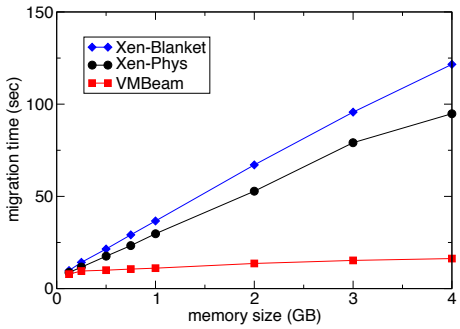

Figure 5. The migration time.

Since we could not obtain the statistics of memory accesses directly from hardware, we estimated them from the number of transferred memory pages and the memory access pattern in each system. We consider only the transfer of the memory image because that is a dominant factor in VM migration. In VMBeam, the host hypervisor relocates all of the memory pages of a source guest VM to a destination one. Therefore VMBeam needs no memory access. The total memory access in the other systems is shown in Fig. 4c.

5.3 Migration Performance

The time needed for VM migration is shown in Fig. 5. The results show that the migration time in VMBeam does not increase largely as the memory size of a guest VM becomes larger. VMBeam achieved the shortest migration time and could complete the migration of a VM with 4 GB of memory in 16 seconds. Compared with Xen-Phys, VMBeam could execute VM migration 1.1 to 5.8 times faster. However, the VM migration in Xen-Blanket was 1.1 to 1.3 times slower.

Next, we measured the time during which a guest VM stopped due to VM migration. As shown in Fig. 6, this downtime in VMBeam was about 0.6 second, but it is about 0.2 second longer than the downtime in Xen-Phys. This is due to the overhead of nested virtualization in Xen. To obtain the CPU state of a guest VM at the final stage of migration, the migration client has to issue many hypercalls to the hypervisor. This largely suffers from the impact of nested virtualization. In Xen-Blanket, the downtime was 0.6 to 0.9

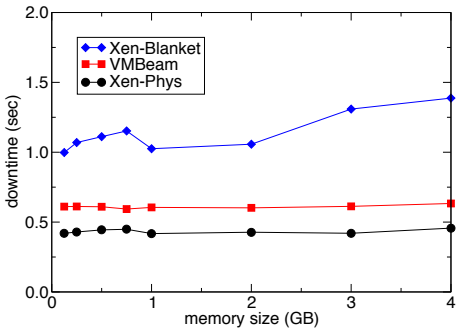

Figure 6. The downtime during VM migration.

second longer than that in Xen-Phys, but the reason is under investigation.

6. Related Work

On system maintenance in a VM, Microvisor [14] starts another VM and performs the maintenance for the system in the new VM. After the maintenance, it migrates applications from the original to the new VM. VMBeam is similar to Microvisor in that it runs old and new VMs at the same host and migrates components that are not maintained. One difference is that VMBeam uses nested virtualization for running two virtualized systems. Another and bigger one is that VMBeam focuses on VM migration for lightweight software rejuvenation. Microvisor focuses on devirtualization for disabling virtualization during a normal run.

RDMA-based migration over InfiniBand [6] needs only one copy by hardware, i.e., zero copy in software. The memory image of a VM is directly copied to the memory of a newly created VM at a destination host using Remote Direct Memory Access (RDMA). However, it still needs to retransfer memory modified during migration. In addition, it cannot encrypt the memory image although the data is transferred via the network. With the encryption of the memory image, RDMA-based migration needs three copies.

Many techniques for page sharing between VMs have been developed mainly for saving memory. VMware ESX Server scans the memory of VMs periodically and makes VMs share identical pages [20]. Satori [16] can detect short-lived sharing opportunities by using sharing-aware block devices. Difference Engine [5] shares not only identical but also similar pages between VMs. Flash cloning in Potemkin [19] creates a new VM from a reference VM image and enables page sharing between the reference and new VMs. Since all of these techniques use copy-on-write, page sharing is ceased when shared pages are modified.

TinyChecker [18] detects hypervisor failures and recovers the hypervisor using nested virtualization. The tiny host hypervisor traps VM exits from guest VMs to the guest hypervisor and detects crash, hang, memory corruption, and silent failure. However, it is difficult to detect software aging that progresses by correct operations, e.g., memory leakage.

Several systems have been proposed to support devirtualization [3, 9, 10, 14, 17]. All of them incur negligible performance degradation after the system is devirtualized. Microvisor [14] is the first system supporting for devirtualization, but it does not virtualize memory or I/O devices. In addition, it depends on the Alpha architecture and redundant hardware. In contrast, Mercury [3] enables dynamically attaching and detaching a full-fledged hypervisor. The guest operating system switches native and virtualized modes by itself. To apply this self-virtualization to nested virtualization, the guest hypervisor and the operating system in the guest management VM would have to be modified largely. Unlike these, BMcast [17] can devirtualize shared I/O devices as well as CPU and memory at the hypervisor level. Thanks to device mediators in the hypervisor, it does not need any modifications to the guest operating system. A drawback is that a device mediator has to be implemented for each I/O device.

7. Conclusion

In this paper, we proposed VMBeam, which enables lightweight software rejuvenation of virtualized systems. Using nested virtualization, VMBeam starts another virtualized system at the same host when it rejuvenates an aged virtualized system. Using zero-copy migration, it efficiently migrates the VMs running on the aged virtualized system to the new one. We have implemented VMBeam in Xen and showed that VMBeam could achieve more lightweight software rejuvenation than the existing systems.

One of our future work is to develop a minimal host hypervisor such as CloudVisor [22]. Although we currently used Xen as a proof of concept, we need the host hypervisor that suffers from software aging less frequently than the guest hypervisor. Another direction is to enable devirtualization [14] in the host hypervisor. This can reduce the overhead of nested virtualization during a normal run.

Acknowledgments

This research was supported in part by JSPS KAKENHI Grant Number 25330086.

References

[1] 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 *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 164–177, 2003.

[2] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour. The Turtles Project: Design and Implementation of Nested Virtualization. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[3] H. Chen, R. Chen, F. Zhang, B. Zang, and P.-C. Yew. Mercury: Combining Performance with Dependability Using Self-virtualization. In *Proceedings of the 2007 IEEE International Conference on Parallel Processing*, 2007.

[4] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation*, pages 273–286, 2005.

[5] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelkerrey, and A. Vahdat. Difference Engine: Harnessing Memory Redundancy in Virtual Machines. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, pages 309–322, 2008.

[6] W. Huang, Q. Gao, J. Liu, and D. K. Panda. High Performance Virtual Machine Migration with RDMA over Modern Interconnects. In *Proceedings of the 2007 IEEE International Conference on Cluster Computing*, pages 11–20, 2007.

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

[8] Intel Corp. 4th Generation Intel Core vPro Processors with Intel VMCS Shadowing, 2013.

[9] E. Keller, J. Szefer, J. Rexford, and R. B. Lee. NoHype: Virtualized Cloud Infrastructure Without the Virtualization. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, pages 350–361, 2010.

[10] T. Kooburat and M. Swift. The Best of Both Worlds with On-demand Virtualization. In *Proceedings of the 13th USENIX Workshop on Hot Topics in Operating Systems*, 2011.

[11] K. Kourai and S. Chiba. A Fast Rejuvenation Technique for Server Consolidation with Virtual Machines. In *Proceedings of the 37th IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 245–255, 2007.

[12] K. Kourai and S. Chiba. Fast Software Rejuvenation of Virtual Machine Monitors. *IEEE Transactions on Dependable and Secure Computing*, 8(6):839–851, 2011.

[13] M. Le and Y. Tamir. ReHype: Enabling VM Survival Across Hypervisor Failures. In *Proceedings of the 7th ACM International Conference on Virtual Execution Environments*, pages 63–74, 2011.

[14] D. E. Lowell, Y. Saito, and E. J. Samberg. Devirtualizable Virtual Machines Enabling General, Single-node, Online Maintenance. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 211–223, 2004.

[15] F. Machida, J. Xiang, K. Tadano, and Y. Maeno. Combined Server Rejuvenation in a Virtualized Data Center. In *Proceedings of the 9th IEEE International Conference on Autonomic and Trusted Computing*, pages 486–493, 2012.

[16] G. Miłós, D. G. Murray, S. Hand, and M. A. Fetterman. Satori: Enlightened Page Sharing. In *Proceedings of the 2009 USENIX Annual Technical Conference*, 2009.

[17] Y. Omote, T. Shinagawa, and K. Kato. Improving Agility and Elasticity in Bare-metal Clouds. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 145–159, 2015.

[18] C. Tan, Y. Xia, H. Chen, and B. Zang. TinyChecker: Transparent Protection of VMs against Hypervisor Failures with Nested Virtualization. In *Proceedings of the 2nd IEEE/IFIP International Workshop on Dependability of Clouds, Data Centers and Virtual Machine Technology*, 2012.

[19] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. C. Snoeren, G. M. Voelker, and S. Savage. Scalability, Fidelity, and Containment in the Potemkin Virtual Honeyfarm. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, pages 148–162, 2005.

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

[21] D. Williams, H. Jamjoom, and H. Weatherspoon. The Xen-Blanket: Virtualize Once, Run Everywhere. In *Proceedings of the 7th ACM European Conference on Computer Systems*, pages 113–126, 2012.

[22] F. Zhang, J. Chen, H. Chen, and B. Zang. CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization. In *Proceedings of the 23rd Symposium on Operating Systems Principles*, pages 203–216, 2011.