Stony Brook University

Optimization Techniques for Memory Virtualization-based Resource Management

A Dissertation Presented

by

Jui-Hao Chiang

to

The Graduate School

in Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

Stony Brook University

December 2012

Stony Brook University

The Graduate School

Jui-Hao Chiang

We, the dissertation committee for the above candidate for the Doctor of Philosophy degree, hereby recommend acceptance of this dissertation.

Tzi-cker Chiueh – Dissertation Advisor
Professor, Department of Computer Science

Jie Gao – Chairperson of Defense
Associate Professor, Department of Computer Science

Rob Johnson
Assistant Professor, Department of Computer Science

Ted Teng
Professor, Department of Technology and Society

This dissertation is accepted by the Graduate School.

Charles Taber
Interim Dean of the Graduate School

Abstract of the Dissertation

Optimization Techniques for Memory Virtualization-based Resource Management

by

Jui-Hao Chiang

Doctor of Philosophy

in

Computer Science

Stony Brook University

2012

Memory virtualization abstracts the physical memory resources in a virtualized server in such a way that offers many resource management advantages, such as consolidation, sharing, compression and migration. The main goal of this dissertation project is to develop optimization techniques to resource management schemes based on memory virtualization. Although migration of virtual machine (VM) memory state is a standard feature of most modern hypervisors, migration of physical machine memory state is largely non-existent. We applied the standard VM migration technique to building the first known physical machine state migration system for Linux servers, which significantly increases the system management flexibility for physical machine administration. Virtual machine introspection (VMI) allows the internal states of a VM to be analyzed. We exploited VMI to identify free memory pages, and leverage this knowledge to significantly improve the efficiency of memory de-duplication and memory state migration. To analyze undocumented data structures in different kernel versions, we

developed a novel memory analysis procedure that programmatically takes advantage of the availability of guest kernel source code when it exists. To further increase memory utilization, we propose an adaptive memory compression scheme that makes better use of the physical memory resources of virtualized servers by accurately and efficiently tracking the working sets of individual VMs. Finally, cloning a VM involves copying of the VM's memory pages. To minimize the memory copying overhead when cloning a VM on the same physical machine, we propose a lazy memory state creation scheme that defers the copying of a cloned VM's memory pages and its memory mapping tables to the last possible moment.

Contents

List of Figures

List of Tables

Acknowledgements

Thanks to the uninterrupted guide of my advisor, Professor Tzi-cker Chiueh, through the entire PhD life. From him, I have learned the way to move forward while facing obstacles. I also thank to my committee members Professor Jie Gao, Professor Rob Johnson, and Professor Ted Teng for their guidance on my dissertation.

Thanks to the ECSL lab members. Maohua Lu has cooperated with me on part of the dissertation and helped on solving several difficult kernel problems during the software development. Cheng-Chun Tu, Dilip N Simha, Xiang Gao, and Sun Yifeng have always been there to listen to my questions and then share their great ideas. All these members help me to walk through the work time in the lab.

During the last two years of my PhD life, I specially thank to Han-Lin Li, who has paid a great amount of time in solving questions and performing experiment with me on various kinds of research issues. He is open-minded in discussion and agile when handling research problems.

Finally, thanks to the love and support from my parents and my better half, Wan-Chiu Chen.

Chapter 1

Introduction

In the virtualized environment, the *Virtual Machine Monitor* (VMM), *hypervisor* in short, manages the hardware resource for VMs running on top of it, which resembles to the way that traditional operating system (OS) manages the hardware resources for processes. The VMM abstracts the hardware resources including CPU, memory, and I/O devices from the underlying bare-metal hardware of the physical machine, and then allocates them as virtual CPU, virtual memory, and virtual I/O devices to the VMs running on top of it. The system administration in the virtualized environment requires the same kind of flexibility for system fault tolerance, low-level maintenance, and load balancing as that of the traditional data center without virtualization techniques.

In the following sections, we first introduce the concept of virtualization, and talk about various applications of it with their vanilla designs and limitations. We are motivated by finding that these applications are either lacking important features or come with poor performance mostly due to the immaturity of its memory virtualization technique. As a result, we proposed the corresponding optimization mechanisms to these applications.

1.1 Server Virtualization Technology

In the traditional computer architecture as Figure 1.1 shows, the OS is running directly on top of bare-metal hardware, which is responsible for probing hardware devices, setting up communication channels and configuring them with appropriate hardware drivers from various hardware vendors. After proper setup of hardware, the OS provides them as system resource including CPU, memory, and I/O devices. Later on, the OS is able to allocate these system resource to upper processes or applications (APs) based on its own policy. Tak-

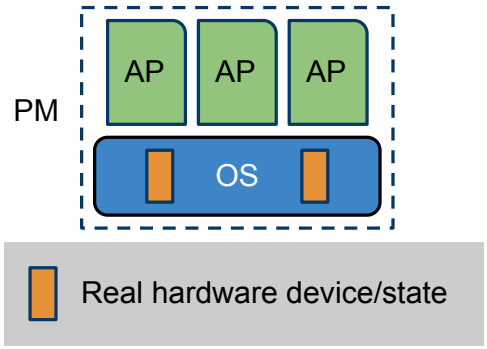

Figure 1.1: *Software architecture of physical machine. The operating system (OS) is running on bare-metal hardware and applications (APs) are running as processes/threads scheduled by OS.*

ing CPU as an example, the OS can allocate CPU cycles to each process using time-sharing concept so that the process or the corresponding application can perform its own work by sharing the CPU resource with other applications. And the application can serve for any purpose based on the requirement of user or system administrator, e.g., office applications for desktop users and web server for Internet service providers.

As Figure 1.2 shows, unlike the traditional computer architecture, the virtualization technology has another piece of software, the Virtual Machine Monitor or *hypervisor* in short, to run on top of the bare-metal hardware. It sits at the same level as the OS in the traditional architecture, and has the same kind of responsibility to manage hardware resource. The hardware including CPU, memory, and I/O devices are abstracted as virtual CPU, virtual memory, and virtual I/O devices to the upper running VMs. This technique is referred to as *Hardware Abstraction Layer* based or **HAL-based virtualization**. From the viewpoint of hypervisor, VMs are just similar to processes or applications in traditional computer architecture. The VMs run on these virtual hardware have an illusion that they are running on bare-metal hardware, and have their own OSs, processes or applications running inside as usual. The VM can also be referred to as *guest* and the OS running inside VM is referred to as *guest OS*.

The hypervisor schedules the virtual CPUs of VMs as its basic scheduling unit, which is similar to scheduling processes in traditional OS. Also, the physical memories are allocated to VMs by hypervisor as the traditional OS assigns virtual memory to processes. Finally, all hardware I/O requests for the VMs are served by hypervisor, and sent to or received from VMs by using event channels.

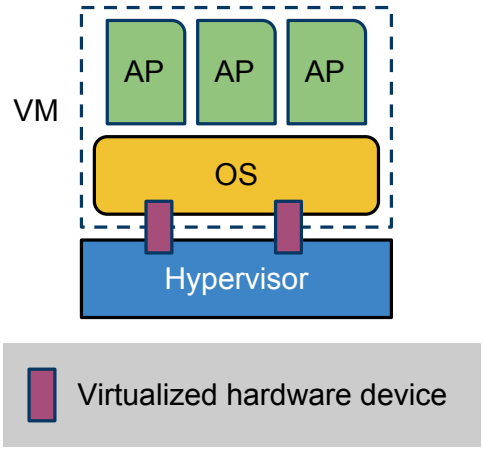

Figure 1.2: *Software architecture of Virtual Machines. The Virtual Machine Monitor (VMM) or hypervisor in short is running on bare-metal hardware, and Virtual Machines are scheduled by hypervisor as processes/threads running on the traditional OS.*

In the cloud or virtualized environment, there are several important applications among the area of resource consolidation, fault-tolerance, and malware detection for security purposes. These applications include VM migration, VM memory de-duplication and compression, and VM state cloning. The memory virtualization technique plays an important role in these applications, but has various issues and open questions left. Now we first introduce the concept of memory virtualization and then go through all these applications along with our proposed work.

1.2 Memory Virtualization

In traditional computer architecture, OS is responsible for allocating physical memory to processes which run inside its own process virtual address space. When accessing memory, the process virtual address must be translated by hardware *memory management unit* (MMU) to traverse the corresponding page table setup by the OS. When running inside the hypervisor environment, one more in-direction of memory address translation is performed.

As Figure 1.3 shows, when running on the hypervisor, the guest OS maintains a mapping from *guest virtual* addresses (GVA) to *guest physical* addresses (GPA), and the hypervisor translates from *guest physical* to *machine physical* addresses (MPA), i.e., the real physical addresses used to access the memory. Accordingly, the Machine Frame Number (**MFN**) is a page number in the

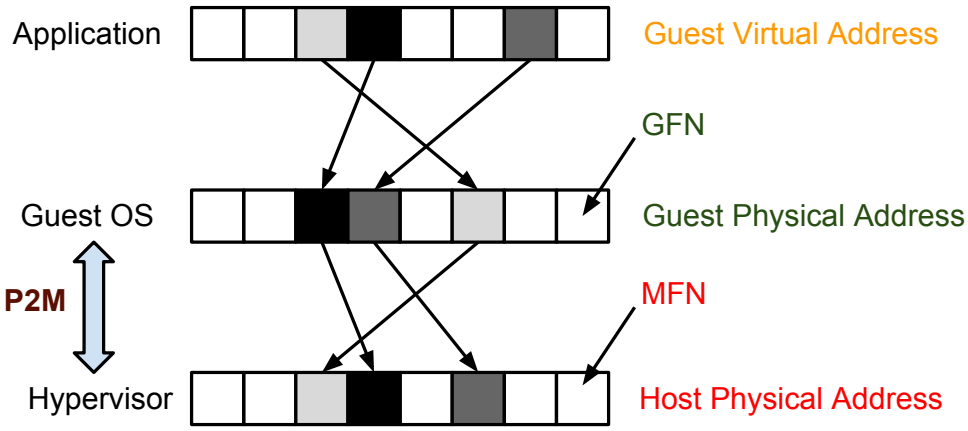

Figure 1.3: *Three kinds of memory addressing mode in hypervisor context.*

machine physical address space whereas the Guest Frame Number (**GFN**) is a page frame number in the guest physical address space. Normally, each GFN of a guest OS is mapped to a unique MFN allocated by the hypervisor. And the hardware page table used by hypervisor to translate from guest physical to machine physical address is usually referred to *Guest physical to machine physical* (P2M) table. In modern CPU architecture, most CPU now has hardware support for memory virtualization techniques. Taking Intel VT (Virtualization Technology) [1] as an example, when Extended Page Table (EPT) feature is enabled in the Intel CPU, the hardware MMU, for each memory access from guest VM, first walk through the page tables used by guest OS to translate from GVA to GPA, and then walk a separate page table, i.e., EPT, setup by hypervisor to translate from GPA to MPA. Thus, the conceptual P2M table we have mentioned above just maps to the EPT table in the case of Intel architecture.

1.3 Memory Resource Optimization

The virtualized environment is currently memory-bound [2–4], i.e., the physical memory is the bottleneck of the resource utilization in large data centers. In terms of memory virtualization, the following applications are crucial to the resource management and system administration. Each of them is summarized as the following and detailed in later sections.

- Virtual Machine Migration: move VMs from highly-utilized hosts to lower-utilized ones to balance the workload of physical hosts and consolidate hardware resource.

- Memory De-duplication of Virtual Machines: remove duplicate memory pages between VMs to increase memory utilization.
- Memory Compression of Virtual Machines: compress memory page content to further increase memory utilization.
- Virtual Machine Cloning: clone the state of a source VM so that the cloned VM can be used as a sandbox environment for security or software development purposes.

The same background of these applications is that the memory resource is ***difficult to share*** among different hosts or different VMs on the same host. When balancing the system resource between different hosts, the memory state can not be easily shared as the disk, e.g., iSCSI [5] technique to share disk as the network volume, and the memory content has to be moved from one host to another. With virtualization technique, the VM migration can easily move the memory state between hosts due to the hardware abstraction, but unfortunately there is no existing capability for physical machines. In addition, the VM migration application has to copy large volume of memory through the network and consumes a significant amount of network bandwidth.

On the other hand, the memory resource on the same host can not be easily shared among VMs as the CPU resource, whose cycles can be distributed to each VM using the time-sharing policy, and the memory resource becomes monopolized by each VM and the overall utilization is limited. The memory de-duplication and compression applications try to increase the memory utilization but then have bottlenecks on the VM performance degradation due to the resource sharing. Finally, the VM cloning suffers the performance overhead by copying the large volume of memory content from the source to cloned VM. In the following sections, we describe these applications in detail, focus on their problems of memory virtualization, and propose our corresponding optimization techniques.

1.4 Virtual Machine Migration

The simplest approach to shift or balance the workload of physical hosts is to move their corresponding VMs from highly-loaded hosts to lower ones. The VM migration mechanism are designed to migrate the machine states of VM from assigned source physical host to the destination one without interrupting any ongoing services running inside the VM. Traditionally in PM, the machine state consists of the state of CPU (e.g., registers and instruction pointer), memory content where all the program data or OS data structures are held,

and all the state of I/O devices such as hardware registers of network interfaces. With all these states to be consistent, a machine or the running OS on it can work properly and provide services locally or remotely on Internet. The VM states are consist of the same thing as regular machine states except all it has are virtual hardware resource. Note that the disk content will not be considered as the VM states because it is far more too large for migration. Thus, the source and destination machine of the migration always share the same disk for the migrated VM, e.g., using the iSCSI technique[5]. For the rest of the VM states, the **memory state is dominant** because the states of CPU and I/O devices are just bunch of registers each with a few bytes while the memory size can go up to GBs.

The consistency is the first criteria to complete the VM migration, e.g., during the network transfer of the VM states, the state of the network card should be separated from the transferred VM state. Otherwise, it can be devastating to the migrated VM whose state is changed and no longer consistent during the migration. To catch the machine state with ensured consistency one must suspend all system activities including CPU, memory, and I/O devices, and take a snapshot of it. These steps are extremely easy for VMs with the underlying hypervisor support where the hypervisor can suspend the VM activity as suspending regular process in traditional OS. And the hypervisor is able to take the snapshot of all VM states, which can be used to restore or re-construct a VM when necessary.

The high-level procedure of VM migration is described as follows.

- First, all system activities of the migrated VM are suspended by hypervisor. Thus, from now on, no virtual CPU resource is given to VM, and no I/O devices requests are served by hypervisor for the VM.

- Second, the hypervisor establishes network connection between source and destination physical hosts, either locally if the source and destination physical hosts are the same or remotely if they are different. Note that the network connection of hypervisor will not make any change to the saved states of migrated VM and thus the consistency is always kept.

- Third, take the snapshot of states of virtual CPU and I/O devices of migrated VM. Together with the memory content of the VM, the VM state are formed.

- The VM states are transferred to the destination machine.

- The hypervisor of destination machine constructs the system environment of the migrated VM by allocating virtual CPU, memory and I/O

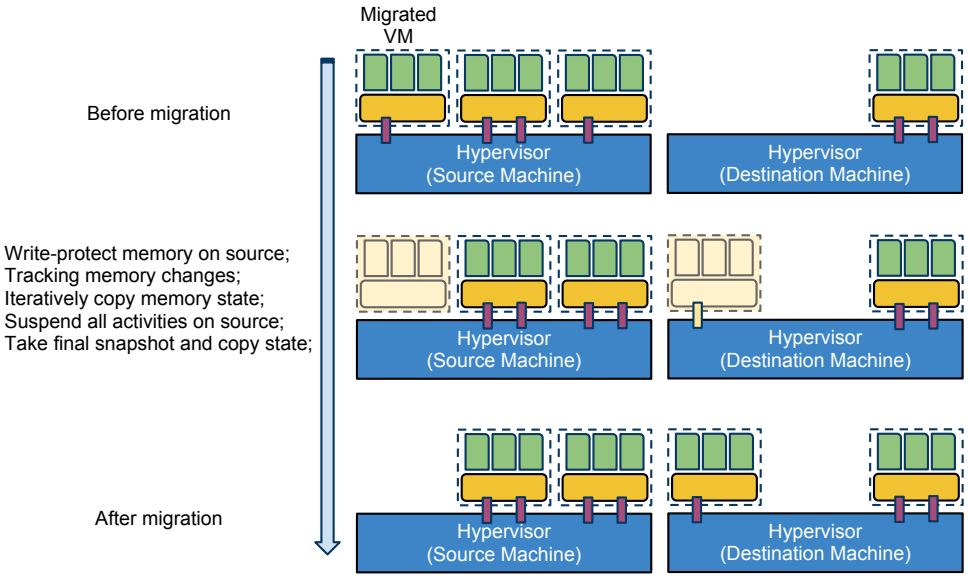

Figure 1.4: *Virtual Machine Live Migration mechanism where the migrated VM is being moved by underlying hypervisors from source to destination machine.*

devices. Afterwards, all the states of migrated VM are restored within the newly created environment, and the VM is resumed to continue its original work.

- The hypervisor of source machine releases all resource for the migrated VM, and finish the entire operation.

Although the above migration mechanism guarantees the consistency of VM states, but the applications running inside guest OS will notice disruption during these operations, mostly due to the memory state transfer because its size can go up to a few GBs. Thus, VM *live migration* or Pre-copy mechanism is designed [6] to transfer the **memory state** while not causing too much service downtime for applications. As Figure 1.4 shows, the way to achieve is to run the migration mechanism iteratively as described in the following.

- First, all system activities of migrated VM are suspended as normal VM migration.

- Second, all memory of migrated VM are marked as read-only, so that any modification to the memory of VM will be trapped by hypervisor, which is the same as *Copy-on-Write* (COW) as we will discussed more details in the next section. Here, when memory is modified, they are marked as *dirtied* memory.

- Third, the system activity is resumed again without interrupting the service for too long.
- Fourth, the source machine sends all memory state that have not been transferred to the destination including the dirtied memory since the previous iteration.
- If there are still lots of states left for transfer, the flow goes back to the first step to start another iteration. Otherwise, the VM activities are suspended again, and all left dirtied memory are transferred to destination.
- In the end, the VM states except for memory part are taken as snapshot and sent to destination. The source machine destroys the VM environment while the destination re-constructs it as regular VM migration.

Because of the hardware abstraction of virtualization, the underlying hypervisor can act as a separate software to perform the state transfer of the VM between the source and destination physical host. Therefore, the *VM migration* enables this unprecedented flexibility for system administration in the cloud by moving the VM from one physical host to another without interrupt to any application running inside the VM.

However, no similar capability exists for physical machines. Thus we propose the *Physical Machine State Migration* (PMSM) mechanism to generalize the machine state migration concept to PMs, which is more challenging than VM migration because it cannot rely on a separate piece of software to perform the state transfer, e.g., the hypervisor in the case of VM migration. The PMSM prototype proposed in this dissertation is adapted from Linux's hibernation facility.

1.5 Memory De-duplication of Virtual Machines

On the other hand, a major design goal of modern hypervisors is to run as many VMs on a physical machine as possible. As researches and commercial reports [2–4] indicate, the maximum number of VMs that can run on a physical machine and still exhibit decent performance, known as the *virtualization ratio*, is mostly limited by the amount of physical memory installed on that PM. Therefore, the key to maximizing the virtualization ratio is to improve the memory utilization efficiency for which the *memory de-duplication* technology was developed with target on VMs.

Traditional memory de-duplication involves identifying common memory pages and storing only one copy for each of these pages, and common pages

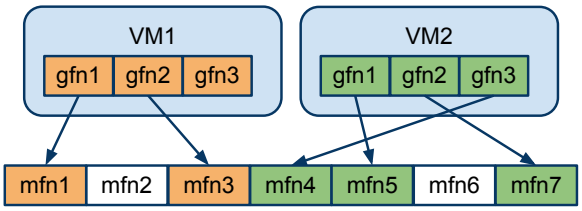

Figure 1.5: *Mapping of GFNs to MFNs in the hypervisor context where we take two virtual machines VM1 and VM2 with limited number of GFNs and MFNs as an example.*

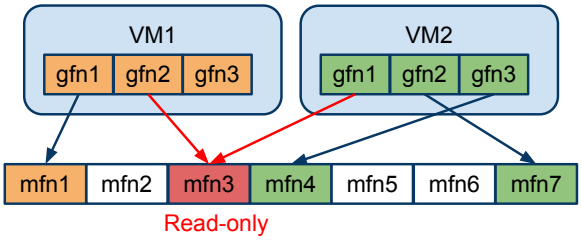

Figure 1.6: *Mapping of GFNs to MFNs in the hypervisor context where we take two virtual machines VM1 and VM2 with limited number of GFNs and MFNs as an example. The content of GFN2 from VM1 and GFN1 of VM2 are found to be the same, and thus both of these two GFNs point to the same MFN3.*

are identified by computing the hash value from each physical memory page's content and checking it with the hash values of other physical memory pages on the same machine. Taking two VMs, VM1 and VM2, as an example shown in Figure 1.5, two MFNs 1 and 3 are being assigned to VM1 and mapped to GFN 1 and 2 respectively while MFNs 4, 5, 7 are mapped to GFN 1, 2, 3 of VM2 in the P2M table maintained by hypervisor. Assume that after hash computation MFN 3 and 5 are found to be the same, the corresponding mapping of these two MFNs will be marked as read-only in the P2M table. As Figure 1.6 shows, right after the hypervisor compares their content byte-by-byte and make sure they are exactly the same, the hypervisor discards MFN 5 and maps both GFN 2 of VM1 and GFN1 of VM2 to MFN 3. The mechanism for pointing multiple GFNs to a single MFN is referred to as *memory sharing* in the hypervisor context. When any write access is performed to the shared MFN3, the operation will trigger a hardware page fault and trap into hypervisor page fault handler. Then hypervisor will assign a new MFN for the trapped VM, and copy the content from the original MFN 3, which is referred to as the COW mechanism similar to the traditional OS. Also the read-only protection

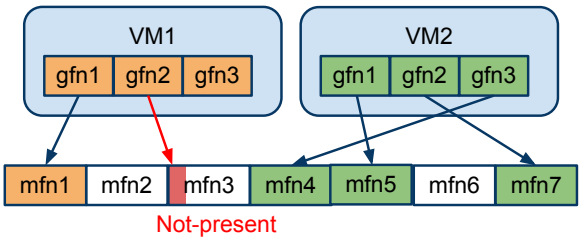

Figure 1.7: *Mapping of GFNs to MFNs in the hypervisor context where we take two virtual machines VM1 and VM2 with limited number of GFNs and MFNs as an example. Here the content of MFN3 mapped by GFN2 of VM1 is compressed to save memory space.*

is removed for the new MFN.

However, the hash computation and content comparison of memory pages consume lots of CPU cycles of hypervisor, and makes it infeasible as the memory size of VMs increases. Thus, we propose a promising new memory-deduplication technique that significantly improves the de-duplication gain and speed by taking into account a physical memory page's *type* in addition to its *content*. More concretely, this technique focuses on the free memory pool of guest virtual machines (VMs). Because the pages in the free memory pool contain *don't-care* contents, they could in theory be treated as duplicates to a zero page, and de-duplicated accordingly, which generalizing the content-based approach to the don't-care page content. However, discovering pages in a guest VM's free memory pool is non-trivial. In our work, we have proposed and developed a *bootstrapping VMI* (Virtual Machine Introspection) technique, and applied them to discovering free memory pages required by the proposed generalized memory de-duplication scheme.

Moreover, In the middle of developing the bootstrapping VMI technique, we reminded that the network bandwidth consumed by VM live migration is mostly due to transferring memory pages. If we are able to find redundant page without sending them to the destination machine, it may reduce the overhead significantly with extremely low overhead from VMI technique. Thus, we have proposed an *Introspection-assisted VM Migration* mechanism to identify free memory pages during VM migration and skip them for transfer to save network bandwidth and time consumption of the entire process.

1.6 Memory Compression of Virtual Machines

Similar to the conventional OS, the last resort to increase memory utilization of hypervisor is to reclaim memory from guest VM by *host swapping*, i.e., to

copy the memory pages of VMs to host swap disk, referred to as *swap-out*, mark the corresponding PTE (Page Table Entry) of the VM's P2M table to be not-present, and then free the corresponding page to the free memory pool of hypervisor. Later on, if the page is accessed again by the owner VM, *page fault* is triggered and the Copy-on-Access (**COA**) mechanism is performed to bring the page content from swap disk into a newly allocated memory page, referred to as *swap-in*. However, the overhead is far worse than the memory de-duplication because of the long latency of disk I/O.

As another way to increase the memory utilization, memory compression of VMs can be done by compressing the swapped-out pages of VMs into smaller size of data, and putting them together in memory to save the number of physical memory used to store the original content. The idea is that the swapin from compressed memory, referred to as ***pseudo page fault***, would be faster than the swapin from disk, referred to as ***true page fault*** because the memory access is faster than the disk access. To describe the basics, we refer to the mechanism used by the VMware ESX server. As Figure 1.7 shows, when GFN 2 of VM1 is compressed, the mapping in the P2M table must be marked as *not-present*. When any read or write access is performed on the compressed page, it will trap into hypervisor for COA mechanism, which allocates a new page (MFN), decompresses the page data, and stores it to the newly allocated MFN. The page mapping will be restored to its original access.

However, most research works such as VMware put the compression as a secondary choice because it not only causes the COA, which triggers hardware trap and stops the current application execution, but also consumes the host CPU cycles to compress and decompress the page content and incurs more overhead than the COW of memory de-duplication. Thus, the ideal situation is to avoid compression for the memory pages that are accessed frequently by guest OS, i.e., the *working set*, but to find out the idle memory pages, i.e., the guest memory pages outside of the working set, for memory compression.

In this dissertation, we proposed the *Working Set-based Memory Allocation and Compression* to identify guest working set by leveraging the page reclamation mechanism of guest and compress pages outside of it. We aim to pack more VMs onto the same physical host without losing the application performance. With proper amount of compression, for example, we may be able to turn a machine with memory size X into a machine with memory size Y with similar performance in the application-level where Y is greater than X. By leveraging the existing page reclamation mechanism of guest OS, e.g., guest swapping is one method[1], we have designed and implemented a *True*

[1]It will be detailed in Chapter 2.6.

Working Set-based Ballooning mechanism, referred to as *TWS-ballooning*, to probe guest working set and evict idle memory pages as the target of compression. For implementation, we focus on Linux guest OS specifically and expect the mechanisms we have designed here can be easily leveraged to other guest OSs such as Windows.

As to compress the swapped-out guest idle memory, we leverage zram [7] driver in Linux OS, which presents as a swap disk in guest, compresses and stores the swapped-out pages in guest memory. With zram, a pseudo page fault would trigger the compressed page of zram to be decompressed and stored into a newly allocate guest memory page, which is intuitively faster than the real page fault from swap disk. However, to store the compressed pages in zram, the guest OS needs to consume guest memory and could bring more swap in/out operations. To resolve this dilemma, we propose a *dyn_memlimit* mechanism to dynamically adjust the zram size and address it in chapter 6.4.

Moreover, when there are multiple VMs running on a host with low free memory, the memory distribution to VMs becomes important to the application performance. To prevent the VMs from severe performance degradation due to insufficient memory, we have proposed several *Memory BaLancing* (**MBL**) mechanisms to equalize the performance overhead of all VMs running on the same host so that each VM can degrade gracefully without starving on memory resource. To achieve this, we intercept guest kernel events, e.g, swapin, to quantify the performance overhead of guest VM and adjust the VM allocated memory to equalize the overhead of each VM.

1.7 Virtual Machine Cloning

As the hypervisor isolates the operations between each VM running on top of it, the program running inside a particular VM will not affect the states of other VMs. Thus, the confined VM environment can be used for several purposes, and examples are listed as the following.

- Opening DRM (Digital rights management) documents without leaking copyright information,
- Executing suspicious program and analyzing its behavior with antivirus software without tampering the host and source VM environment, i.e., as a sandbox,
- Testing software patches for system upgrade or analyzing bugs for software development,

- Acting as honeypot [8] for production servers, i.e., a server system with the similar environment as the production servers but is isolated and monitored to capture information of intrusions and attacks. The gathered information is used to analyze the methods of attacks to prevent compromise on the real production system.

It can be very easy if one can *lively* clone a VM from the source one, and perform the above operations inside the newly cloned VM, e.g., when a user opens a DRM document, the application can request a live VM cloning of the current environment, and present the document to user in the cloned VM with networking turned off. Therefore, the user can not infringe the copyright by distributing the DRM document to the Internet. All these require the VM cloning to be done lively, timely, and as light-weight as possible.

In the *OS-level* virtualization, e.g., Jail [9] and FVM [10], the VM can be cloned as light-weight as a process forking in the traditional OS. This is because the OS-level virtualization abstracts the important OS data structures, e.g., file system descriptors, and leaves the VMs to run directly on the underlying bare metal hardware. Comparing to HAL-based virtualization, the OS-level virtualization is difficult for implementation and less portable because it requires the specific domain knowledge of an OS, e.g., the structure of Linux file system descriptors and how they are accessed, which is complicated and different from one version of OS to another.

With the popularity of HAL-based virtualization in data centers, it is desired to build up a HAL-based VM cloning mechanism which is as light-weight as the VM cloning in OS-level virtualization, and serves as sandbox utility in the virtualized environment. Similar to the VM migration, most of the work in VM cloning is to clone the memory state. Without copying the entire memory state at the moment of cloning, the cloning time can be reduced by copying P2M table, the mapping of GFNs to MFNs, from the source VM to the cloned one, and setting all memory pages as read-only, which is the same as sharing memory pages between source and clone VMs. Upon a write access, either from source or cloned VM, the COW mechanism is applied to allocate a new page with copied content from source page to the VM which performs the access, i.e., to synchronize the memory state in an on-demand fashion. However, as the memory size of VM grows, the copying time of P2M table also increases linearly, which is not scalable to VMs with large memory size. To solve this problem, we have designed and implemented a lazy approach to clone only a small portion of the page tables from the source VM to the cloned one, and track the memory access from both VMs to synchronize the page table entries as well as populate the memory page content. The time consumption of our lazy approach is proved to be sub-second, and remains constant as the memory

size of VM grows.

1.8 Contributions and Outline

To summarize, this dissertation makes the following research contributions:

- Propose *PMSM* mechanism to generalize VM migration to PMs.
- Propose *Introspection-assisted Memory De-duplication* to increase memory utilization using our developed *bootstrapping VMI technique*, which generalizes memory de-duplication to don't-care page content.
- Propose *Introspection-assisted Virtual Machine Migration* to reduce VM migration overhead using bootstrapping VMI technique.
- Propose *Working Set-based Memory Allocation and Compression* based on guest memory reclamation mechanism to increase memory utilization while maintaining application performance, and further balancing the memory of VMs according to their performance overhead.
- Propose *Fast and Light-weight Virtual Machine Cloning* as a sandbox in HAL-based virtualized environment.

The rest of the dissertation is organized as follows. Chapter 2 reviews the previous works on machine state migration, memory de-duplication and compression of VMs, VMI techniques, and HAL-based VM cloning mechanisms, and briefly describes the differences between previous research works and our proposed optimization techniques. Chapter 3 describes the design, implementation, and evaluation of PMSM mechanism. Chapter 4 details the design, implementation, and evaluation of *Introspection-assisted Memory De-duplication* with our bootstrapping VMI techniques. Based on the proposed bootstrapping VMI techniques, Chapter 5 describes the design, implementation, and evaluation of *Introspection-assisted Virtual Machine Migration*. Chapter 6 discusses and evaluates the work of *Working Set-based Memory Allocation and Compression* including TWS-ballooning, dynamic adjustment of zram size, and memory balancing mechanism. Chapter 7 describes the design, implementation, and evaluation of the *Fast and Light-weight Virtual Machine Cloning*. Chapter 8 concludes the dissertation.

Chapter 2

Related Work

2.1 Machine State Migration

Among modern VMSM techniques, *live migration* in Xen virtual machine monitor (VMM) [11] and VMware VMotion [12] have provided mature solutions for migrating virtual machine (VM) instances across different physical machines. Take Xen *live migration* as an example, the Xen VMM takes a snapshot of the VM's state, transfers the state in rounds to the destination host while the original VM is active, stops the original VM when the size of inconsistent VM states is within a threshold, and activates the migrated VM on the destination host. Unlike physical hardware devices used by PMSM, the hardware device of VM is either emulated or abstracted by VMM. During the migration, the device states of VM can be seamlessly saved, transferred, and restored by VMM without physically suspend and resume of the hardware. Thus, VM can quickly detach and re-attach the device without incurring too much overhead. Similar VMSM techniques also appear in past researches such as NomadBIOS [13] built on top of the L4 micro-kernel [14] and Self-migration [15] built atop Xen.

Resembling to our current PMSM design, Kozuch et al. [16] have proposed a new infrastructure for operating systems to lively migrate physical machine states between two different hosts. Their proposed infrastructure is also based on the *hibernation* facility in modern operating systems, but there is no full-fledged implementation yet. In their work, the most challenging issues to enable the physical machine state migration are as the following:

- The target machine must be prepared for state transfer from the source machine,
- Devices must have the ability to be cleanly detached from the source machine and then correctly attached to the destination machine,

- To enable the live migration feature for state transfer, the source machine must obtain a consistent state of the operating system from within it, and
- If hardware devices differ on source and destination machines, the abstraction of device state is necessary so the difference of hardware can be hidden.

In our current PMSM design, we solve their first three proposed issues with Linux *hibernation* facility. Their last issue, the most challenging one, is not completely solved in the current PMSM design in which we assume identical hardware profile on source and destination machines.

As a forked project from the original Linux suspend [17], *TuxOnIce* is of interest because it supports SMP, memory larger than 4 Gbytes (essential for modern servers), and outperforms the standard *swsusp* code in the mainstream Linux kernel code base, due to the use of asynchronous I/O, multithreading, read-ahead, and compression. Thus, we adapt its functionalities as the fundamental parts of PMSM.

2.2 Memory Sharing Mechanism in Xen

To increase the memory utilization efficiency, Xen supports a *memory sharing* mechanism similar to that for sharing memory among processes in a conventional OS, the COW mechanism, i.e., all processes map a shared page as read-only and, upon a write to the shared page from a process Z, a write-protection fault happens and the kernel allocates and maps a new memory page for Z, which is marked as read-writable and initialized with the faulted page's contents. This is achieved by the *unshare* function in Xen.

Xen introduces two new API functions for memory sharing: *nominate* and *share*. The first function accepts a VM identifier and a GFN as input parameters, and marks the corresponding page as read-only with a returned *handle*, which uniquely identifies the page. Accesses to such a *handle* are protected by a global exclusively lock. The second function takes two *handles* as input parameters. If both *handles* are valid, Xen maps the two GFNs associated with these two *handles* to the MFN associated with the first *handle*, and thus frees the physical page corresponding to the MFN associated with the second *handle*. Note that the *share* function does **not** perform byte-by-byte comparison between the two pages, and leaves this task to the developer. Because the comparison for de-duplication must be done atomically inside the *share* function, we have modified the *share* API such that the comparison is performed if it is called due to de-duplication. As an example, assume there

are two VMs, VM_x and VM_y, where GFN_x of VM_x maps to MFN_x and GFN_y of VM_y maps to MFN_y. Xen allows a privileged program, e.g., one running on Dom0[1], to take the following steps to share these two pages: (1) nominating GFN_x and GFN_y with two separately returned *handles*, $handle_x$ and $handle_y$, and (2) sharing these two pages with $handle_x$ and $handle_y$, which maps both GFN_x and GFN_y to MFN_x, and frees MFN_y.

In addition to *nominate* and *share*, Xen also allows the *Dom0* kernel to *map* a GFN of a DomU VM into the virtual address space of a user-level program running on it, and to *translate* a given guest virtual address to its corresponding guest physical address. With these API functions, our memory de-duplication engine is implemented as a user-level daemon running on $Dom0$.

2.3 Memory De-duplication of Virtual Machines

Memory de-duplication for virtualized servers was first introduced in the ESX hypervisor of VMware [18], which identifies common physical memory pages shared by virtual machines running on the same physical machine, keeps one of them, and replaces the rest with pointers to that kept copy, which is marked as read-only. Whenever any virtual machine pointing to the kept copy writes to it, a COW action is triggered, which creates a new copy for the writing VM and allows the write to go through. In concrete, the ESX's *memory de-duplication engine* periodically computes the hash value of every physical memory page, and checks if the resulting hash value is already in the page hash value database. If it is not in, the newly computed hash value is inserted into the database; otherwise, the physical memory page underlying the computed hash value is a possible duplicate, which is then confirmed through a byte-by-byte comparison. The hypervisor reclaims the physical memory page if the confirmation turns out to be positive.

The above approach works independently of the types and versions of the guest OSs (operating systems), and is so simple as to allow the memory de-duplication engine to be implemented inside the hypervisor. However, this approach is not particularly efficient because many of the hash value computation and checking steps could be wasted when the underlying memory pages are not duplicates. Therefore, the key design issue of a memory de-duplication engine is how to discover as many duplicate pages as possible while minimizing the amount of computation efforts required to do so. Thus, our proposed work will utilize our developed bootstrapping VMI technique to eliminate unnecessary hash computation.

[1]Dom0 is the privileged VM in Xen while other VMs are called DomU.

Without directly hashing the page content, Linux's KSM (Kernel Same-Page Sharing) feature [19] uses the page contents as keys to construct a Red Black tree. Each time a new page is encountered, its content is used as a key to search the tree for a possible match. If a match is found, the new page may be de-duplicated. In addition to storing page contents, the Difference Engine [20] applies an additional *page patching* technique to store some pages as deltas with respect to pre-chosen reference pages.

Satori [21] considers the guest OS's page cache as a promising source for duplicated pages. To avoid periodically scanning the memory pages in DomU VMs, it modifies the block device layer of the guest OS (Linux) so that it is *sharing-aware* and enables the Xen hypervisor to intercept all disk accesses. Upon each disk access that populates the page cache, Satori computes the hash values of those new pages that are pushed into the page cache and determine if they are duplicates. In addition, Satori also maintains a list of pages containing replaceable contents, similar to our *don't-care* pages, and provides them to the hypervisor as the new pages used in COW. However, maintaining this list requires modifications to the guest OS, which means that it is not applicable to closed-source OSs.

Besides of *memory de-duplication*, there are other approaches to increasing the memory utilization efficiency. IBM's Collaborative Memory Management (CMM) system [22] modifies the guest OS to provide hints to the underlying IBM z/VM hypervisor, so that the hypervisor can page out memory pages of guests, called *host paging*, and use the reclaimed memory space for other purposes. In addition to guest OS modification, this host paging mechanism also requires stopping the guest whose memory is to be paged out, which incurs noticeable performance overhead.

Transcendent [23] modifies the *page cache* implementation of Linux such that the hypervisor provides a free memory pool, called *precache*, that serves as a second-level cache of the guests' page cache. Every time a guest needs to access the disk, it first queries the *precache* to avoid performing any disk I/O operation whenever possible. On the other hand, when a page is evicted out of a guest's page cache, the page is stored in the *precache* for future accesses. The hypervisor could dynamically shrink and expand this *precache* in an on-demand fashion. Because the *precache* is made visible to the guest OS, significant kernel modifications are required, which again limits its applicability.

Ballooning [24] takes away free memory pages from a guest in a way that the guest OS is aware of the resulting shrinkage in the free memory pool. It uses a *balloon driver* installed inside a guest that utilizes the standard memory allocation API exposed by the guest OS to allocate/free (inflate/deflate) the memory pages of guest. The memory allocated by the balloon driver is typi-

cally put into the hypervisor's free memory pool via hypercalls, which is the interface for guest OS to communicate with hypervisor and similar to system call in conventional OS. The balloon driver receives the balloon target value, and adjusts the guest system memory to the target by inflation or deflation. Hines et al. [25] further applied this technique to decrease the amount of VM state that needs to be transferred and thus the total VM migration time.

However, when a guest OS receives a bulk memory allocation request, its memory allocator may suffer from an *Out-Of-Memory* (OOM) problem [23] because its free memory pool is considered exhausted at that moment, although in theory the guest should still have free memory left. Thus, Hines et al. set aside a certain amount (5% in their paper) of guest memory to prevent the balloon driver from triggering the OOM problem. However, there is no guarantee that this reserved amount could always avoid the OOM problem. In addition, they chose to tune down the invocation frequency of ballooning so as to lower the guest memory pressure. From our experience of developing the TWS-ballooning, it requires extra carefulness to take into account the reserved memory of guest kernel components when adjusting the balloon target, which will be addressed in chapter 6.3. Unlike balloon drivers, which have to be installed in guest VMs, our introspection-based mechanism is completely transparent to and does not require any modifications to the guest OS.

2.4 Virtual Machine Introspection

Virtual machine introspection (VMI) was originally proposed to detect malware intrusion, e.g. kernel rootkits, in guests running on a virtualized server. Pfoh et al. [26] have proposed to leverage the virtualization extension of Intel X86 CPU, e.g. Intel VT-x [1] and AMD SVM [27], to detect in-guest events. For example, in order to intercept context switches inside a guest, the hypervisor could instruct the CPU to trap writes to the CR3 register inside the guests. This hardware-assisted interception makes it very difficult for attackers inside a guest to circumvent. Jiang et al. [28] proposed applying binary translation inside QEMU [29] to intercept all system calls invoked inside a guest so as to analyze attack behaviors in detail. However, these low-level interception methods cannot be applied to the problem of identifying free pages as required by our project.

To narrow the semantic gap between the byte values in a guest's memory pages and the high-level state information they contain, Dolan-Gavitt et al. [30] proposed to log an execution trace of a guest program, extract the instructions from the trace, create a *translated program* from the extracted instructions, and run the resulting program on Dom0 of Xen outside the mon-

itored guest. With guest running on top of QEMU, their guest program is written to use some special instruction to signal the underlying QEMU to start/stop logging of the execution trace. In their work, once the guest program issues CPUID instruction with EBX register set to certain value, the logger in QEMU starts. Also, they use CPUID and other registers as parameters to inform QEMU of the buffer location used by the guest program. The logger then records the current micro-operation, its parameters, and concrete physical memory addresses for each *load* and *store* instruction. Moreover, during the execution of the translated program, all data memory accesses are directed to the physical memory space of the monitored guest using a COW mechanism. This approach leverages the exact instructions used by the guest to interpret the bytes in its memory space, but requires a twin-memory-view execution model that assumes data structures in the guest do not move at run time. This may not always hold in practice because of copy-based garbage collection. Moreover, this approach requires an OS version-specific step that identifies the instruction sequence to be recorded, extracted and translated, and can not be done programmatically.

In order to understand the high-level data structures embedded in a guest's memory pages, Garfinkel et al. [31] have leveraged the *crash* utility [32] to interpret the memory pages of Linux guests and use the results in its intrusion detection engine. The *crash* utility was originally used to debug the Linux kernel, and has several hard-coded parts that are specific to each Linux kernel version. Xenaccess [33] was developed recently to extract process and module information of both Windows and Linux guest OS in the Xen hypervisor. However, it does not provide support for parsing the free memory map, and still contains many hard-coded values specific to each guest OS version.

Bryant et al. [34] also proposed to use introspection to identify free memory pages in Linux guests and avoid copying them during VM state cloning. This research gave another example of memory virtualization optimization that benefits from VM introspection. However, this work did not address the issue of how to programmatically extract the free memory map information from different versions and configurations of Linux kernels. Our work does address this issue and the solution is discussed in Section 4.2.2. With certain domain knowledge, our *bootstrapping VMI* is developed as a systematic method that can extract a certain class of *undocumented* kernel data structures, e.g. the per-page descriptor, from both Windows and Linux guest OSs. Especially for Linux, we leverages the availability of kernel source code to automate the entire process of introspection.

2.5 Memory Compression of Virtual Machines

Besides for the memory de-duplication mechanism, memory compression is another technique to increase memory utilization. This feature has appeared in previous researches, Transcendent [23] and Difference Engine [20], as the last resort of increasing memory utilization. They apply memory compression optionally on the de-duplicated memory pages to further save more memory space.

Also, the recent update of VMware ESX server [18] has started to support memory compression but only when the host is running very low of memory. This feature is tied with its *host swapping* mechanism which swaps out the guest VM pages to host swap disk. Different from the guest swapping which marks the guest page table as not-present for COA, the host swapping marks the P2M mapping of the guest VM.[2]. To choose the candidate page for host swapping, the ESX server randomly picks up pages from the guest physical address space of the target VM. If the compression ratio of the page is smaller than 50%, it is compressed and stored in memory; otherwise, it is sent to the host swap disk.

These researches dislike the memory compression mostly due to the insufficiency and application overhead brought from compression and decompression of the memory pages when the compressed pages are accessed. Thus, our proposed *Working Set-based Memory Allocation and Compression* framework is trying to skip the working set of guest OS, and keep the guest application performance while completing the job of increasing memory utilization. To achieve the proposed work, we leverage page reclamation mechanism of guest OS. For implementation purpose, we focus on the Linux guest OS.

2.6 Page Reclamation in Linux OS

For page reclamation, the Linux OS maintains two LRU (Least Recently Used) lists, Active and Inactive, for the following two major types of memory.

- Anonymous Memory: The memory page used by the heap and stack of user processes.
- Page Cache: The memory pages backed by disk data where the content is cached in memory after the first access to the disk data to reduce future disk I/O.

[2]In practice, the VMware ESX server reclaims VM memory pages by sequentially activating memory de-duplication, ballooning, memory compression, and host swapping mechanisms when the system free memory is below 6%, 4%, 2%, and 1% respectively.

The page in the Active list are considered to be accessed more frequently, referred to as *hot* page, while the page in the Inactive list are considered to be accessed less frequently, referred to as *cold* page. Upon allocation, each page is put into the Active list by default.

The page reclamation can be triggered directly when the kernel fails to allocate memory, e.g., a user process requests a memory page but the kernel fails to find one from its free pool, or indirectly when a specific kernel thread `kswapd` is scheduled due to low system free memory. The kernel reclaims the memory from the Inactive list containing the memory pages which are considered to be relatively cold so that the reclaimed memory will not be soon accessed in the near future. When the number of pages in the Inactive list is not sufficient to fulfill the memory allocation request, the kernel traverses the Active list and moves the cold pages from Active to Inactive lists, i.e., balancing the two lists, until the request is fulfilled. The way to judge whether a page is hot or cold is to check and clear the hardware referenced bit of PTE for the page, which is set by hardware upon an access to the corresponding memory page. Thus, if the referenced bit is turned on while the kernel traverses the Active list, the bit is cleared and the page is considered as hot page and keeps staying in the Active list. Otherwise, the page is considered as cold page, and moved to the Inactive list.

After the two LRU lists are balanced, the page reclamation mechanism continues to iterate the pages in the Inactive list. If the page on the Inactive list is anonymous memory, the kernel swap-out the content to swap disk, mark the corresponding PTE of the process to be not-present, and then free the corresponding page. Later on, if the page is accessed again by the owner process, COA mechanism is performed by bringing the page content from swap disk into a newly allocated memory page, i.e., swap-in. Or if the page on the Inactive list belongs to page cache, the kernel flushes its content to disk if it has been dirtied, and then the page is freed. Upon the next file access, kernel has to again perform the disk access, referred to as ***refault***, to bring the disk content back to a newly allocated page in the page cache. Whenever the swap-in or refault happens, the corresponding guest process will have performance degraded due to the disk I/O delay. From the viewpoint of page reclamation, we quantify the performance overhead of VMs by the sum of its swapin and refault count, referred to as ***overhead_count***, as the following equation.

$$overhead\_count = swapin\ count + refault\ count \tag{2.1}$$

2.7 Working Set Estimation of Virtual Machines

The VMware ESX server [18] uses sampling based mechanism to predict the working set size of VMs. To perform the sampling, the ESX server randomly chooses a few hundreds memory pages periodically, e.g., the default setting is to choose 100 pages per 60-second for each VM, marks them as not-present to detect the memory access of guest among these sampled pages with COA mechanism. However, this mechanism only gives a rough estimation of the VM working set size, and it can not reflect the working set size exceeding the current allocated memory. This is because the sampling only works on the current allocated memory of each VM but the working set exceeding allocated memory size is hidden from the sampling result, e.g., the currently swapped-in pages. In our proposed work, the exceeded part is reflected by detecting the number of swapin and refault events and adjust the VM memory accordingly.

To predict the working set size more accurately, the following researches trace guest memory access, maintain LRU statistics of guest memory pages, and use the LRU miss ratio curve to predict the guest working set size. The LRU miss ratio curve gives estimation of page miss when given a particular size of allocated memory to guest. Lu et al. [35] have proposed to allocate a small portion of memory to each VM while the rest is managed by hypervisor as an exclusive cache. Thus, the memory access of VMs can be intercepted within the exclusive cache, and the LRU miss ratio curve is derived to measure the working set size. Zhao et al. [36] track the memory access of VMs by changing the user/supervisor privilege bit of guest page table entries to supervisor mode so that all memory access of VM will be trapped because the VM runs in user mode. Similarly, the LRU miss ratio curve is also derived for working set size prediction.

However, in Zhao's method, the LRU miss ratio curve can only cover the current allocated memory to the VMs, but not the working set size beyond the current allocation. This is because they only track the memory that is currently mapped to the guest VM, i.e., appeared in the P2M table, but not the one that is reclaimed by hypervisor. Thus, they further trace the guest OS page table modification to track the swapin event, and include the number of swapin into their working set size estimation. However, they do not consider the working set contributed by page cache, which is important to the workload with heavy disk access. In addition, since the guest OS has already maintained the LRU information of current allocated memory of VMs in its reclamation mechanism, it becomes redundant for hypervisor to trap memory access and maintain another LRU statistics for guest VMs. Thus, our proposed *Working*

Set-based Memory Allocation and Compression directly leverages the guest reclamation information.

2.8 Memory Balancing of Virtual Machines

The VMware ESX server [18] samples the working set size of each VM and then determines the amount of memory allocated to each VM proportional to the amount of its shared pages, e.g., due to memory de-duplication, and working set size[3]. However, if we assign memory to VM proportional to the working set size, the VM with larger working set size has to suffer more on performance because the hypervisor reclaims more memory from it and its overhead_count will be proportional to the working set size of the VM. Considering that the swapin plus refault events are the major overhead of VMs, our proposed memory balancing mechanism tries to equalize the overhead resulting from the swapin and refault events. so that each VM has the same performance degradation. As for Lu et al. [35] and Zhao et al. [36], they both try to balance the VM memory allocation with the goal to minimize the global page miss rate among all VMs, i.e., the the sum of overhead to guest VMs. However, optimizing the global page miss of VMs does not prevent any VM from starving on memory resource because some VM may still be inferior during the memory allocation with global optimization. Different from their goal of memory balancing, we aim to equalize the swapin and refault overhead for each VM.

2.9 HAL-based Virtual Machine Cloning

The VMware [18] has supported VM cloning in their latest product, but it only performs snapshot on the disk content without concerning about the current state of CPU, memory, and the I/O device model[4]. The Potemkin framework [8] developed by Vrable et al. is aimed to facilitate the honeypot creation in large-scale network by using VM cloning technique. They use COW mechanism on the PTEs between source and cloned VMs, which is the same as memory sharing. However, they do not consider about the scalability when the page table grows up as the VM memory size grows. In addition, the state of source VM is static and acts as a template for all cloned VMs, which is less flexible than our mechanism to lively clone the source VM state. Also, their

[3]These two numbers are not added directly, but using certain formula in their paper.

[4]The snapshot of disk content is different from the state of I/O device model which contains hardware registers of I/O devices.

system has limited support for I/O devices, e.g, the hard disk is not supported but only ramdisk drive.

Andres et al. [37] have designed and implemented a rapid VM cloning mechanism in Xen hypervisor to enable fast and efficient job deployment in the cloud environment. The memory state of VM is synchronized from source to cloned VM upon any memory access from the VMs. In order to track the memory access from either source or cloned VMs, they traverse and mark all page table entries of source VM as write-protected for COW mechanism while the page table of cloned VM is kept empty and will grow up using COA mechanism. In addition, they have modified the guest OS, i.e., Linux, extensively to (1) provide APIs for guest application to fork VMs to deploy new jobs, and (2) intercept new memory allocation in guest OS to skip the synchronization overhead between source and cloned VM for the newly allocated memory page because the content of this page is a *don't-care* and there is no need to refer to the original content of source VM. However, these modifications restricts their guest OS only to Linux but not to propriety guest OS such as Microsoft Windows.

Sun et al. [38] have done similar implementation on top of Xen hypervisor with focus on unmodified guest OSs. Similar to our discussion in Chapter 1.7, they clone the memory state of VM by copying the P2M table from the source VM to the cloned one, and setting all memory pages as read-only for COW mechanism. From their experiment report, the cloning mechanism is not scalable when the memory size of VM grows because the time to walk through all page table entries during cloning increases linearly with the memory size of VM. In our paper, we have proposed to walk only a fixed portion of page table entries, i.e., the top level of the EPT page table used by hypervisor, and mark the table entries as *not-present* for COA mechanism. Upon any memory access, we lazily propagate the changes from top to low levels of EPT page tables to perform synchronization between source and clone VMs. The details will be addressed more clearly in chapter 7.

Chapter 3

Physical Machine State Migration

3.1 Overview

In modern data centers, Virtual Machine State Migration (VMSM) [6] becomes an essential management primitive because it minimizes the service disruption time in resource consolidation. However, virtualization has not yet been fully embraced [39, 40] for two reasons. First, virtualization cannot achieve the optimal performance because it does not fully leverage the capability of physical hardware. Second, virtualization leads to reliability issues because a single point failure of a physical machine (PM) fails all VMs on it. In contrast to VMSM, PMSM offers the same management flexibility without two issues above. There are four main use cases that motivate the development of PMSM. First, the technology underlying PMSM enables virtual machine migration across virtualized servers that support direct I/O access, which enables a VM to enjoy native I/O performance safely and securely. Shadow Driver [41] was proposed to achieve exactly the same goal. Second, PMSM can complement VMSM by enabling physical-to-virtual (P2V) and virtual-to-physical (V2P) migration. With the help of Shadow driver, P2V and V2P migration are feasible and has the potential to combine the strength of VMSM and PMSM. Third, PMSM can migrate those applications that run on non-virtualized servers. Such applications include I/O-intensive applications that choose to run in native mode directly, such as data acquisition or DBMS, and those applications that are deployed on low power CPU-based servers (e.g. Atom-based or ARM-based) that do not provide hardware virtualization support. Fourth, PMSM-like technology can enable the hypervisor to move all VMs on a PM to another PM in one shot.

Compared with VMSM, PMSM is even more challenging because a VM's state is captured and moved by a separate software entity, typically the hypervisor, whereas in PMSM, the software entity that moves a PM's state is part of the state being moved. Much of the complexity of a PMSM design lies in how to cleanly separate the state associated with the state-capturing software from the original machine state that it is being captured. Another technical challenge in PMSM is how to put the hardware devices in proper states so that they can be properly resumed in a *device-independent* way. This problem is less an issue in VMSM because the hypervisor abstracts away many device-specific details to isolate virtual machines from one another and consequently simplify virtual machine migration.

The goal of this proposed work is to adapt Linux's hibernation technology [42] to support PMSM, which takes a snapshot of a physical machine's state, moves the snapshot to another machine and guides the second machine to resume execution from the captured snapshot. This chapter describes three design variants for a Linux-based PMSM prototype: *swap disk-based*, *memory-to-memory*, and *iterative memory-to-memory migration* where the source and destination machines have the same hardware profile. We implemented all three variants of PMSM, which, to the best of our knowledge, is the first working PMSM implementation that has ever been published in the open literature. The rest of this chapter is organized as follows. Section 3.2 describes in detail the design of the 3 PMSM variants. Section 3.3 presents a performance evaluation study on the PMSM prototype. Section 3.4 gives this chapter a summary of research results.

3.2 Physical Machine State Migration

As shown in Figure 3.1, PMSM migrates the machine state of the source machine M1, which runs the *migrating kernel*, to the target machine M2, which starts from its own *boot kernel*. The machine state includes CPU, memory and network states (e.g., TCP connections), but not states of I/O devices. After the state is transferred, M2 resumes from M1's suspended state. To adapt *TuxOnIce*, a disk-based suspend and resume tool, to support PMSM, we need to address three challenges:

- How to set up the back-end storage environment so that the target machine can access all the files accessible from the source machine?
- How to transfer the snapshot of the source machine to the target machine directly?

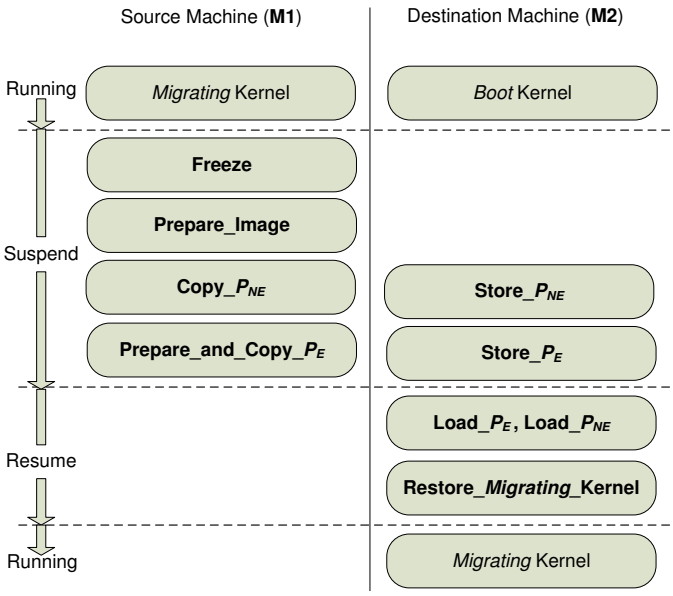

Figure 3.1: *Timeline of a suspend/resume operation in PMSM.*

- How to overlap the suspend operation on the source machine and the resume operation on the target machine as much as possible?

In the following, we describe three network-based migration schemes in terms of how they address these three issues.

3.2.1 Swap Disk-Based Migration

The simplest form of network-based migration, *Swap Disk-Based Migration* (SDM), follows the same stages used in *TuxOnIce*, but with two differences. In this section, we describe in details how swap disk-based migration works and these differences. Because all our three migration schemes are based on *TuxOnIce*, it is necessary to elaborate the details of *TuxOnIce*'s design to better understand the three migration schemes we proposed.

The most challenging part of a suspend operation in *SDM* is that the system needs to be brought to a quiescent state, i.e., no process and device activities or even hardware interrupt, before taking a snapshot of the current machine state and saving it to the swap disk for subsequent restore at the resume time. To ensure the consistency of machine states, *SDM* performs the following procedure, referred to as *atomic-copy*: allocates as many memory pages (referred to as P_{atomic}) as possible so that they can be used to hold the snapshot, stops all device activities, disables hardware interrupts, takes a snapshot of the machine image by using a crafted piece of assembly code, and

resumes the devices with hardware interrupts enabled. A similar mechanism, referred to as *atomic-restore*, is used at resume time.

During the atomic copy/restore step, some pages stay unchanged because they are related to frozen processes and file systems. We refer to these pages as *non-essential pages* (P_{NE}), and the rest of used memory pages as *essential pages* (P_E). *SDM* first saves P_{NE} (unchanged by *atomic-copy*) and then P_E during the suspend operation, whereas the order is reversed during the resume operation.

Suspend

As shown in Figure 3.1, there are four high-level stages involved in *SDM*'s suspend operation on M1:

a) *Freeze*: *SDM* freezes all user and kernel processes at points where they do not attempt to perform I/Os.

b) *Prepare_Image*: *SDM* frees as many memory pages from the page cache as possible to hold the snapshot to be taken, and build up a list of P_{NE} pages, i.e., memory pages including page cache and memory used by user/kernel processes.

c) *Copy_*P_{NE}: *SDM* saves pages in P_{NE} to M2's swap disk using an in-kernel TCP-based data transport utility. These P_{NE} pages can be simply saved before the subsequent *atomic-copy* operation because they will not be changed after all processes are frozen. In contrast, *TuxOnIce* saves all memory pages, including P_{NE} and P_E, to the local swap disk, which is first major difference between *SDM* and *TuxOnIce*.

d) *Prepare_and_Copy_*P_E: *SDM* calculates and atomically makes a copy of P_E to another memory region, P_{atomic}. To move P_{atomic} to M2's swap disk, *SDM* powers on I/O devices, restarts all device drivers required for normal I/O, and then transfers P_{atomic} to M2's swap disk through the in-kernel TCP-based data transport utility. The corresponding stage on M2 is *Store_*P_E, which is the receiver of the TCP-based data transport utility. Finally, all drivers on M1 are called to power down all devices, and then the machine is halted.

Resume

The resume operation consists of three steps:

a) *Start_Boot_Kernel*: When M2 boots with a network-based migration option, its *boot kernel* blocks in a listening call of TCP to receive M1's suspended snapshot image over the network.

b) *Load_P_E, Load_P_{NE}*: *SDM* atomically restores the P_E to its original location at the source machine. Afterwards, the kernel in control on M2 is the *migrating kernel*, or the kernel that was running on M1 immediately before the suspend operation took place. To continue, this kernel calls all drivers to power on devices, calls drivers to restart I/O operations on all devices. Then, pages in P_{NE} are loaded and copied into their original location as in the source machine.

c) *Restore_Migrating_Kernel*: Finally the *migrating kernel* thaws all file systems and all frozen kernel/user level processes, and eventually the entire system resumes. In addition to M1's suspended machine state, to successfully resume M2 also needs to have access to all the storage accessible from M1. For this, M1 and M2 share the same *root* partition on an iSCSI storage server [5]. Coping with shared network storage is the second major difference between *SDM* and *TuxOnIce*.

3.2.2 Memory-to-Memory Migration

Copying M1's suspended machine state to M2's local disk and then restoring it to M2's memory is slow because it involves disk accesses. In contrast, the proposed memory-to-memory version of network-based migration attempts to directly transfer M1's suspended machine state to M2's memory while M2 is under the control of the boot kernel. The main technical difficulty of this approach is that the set of physical pages (e.g., physical pages 1-5000) on M1 used to hold M1's suspended state may be in use by the boot kernel on M2. Therefore, it is not always possible to send M1's suspended state directly to their corresponding physical pages on M2. If the physical page of a M1's state page is being used, this M1's state page is first transferred to a temporary area and then copied back to its corresponding physical page during the network-based resume operation. Then the memory-to-memory migration scheme is derived from *SDM* as follows:

1. In the beginning of *Copy_P_{NE}*, PMSM first determines the set of physical pages on M1 that holds P_{NE} pages, and sends their physical frame numbers to M2.

2. The PMSM component of M2's boot kernel first allocates as much virtual memory as possible, and identifies the set of physical pages actually

allocated, which is denoted as P_{alloc}. The intersection of the physical frame numbers of the memory pages holding P_{NE} in M1 with those associated with P_{alloc} are called *non-conflicting* pages. Those in P_{NE} that are not in the intersection are called *conflicting* pages.

3. PMSM copies the non-conflicting pages in P_{NE} directly to their corresponding physical page frames in M2, and keeps them untouched in the *boot kernel* so that they can safely reside in memory *before and after* the migrating kernel is re-instantiated on M2. Then PMSM copies the conflicting pages to a temporary memory region in M2, denoted as P_{temp}. PMSM also copies P_E pages from M1 to P_{temp} on M2, because it is very likely that P_E pages conflict with pages used by M2's boot kernel.

4. PMSM switches the kernel on M2 from the boot kernel to the migrating kernel, and copies the conflicting pages in P_{NE} and all pages in P_E from P_{temp} to their corresponding physical frames as they are on M1.

3.2.3 Iterative Memory-to-Memory Migration

In the previous two schemes, user-level processes and the file system on M1 are frozen in the beginning of the suspend step, and therefore the perceived service disruption time tends to be long because applications stop functioning during the entire period in which P_{NE} pages are transferred. The *iterative memory-to-memory migration* version of PMSM addresses this deficiency by deferring the freezing of user-level process and the file system until the last possible moment, that is, immediately before saving P_E pages and the CPU state. As a result, this scheme greatly minimizes the perceived service disruption time. As Figure 3.2 shows, we applied similar iterative mechanism used in the *Pre-Copy* phase of VMSM to P_{NE} pages.

The major difference is that we use different dirty page detection technique on page cache pages. In *Pre-Copy* phase of VMSM, dirty pages are detected by write-protecting all pages with the support of hypervisor where a list of dirty pages are built when write-protection violation happens. But in our case, we do not have a hypervisor. The page cache is a part of the kernel address space, and therefore leveraging page write-protection techniques requires modification to the kernel page table and thus complicates the implementation. Thus, we chose to implement a simpler approach. At the beginning of each iteration, PMSM computes the 16-byte md4 checksum of each page in the page cache. At the end of each iteration, for each page, PMSM compares its checksum with the stored one from the previous round. Any mismatch means a changed page in the page cache. In fact, the checksum is also applied to all memory

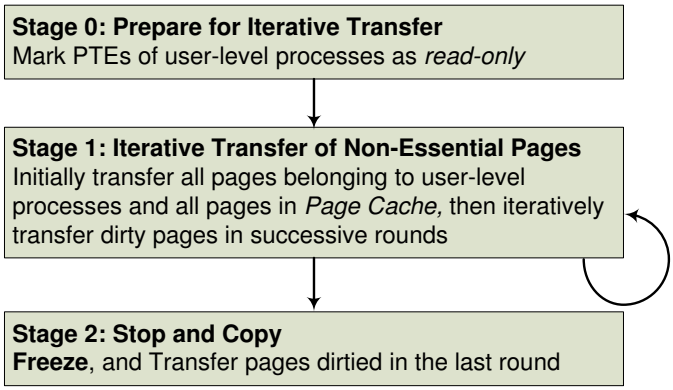

Figure 3.2: *Timeline of iterative state copy in the* iterative memory-to-memory migration *version of PMSM.*

pages exclusively used by the kernel to detect their dirtiness. Although the checksum approach incurs a higher overhead, it is much simpler to implement and greatly speeds up the prototype implementation.

3.3 Performance Evaluation

3.3.1 Methodology

In this section, we first evaluate the correctness of PMSM using the following three standard benchmarks, the AB workload generator [43], which is designed to assess a web server such as Apache, the SPEC SFS benchmark [44], which is designed to evaluate an NFS server, and the TPC-C benchmark [45], which is designed to assess a DBMS server such as MySQL. We configured the AB workload generator so that it generated a total of 500000 HTTP requests over 50 concurrent HTTP connections. The AB workload generator resides on a client machine while the Apache server under test is on a different server machine. We configured the SPEC SFS benchmark so that the input rate of the NFS requests is 100 NFS operations/second. Again, the SPEC SFS benchmark runs on a client machine while the NFS server under test is on a separate server machine. The TPC-C benchmark is configured to run 10 concurrent ODBC (*Open DataBase Connectivity*) connections with 20 warehouses in total. The TPC-C load generator resides on a client machine while the MySQL server under test sits on a different machine configured with 160 MB memory to stress-test disks. For all three workloads, we used PMSM to migrate the state of the server machine to another physical machine while continuing to run the test benchmark on the client machine.

Configuration / Elapsed Time (unit:msec) / Disruption Stage	Apache+AB			NFS+SPEC SFS			MySQL+TPC-C		
	Swap-Disk	**Mem-to-Mem**	**Iterative Mem-to-Mem**	**Swap-Disk**	**Mem-to-Mem**	**Iterative Mem-to-Mem**	**Swap-Disk**	**Mem-to-Mem**	**Iterative Mem-to-Mem**
Freeze	519	475	138	2131	1756	101	667	736	156
Prepare_Image	171	176	-	184	220	-	491	494	-
Copy_P_{NE}	4126	3114	-	5173	3463	-	5454	2676	-
Prepare_and_Copy_P_E	3015	3332	3184	3246	3307	3395	2890	2879	2581
Load_P_E	3022	1960	1966	2997	1964	1936	2538	1962	1988
Load_P_{NE}	5158	-	-	6391	-	-	6717	-	-
Restore_*Migrating*_Kernel	199	1533	1589	37	1463	1469	29	1522	1449
Total	16210	10590	6877	20159	12173	6901	18786	10269	6174

Table 3.1: *Breakdown of the total service disruption time for the three versions of PMSM under the AB benchmark, the SPEC SFS benchmark and the TPC-C test suite, respectively. "-" means the corresponding stage does not contribute to the service disruption time.*

The source server machine (M1) and target server machine (M2) are both Dell PowerEdge SC1425 machines, with one 2.8GHz CPU, 1GB RAM memory, one 250GB SATA hard drive, and two 1Gbps Ethernet NICs, one of which is dedicated to server machine state migration while the other NIC is used to connect the server and the test client.The test client and servers are connected with an 8-port Netgear (Model GS805T) 1Gbit Ethernet switch. Both the M1 and M2 machines use a 80-GB iSCSI disk as the root partition, and this iSCSI disk is hosted on a third server machine. The M1 and M2 machines have the same set of hardware devices, including their local hard drive, on-board NIC and PCIe-based NIC. The client machine runs Fedora 6 with Linux kernel 2.6.22 as the operating system. The M1 and M2 machines run CentOS 5.2 with Linux kernel 2.6.25 as the operating system patched with *TuxOnIce* version 3.0-rc7.

The main performance metric of concern to us is the overall service disruption time, which is defined as the interval between the time when an application process is frozen as part of system suspend on M1 and the time when the application process becomes operational again after system resume on M2. In addition, we are interested in the detailed break-down of the service disruption time among the various stages of suspend and resume. Because application throughputs may suffer during the state migration period, we also compare the throughputs of the three benchmarks over time under the three different implementations of PMSM, i.e., swap disk-based, memory-to-memory and iterative memory-to-memory.

In the fourth step, we evaluate the effectiveness of our two design decisions. These two design decisions include (1) the number of rounds in iterative copy cycles for the *live-migration* implementation PMSM scheme, and (2) the *hit-ratio* in pre-loading P_{NE} pages on M2. As the last step, we show our experience in applying the PMSM schemes over the existing hardware.

3.3.2 Correctness

To prove the correctness of the PMSM implementations, we analyzed the application-level logs generated during the runs of the three test benchmarks, and made a full file system check for the target server machine after each benchmark run. Furthermore, we also ran each test benchmark by turning off server machine migration. That is, for each test benchmark, we have four runs corresponding to the following four configurations: the baseline run without any migration, *swap disk-based* migration, *memory-to-memory* migration, and *iterative memory-to-memory* migration.

By analyzing the logs generated in these four runs for each of three test benchmarks, we found that none of the twelve runs produced any failed re-

Configuration	**No Migration**	**Swap -Disk**	**Mem-to -Mem**	**Iterative Mem-to -Mem**
Successful Requests	500000	500000	500000	500000
Erroneous Requests	0	0	0	0

Table 3.2: *Successful and erroneous requests for the AB workload generator runs under the four configurations: No migration, Swap disk-based migration, Memory-to-memory (mem-to-mem) migration, and Iterative memory-to-memory (mem-to-mem) migration*

Configuration	**No Migration**	**Swap -Disk**	**Mem-to -Mem**	**Iterative Mem-to -Mem**
Successful Requests	29910	30168	30065	29984
Erroneous Requests	0	0	0	0

Table 3.3: *Successful and erroneous requests for the SPEC SFS benchmark runs under the four configurations: No migration, Swap disk-based migration, Memory-to-memory (mem-to-mem) migration, and Iterative memory-to-memory (mem-to-mem) migration*

Configuration	**No Migration**	**Swap -Disk**	**Mem-to -Mem**	**Iterative Mem-to -Mem**
Finished Requests	1833	2099	1561	1972
Late Requests	89	76	76	154
Erroneous Requests	0	0	0	0

Table 3.4: *Finished, late and erroneous requests for the TPC-C benchmark runs with a 20-warehouse database under the four configurations: No migration, Swap disk-based migration, Memory-to-memory migration, and Iterative memory-to-memory migration configuration.*

quest, which is strong evidence that all three versions of PMSM are implemented properly. Table 3.2 shows the numbers of successful and erroneous requests for the AB workload generator runs under the four configurations. None of the four configurations produces any erroneous request. Table 3.3 shows the numbers of successful requests and erroneous requests for the SPEC SFS benchmark runs under the four configurations. The number of successful requests is around 30000 for all four configurations because the benchmark runs for 300 seconds and the input rate is 100 NFS operations/second. The number of erroneous requests is 0 for all four configurations. Table 3.4 shows the numbers of finished requests, late requests and erroneous requests for the TPC-C benchmark runs under four configurations. In each of the four configurations, the number of erroneous request is zero. Late requests for the *iterative memory-to-memory* migration scheme is particularly high because the iterative state copy mechanism lengthens the total migration period, which is defined as the interval between when a physical machine state migration transaction starts and when it ends, even though the service disruption time is reduced.

After each benchmark run, we also performed a full file system check on the target server machine using *Fsck* [46], and found that the file system was internally consistent in all cases, which is another strong evidence that the three PMSM implementations are correct.

3.3.3 Service Disruption Time Breakdown

Table 3.1 shows the breakdown of the service disruption time of the three PMSM implementations under the following three test scenarios, the AB workload generator for the Apache server (denoted as `Apache+AB`), the SPEC SFS benchmark for the NFS server (denoted as `NFS+SPEC SFS`), and the TPC-C test suite with 20 warehouses for the MySQL server (denoted as `MySQL+TPC-C`).

The same as Figure 3.1, the service disruption time is broken down into 7, 6 and 4 stages for swap disk-based migration, memory-to-memory migration, and iterative memory-to-memory migration, respectively in Table 3.1. The memory-to-memory migration scheme does not have the $Load\_P_{NE}$ stage because P_{NE} is directly transferred from M1 into M2's memory before P_E. For iterative memory-to-memory migration, the time spent on the $Prepare\_Image$ and $Copy\_P_{NE}$ stage is excluded from the service disruption time because the application continues to run during these two stages. As a result, the total service disruption time of iterative memory-to-memory migration is about 42.4%, 34.2%, and 32.9% of that of swap disk-based migration under the `Apache+AB`, `NFS+SPEC SFS`, and `MySQL+TPC-C` configuration, respectively, as shown in the last row of Table 3.1.

The most time-consuming part of the $Freeze$ stage is flushing dirty file

system cache pages to disk or the *Sync* operation. For swap disk-based migration and memory-to-memory migration, the file system must be synced before the system can be frozen. Therefore, the sync delay is included in the service disruption time. In contrast, iterative memory-to-memory migration masks most of the sync delay by flushing the file system as much as possible while it transfers P_{NE}. Because the sync delay is a major component of the $Freeze$ time, the $Freeze$ time depends largely on the number of dirty pages in the file system cache when the system is to be frozen. Accordingly, the $Freeze$ time for `NFS+SPEC SFS` is the highest, while the $Freeze$ times for `Apache+AB` and `MySQL+TPC-C` are comparable.

In the $Prepare\_Image$ stage, PMSM allocates pages for P_E and P_{NE}, and copies P_{NE} pages to the allocated area. Because the numbers of P_{NE} pages for `Apache+AB`, `NFS+SPEC SFS` and `MySQL+TPC-C` are 79132, 98337 and 103188, respectively, the elapsed time of this stage is highest for `MySQL+TPC-C`. For the same workload, the elapsed time of this stage is almost the same for all three PMSM variants.

The time consumed by the $Copy\_P_{NE}$ stage is proportional to the number of pages in P_{NE}. For example, in swap disk-based migration, the $Copy\_P_{NE}$ stage takes 4126 msec for `Apache+AB` workload, and 5173 msec for `NFS+SPEC SFS`. The time ratio 0.798 (4126/5173) matches closely with the ratio of pages in P_{NE}, 0.804 (79132/98337). For the same workload configuration, the $Copy\_P_{NE}$ stage in memory-to-memory migration takes less time than that in swap disk-based migration, because the former is bottlenecked by the network bandwidth (121MB/sec), whereas the latter is bottlenecked by the disk bandwidth (76MB/sec).

The time spent on the $Prepare\_and\_Copy\_P_E$ stage is mainly determined by the number of P_E pages. For example, the numbers of P_E pages for `Apache+AB`, `NFS+SPEC SFS` and `MySQL+TPC-C` are 16695, 18954 and 12150, respectively. For the same workload, the elapsed time of this stage is almost the same for all three PMSM variants.

The $Load\_P_E$ stage consists of 3 steps: (1) suspend and power down all peripheral devices, (2) copy pages in P_E to their original physical addresses on the source machine, and (3) power up and resume all peripheral devices. The elapsed time of the $Load\_P_E$ stage for memory-to-memory migration is comparable to that of iterative memory-to-memory migration for all three workload configurations, because steps (1) and (3) dominate the elapsed time, which is about 1699 msec. In contrast, the elapsed time of the $Load\_P_E$ stage in swap disk-based migration is much longer, because Step (2) incurs a disk access overhead to read P_E from disk and thus is the dominant component.

Only swap disk-based migration has the $Load\_P_{NE}$ stage, which involves

loading P_{NE} pages from disk. Therefore, the elapsed time of this stage depends on the size of P_{NE}. For example, the elapsed time of the *Load_*P_{NE} stage is 5158 msec for `Apache+AB`, and is 6391 msec for , `NFS+SPEC SFS`. The ratio of their elapsed times, 0.807 (5158/6391), matches closely with the ratio of their P_{NE} sizes, 0.804 (79132/98337).

In the *Restore_Migrating_Kernel* stage, the file system is first thawed, then kernel threads and user-level processes are thawed. Thawing a file system requires the underlying storage device to reach the ready state. In the test set-up, the file system is based on a disk partition backed up by an iSCSI session, the file system can only be thawed after the Ethernet NIC is ready and the iSCSI session is re-activated. In both memory-to-memory migration and iterative memory-to-memory migration, it takes at least 1600 msec for the Ethernet NIC to enter the ready state after the migration kernel takes control. However, this NIC resume delay is masked by the *Load_*P_{NE} delay in swap disk-based migration. Therefore, the elapsed time for the *Restore_Migrating_Kernel* stage is much smaller for swap disk-based migration than for the other two PMSM variants.

3.3.4 Throughput Degradation

Although service disruption time is a useful metric to compare machine state migration schemes, equally important is the total performance loss during the entire migration period. Contrasting memory-to-memory migration and iterative memory-to-memory migration, the latter reduces the service disruption time by prolonging the running of application programs as much as possible, even though these application programs may run in a degraded mode because the state migration takes place concurrently. So an interesting question is whether iterative memory-to-memory migration decreases the total service disruption time at the expense of end-to-end application throughput degradation. To answer this question, we measured the application throughputs for the three configurations running under the three PMSM variants over a period of time. In the figures reported below, the application's service stops at T=30 sec, and we show the application throughputs until T=90 sec.

As shown in Figure 3.3, in iterative memory-to-memory migration, before the Apache server stops its service at T=30 sec, its throughput already starts dropping at around T=21 sec, from 1190 HTTP requests per second to 450 HTTP requests per second. Two factors contribute to this throughput degradation. First, iterative memory-to-memory migration incurs additional CPU overhead for detecting dirty pages in in P_{NE}. Second, the performance of the `Apache+AB` configuration is bottlenecked at CPU as the CPU usage on the server machine is 100% throughout the experiment. Therefore, the additional

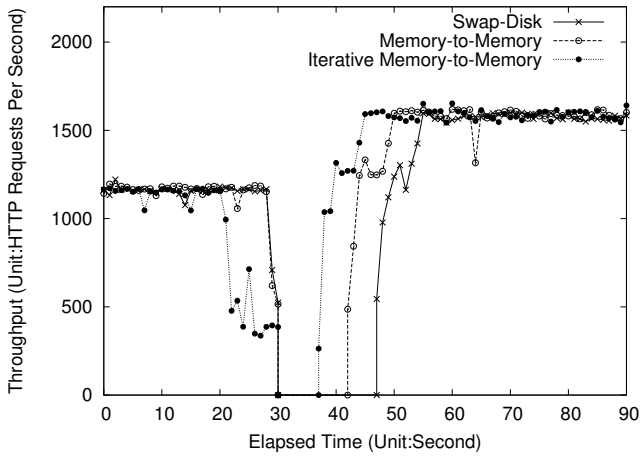

Figure 3.3: *Throughput of the* `Apache+AB` *configuration over time under three PMSM variants.*

CPU overhead incurred by tracking dirty P_{NE} pages noticeably pulls down the throughput of `Apache+AB`.

After the service of the Apache server resumes, it takes roughly the same amount of time, 6 seconds as shown in Figure 3.3, to return to the full throughput (1550 HTTP requests per second), for all three PMSM variants. For example, for iterative memory-to-memory migration (swap disk-based migration), the Apache server resumes service at T=37sec (T=47sec), and runs at full speed at T=43sec (T=53sec). The main factor determining the rate at which the full throughput is recovered is the TCP congestion window. Because the growth of the TCP congestion window is not affected by the actual state migration schemes, the time to return Apache to full speed is thus identical for the three PMSM variants.

Astute readers may notice that the maximal throughput of the Apache server is 1190 HTTP requests/sec on M1 and 1550 HTTP requests/sec on M2. We investigated the root of this performance difference, and found out that this performance difference exists even without PMSM, and mainly comes from the fact that the NICs on M1 and M2 perform differently even though they are advertised to have the same capability.

Unlike `Apache+AB`, iterative memory-to-memory migration does not exhibit significant throughput degradation before the NFS server stops service, as shown in Figure 3.4. The main reason is that the `NFS+SPEC SFS` workload configuration is not CPU-bounded, as the CPU utilization of the server machine throughout the experiment does not exceed 5%. As a result, the additional CPU overhead due to tracking dirty P_{NE} pages does not have any

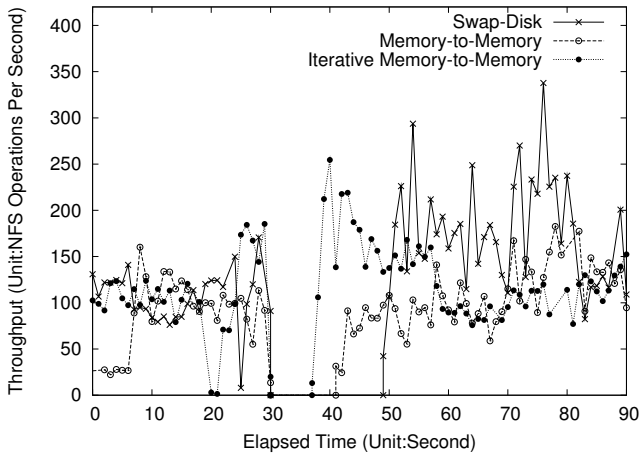

Figure 3.4: *Throughput of the* `NFS+SPEC SFS` *configuration over time under three PMSM variants.*

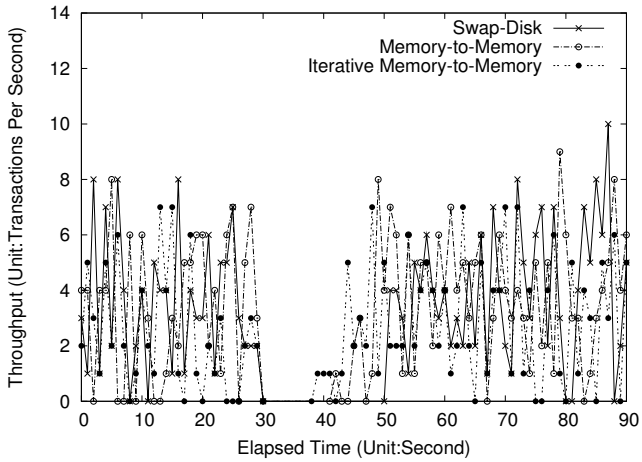

Figure 3.5: *Throughput of the* `MySQL+TPC-C` *configuration over time under three PMSM variants with 20 database warehouses.*

noticeable impact on the performance of `NFS+SPEC SFS`. This means that the reduction in service disruption time enabled by iterative memory-to-memory migration does not come with any hidden performance cost.

Figure 3.5 also shows that there is no noticeable throughput drop for iterative memory-to-memory migration before the MySQL server stop its service at T=30 sec. The reason is the same as in `NFS+SPEC SFS`, i.e., the `MySQL+TPC-C` workload configuration is disk-bounded and therefore the CPU overhead incurred by iterative memory-to-memory migration does not affect the performance of `MySQL+TPC-C` at all.

3.3.5 Design Choices in Iterative Memory-to-Memory Migration

In this subsection, we evaluated the impact of the amount of P_{alloc} memory in M2 on the effectiveness of direct state transfer, and the optimal number of iterations that strikes a good balance between the dirty page tracking overhead and the state transfer overhead.

In the evaluation experiments, iterative memory-to-memory migration uses 960 MB P_{alloc} memory on M2 to hold P_{NE} pages. From our experience, if the P_{alloc} exceeds 960 MB on the server (1GB physical memory), some processes will be killed by kernel because available kernel memory drops to the lowest threshold. We can infer from this phenomenon that the boot kernel occupies around 40 MB of memory pages. Table 3.6 shows the *hit-ratio* of P_{NE} for all three configurations running under iterative memory-to-memory migration. The *hit-ratio* is the percentage of P_{NE} pages that can be directly transferred from their physical memory locations on M1 to the same physical memory locations on M2. Because these pages do not require additional copying, the more they are, the lower the $Load\_P_{NE}$ overhead. Theoretically, if the size of P_{NE} is N, the lower bound of the *hit-ratio* should be $\frac{N-Size(BootKernel)}{N}$, where $Size(BootKernel)$ denotes the size of the residence set of the boot kernel on M2. Table 3.6 shows the calculated lower bounds are indeed smaller than the observed *hit-ratio*s.

Figure 3.5 shows the number of P_{NE} pages that are transferred from M1 to M2 in each iteration round for the three workload configurations when the maximal number of iterations is set to 11. Among all three workload configurations, the number of P_{NE} pages transferred drops to under 30 after **three** iterations. More specifically, the number of pages transferred at the fourth iteration drops to 2, 1, and 29 for `Apache+AB`, `NFS+SPEC SFS`, and `MySQL+TPC-C`, respectively. The main reason is that the rate of pages being dirtied is much slower than the network bandwidth. More concretely, the aver-

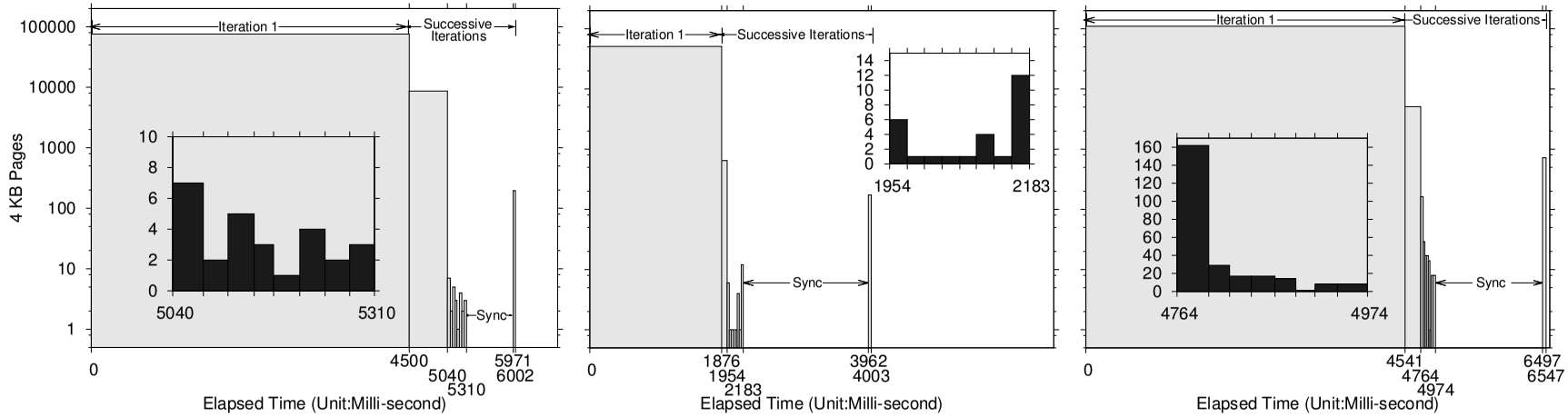

Table 3.5: **(a):** *Number of pages transferred during each memory state transfer iteration for the* `Apache+AB` *workload. The Y axis is in log scale.* **(b):** *Number of pages transferred during each memory state transfer iteration for the* `NFS+SPEC SFS` *workload. The Y axis is in log scale.* **(c):** *Number of pages transferred during each memory state transfer iteration for the* `MySQL+TPC-C` *workload. The Y axis is in log scale. Note that the embedded sub-figures enlarge iteration 3 through iteration 10 in all three figures, and its Y axis is in normal scale.*

Hit-Ratio (Unit: %)	Apache +AB	NFS +SPEC SFS	MySQL +TPC-C
Experimental	93.1	94.8	98.9
Theoretical Lower Bound	87.8	90.6	83.8

Table 3.6: *Hit-ratio of pages in P_{NE} when P_{alloc} on M2 is equal to 960 Mbytes*

age page dirty rates for `Apache+AB`, `NFS+SPEC SFS`, and `MySQL+TPC-C` are 680 KB/s, 480 KB/s, and 3.3 MB/s, respectively, while the raw network bandwidth of the dedicated NIC used is 121 MB/s.

The time gap between the last iteration and the iteration before it is attributed to file system flushing and is thus determined by the number of dirty page cache pages and data locality among these pages. In our experiments, the numbers of dirty pages observed at the beginning of the last iteration are 124, 3056, and 750 for the `Apache+AB`, `NFS+SPEC SFS` and `MySQL+TPC-C`, respectively. Although the number of dirty page cache pages in `MySQL+TPC-C` is no more than $\frac{1}{4}$ of that in `NFS+SPEC SFS`, the time gap between the last iteration and the iteration before it for `NFS+SPEC SFS` (1.78 seconds) is comparable to that of `MySQL+TPC-C` (1.52 seconds), because the dirty pages in `MySQL+TPC-C` have poorer data locality.

Figure 3.5 also shows that the net state transfer rate of `Apache+AB` is smaller than those in `NFS+SPEC SFS` and `MySQL+TPC-C`. For example, it takes 4.5 seconds to transfer 310 MB for the `Apache+AB` workload configuration, but it takes 4.5 seconds to transfer 451 MB for the `MySQL+TPC-C` workload configuration. This is because `Apache+AB` is CPU-bounded and thus its state transfer performance is affected more by the CPU overhead due to dirty P_{NE} page tracking.

3.4 Summary

In this chapter, we describe the design, implementation and evaluation of the first known physical machine state migration system (PMSM) that is capable of migrating the state of a running OS from one physical machine to another with the same hardware profile. Such a capability enables applications that are not suited to run on virtualized servers to enjoy the same benefits associated with migration as those that run on virtualized servers. We have tested the PMSM prototype under various workloads to stress-test its correctness. Empirical measurements show that PMSM is relatively fast: It can migrate a NFS

server in 6.9 seconds and a MySQL server in 6.17 seconds. More concretely, this work makes the following three research contributions:

- Development of three real-time physical machine state migration algorithms for the Linux kernel,
- First known successful implementations of these PMSM algorithms for homogeneous source and destination machines that have withstood tests using real-world server applications, and
- An empirical performance study of these implementations and a detailed analysis of their overheads and the effectiveness of various optimizations.

Chapter 4

Introsepction-assisted Memory De-duplication

4.1 Overview

As chapter 2 describes, most if not all existing memory de-duplication schemes are based on memory page contents, and as a result they cannot de-duplicate memory pages whose contents are *don't-cares*, e.g. pages in the free memory pool of the guest OS or application processes. The contents of these pages are immaterial, and could have been treated as all-zero without affecting the system's correctness. Therefore, if the memory de-duplication engine can successfully identify these pages, it can de-duplicate them with a single physical zero page. In other words, we are proposing a *generalized* memory de-duplication engine that performs duplicate check using the *contents* and *type* of physical memory pages. This engine not only widens the scope of memory de-duplication, but also decreases the average amount of per-page de-duplication computation.

To identify the free memory pools in guest OSs that run Microsoft Windows and Linux, we applied virtual machine introspection to examining the relevant kernel data structures in them. We have observed that a global *memory map* exists in both OSs, which contains an array of memory descriptors representing the state of each physical memory page in the guest VM. For example, the descriptor of every free page[1] of a Windows XP OS's free memory pool has its *type* field set to zero [55].

Although the base location of the free memory map is available from the debug symbol table for the guest OS and can be manually retrieved, as is

[1]In Windows, free pages are stored in a *zeroed page list*, whereas in Linux, they are stored in a *free page list*.

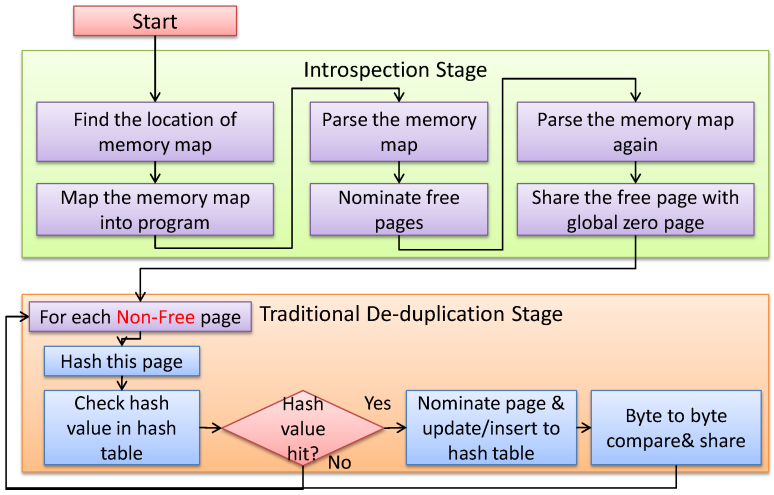

Figure 4.1: *The workflow of the proposed Generalized Memory De-duplication (GMD) engine, which comprises two stages: the* Introspection *stage to identify free memory page in guest VMs and the* De-duplication *stage that de-duplicates pages using hashing and byte-by-byte comparison.*

done in Xenaccess [31, 33], the internal structures of the free memory map are undocumented and tended to vary from one kernel version to another, for example, the per-page descriptor size has been changed from 24 bytes in the 2006 version of Windows XP to 28 bytes in the 2009 version of Windows XP. A major contribution of this work is a *bootstrapping* VM introspection technique that can programmatically collect and assimilate all related information for both Windows and Linux guest OSs.

The rest of this chapter is organized as follows. In Section 4.2, we present the design of the proposed Generalized Memory De-duplication (GMD) engine, and detail the bootstrapping VM introspection technique for examining kernel data structures across multiple versions of both Windows and Linux. In Section 4.3, we demonstrate the effectiveness of the bootstrapping VM introspection technique for multiple versions of Windows and Linux guest OSs, and present the performance gain of GMD over vanilla memory de-duplication. In Section 4.4, we summarize the main research contributions of this chapter.

4.2 Generalized Memory De-duplication

Most existing memory de-duplication systems are based solely on the *contents* of memory pages. In contrast, the proposed Generalized Memory De-duplication (GMD) engine identifies pages that can be de-duplicated based on

their *type*. Specifically, GMD identifies free pages in guest OSs, treats them as duplicates of an all-zero page, and de-duplicates them accordingly. The rationale behind this approach is the contents of free memory pages are "*don't care*" and thus could logically be considered as all zero for de-duplication purpose. That is, if a guest OS has N free memory pages, they can all be de-duplicated with an all-zero page, and returned to the hypervisor.

The current GMD engine prototype is implemented as a user-level program that runs in *Dom0* and implements de-duplication using the Xen hypervisor's primitives described in Section 2.2. As shown in Figure 4.1, the GMD engine consists of two stages: the *Introspection* stage and the *De-duplication* stage. In the *Introspection* stage, the GMD engine

1. Maps the *memory map* of a guest VM x into its own address space,
2. Walks through each descriptor of the memory map, marks those pages that are free in a bitmap, $free\_map1$, and nominate the free pages,
3. Walks through the memory map again to check if those free pages identified in Step 2 are still free, and marks those that are still free in another bitmap, $free\_map2$, and
4. Shares each page is that is in both $free\_map1$ and $free\_map2$ with the all-zero page without comparison.

In the *De-duplication* stage, for each non-free page P, the GMD engine computes a hash for it, looks up the resulting hash value in a global page content hash database, nominates P if a hit is found, shares P with the hit page. If the byte-by-byte comparison comes back with the match, the *share* function removes duplicate page and returns success.

The GMD engine checks *twice* if a memory page in a guest OS is free during the Introspection stage, because the GMD is implemented as a user-level program running in Dom0 and there is a potential race condition between when it detects a guest memory page is free and when it nominates the page for sharing. As shown in Figure 4.2 (a), if a guest page that is identified as a free page is allocated and modified after it is nominated, the modification triggers the COW mechanism, which in turn destroys the page's *handle* returned by the *nominate* call, and the following *share* call will abort because the *handle* is invalid. If, however, the modification appears after the first free page check but before the *nominate* call, as shown in Figure 4.2 (b), the page's *handle* continues to be valid when the GMD engine nominates and shares the page with the all-zero page, and eventually the modification is lost. A usual solution to such a problem is to ensure atomicity through locking. However, this is infeasible because such locks are not available to the GMD engine.

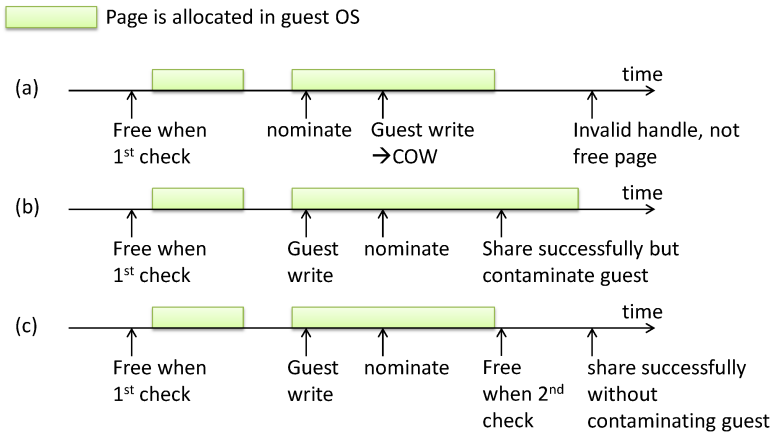

Figure 4.2: *The GMD engine checks if a guest page is free twice to avoid a race condition illustrated in (b), where a page is detected free, modified by the guest, and then nominated and shared by the GMD engine. Checking pages are free twice avoids the data corruption problem due to this race condition, as shown in (c).*

Instead, we solve this problem by checking free memory pages twice. With a second free page check shown in Figure 4.2 (c), a guest page can only be nominated and shared only if the second check also reports it is free. If a guest page is still free in the second check, it means the page could be safely de-duplicated with the all-zero page because its contents can be thrown away.

A key assumption of the GMD engine is the ability to introspect a guest OS and derive the set of memory pages that are free and thus can be de-duplicated. The following two subsections detail the exact mechanisms to accomplish this for Windows and Linux virtual machines.

4.2.1 Introspecting Windows Image to Identify Free Pages

On Windows, executable files, dynamically linked libraries (DLLs), and kernel images are all in the Portable Executable or PE format [56]. When a PE file is loaded into memory, it is stored at a starting address known as the *base address*. The virtual address of every symbol in a PE file is represented as an offset with respect to the file's base address, also known as the Relative Virtual Address (RVA). Therefore, a symbol's absolute virtual address is equal to the sum of its RVA and the *base address* of where the associated PE file is loaded. As shown on the right side of Figure 4.3, when a PE file is loaded, its headers are also loaded. In contrast, an executable file's header is not loaded under Linux. The two items in the PE header that are relevant here are *export data*

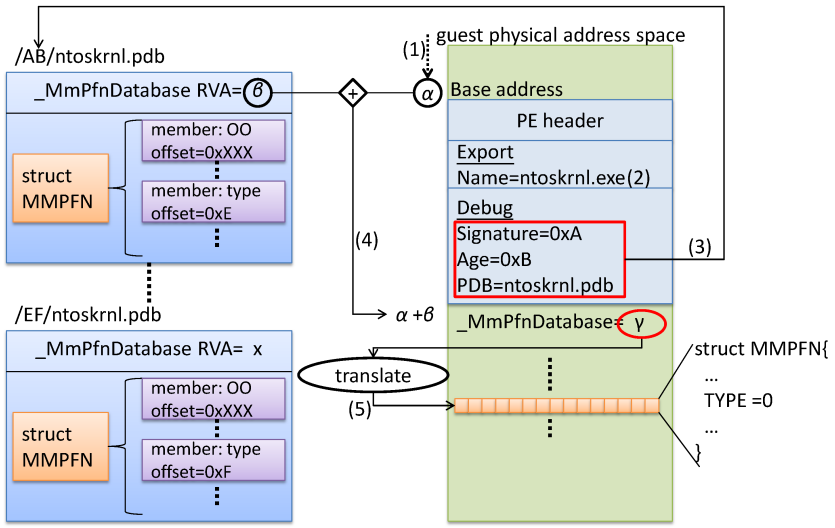

Figure 4.3: *Introspecting a Windows Virtual Machine requires matching a Windows kernel PE file loaded into a guest VM (on the right) with its corresponding debugging PDB file fetched from Microsoft's Symbol Server (on the left).*

directory, which contains the name of the current PE file, e.g., `ntoskrnl.exe` for the Windows kernel [55], and *debug data directory*, which contains a 128-bit *Signature* and a 32-bit *Age*, which together uniquely identify the PE file's associated debugging information.

For each PE file, the linker, e.g., Microsoft Visual C++, may optionally produce a PDB file that contains the associated debug information, such as the symbol table. For widely used PE files, Microsoft even publishes their PDB (Program Database) files on its public web site. The PDB file by default contains the name, address, and type information for functions and variables, e.g., the RVA of a static variable declared in the program. To associate a PE file with its PDB file, both of them include a unique combination of *Signature* and *Age* values. In addition, Microsoft provides a Debug Interface Access (DIA) SDK [57] that allows developers to extract detailed debug information about a PE file, e.g., one can use a depth-first-search algorithm to query a specific member of a user-defined structure, because the latter is stored as a tree hierarchy inside its PDB file.

The *memory map* in a Windows kernel is called the PFN (Physical Frame Number) Database, which is a statically allocated array of page descriptors, each of type `struct _MMPFN`, and is located in a contiguous region of the guest physical address space [55]. The symbol name of the PFN Database is `_MmPfnDatabase`.

After a Windows OS boots up, all its free pages are in the *Zeroed* page list, which is the free memory pool every page in which is all-zero. After a free page is allocated to a process, it is taken out of the *Zeroed* page list. After an in-use page is returned to the kernel, it is put in the *Standby* list if it is clean or in the *Modified* list if it has been written while it was in use. After a process exits, pages in its working set or in the Standby or Modified list are returned to the *Free* page list. For security reasons, the Windows OS uses a *zero page thread* to zero out the pages in *Free* list and put them into the *Zeroed* page list for future reuse.

The GMD engine considers a guest page is free if it is located in the *Zeroed* list, i.e., when the *type* of the page descriptor is equal to `ZeroedPageList`, which is defined as a member of an enumeration type `_MMLISTS`. This rule applies to all Windows OS versions ranging from Windows XP to the latest Windows 7 [55, 58, 59]. However, because the detailed layout of the enumeration type `_MMLISTS` may vary from kernel version to kernel version, we have to rely on kernel version-specific PDB files to help determine the exact layout of this data structure and use this knowledge to perform free memory page check.

To summarize, the GMD engine takes the following steps to programmatically traverse the PFN database and inspect each page descriptor, as outlined in Figure 4.3.

(1) Because the beginning of the first page of the PE file holding a Windows kernel contains a magic string "MZ", the GMD engine identifies the base address of a Windows guest kernel by scanning the guest VM's physical address space range until the magic string is found. We denote the guest kernel's base address as $Base_{kernel}$.

(2) To confirm the PE file containing the magic string is indeed a Windows kernel, the GMD engine locates the PE file's export data directory, and from it extracts the name of the PE file, which is supposed to be `ntoskrnl.exe` if the found PE file is indeed the kernel image[2].

(3) To retrieve the PDB file associated with the Windows kernel PE file, the GMD engine extracts the *Signature* and *Age* fields from the PE file's debug data directory, and uses them to construct an URL that can be used to access the PE file's PDB file from Microsoft's public symbol server, which provides debug symbol information on Windows OSs for kernel developers. All PDB files on the symbol server are structured as a file system hierarchy. The PDB file's URL begins with the IP address of symbol server,

[2]This step is leveraged from Xenaccess [33].

followed by the PDB file's name, then a concatenation of the corresponding *Signature* and *Age* values, and finally the PDB file's name with the last character replaced by an underscore. However, this symbol server is meant to be used by Windows debugging tools rather than web browsers. We have to manipulate the *User Agent* field in the HTTP requests sent to the symbol server [60] to retrieve the PDB files. In particular, we used the *curl* tool [61] under Linux as follows to convince the symbol server to work with our code:

```
curl --user-agent "Microsoft-Symbol-Server"\
 http://msdl.microsoft.com/download/symbols\
 /ntoskrnl.pdb/AB/ntoskrnl.pd_
```

After the kernel image's PDB file is downloaded, we used the DIA SDK to search for the RVA of the symbol `_MmPfnDatabase`, which is denoted as $Base_{PFN}$, and to traverse the `_MMPFN` structure to derive the data structure's size and the offset of its member *type*.

(4) The GMD engine computes the start physical address of the PFN database array by adding $Base_{PFN}$ to $Base_{kernel}$. Although $Base_{PFN}$ is a guest virtual address and $Base_{kernel}$ is a guest physical address, it is semantically correct to add them together because the kernel image resides in a contiguous region of the guest physical address space.

(5) The GMD engine translates the guest physical address of the PFN array's base to its corresponding machine physical address, maps the array into its own virtual address space, and examines the *type* field of every PFN database entry to determine it is free using the layout information extracted from the `_MMPFN` structure.

4.2.2 Introspecting Linux Image to Identify Free Pages

When a Linux kernel image is built, a special file called *System.map* is also generated, which contains a mapping between the symbol names of all exported variables and functions in the kernel source and their absolute virtual addresses. A Linux kernel image (*vmlinuz*) is in the ELF (Executable and Linkable Format) format and comprises two parts: (1) the real-mode kernel image and (2) the compressed protected-mode kernel image. The first part is loaded into memory by the disk boot loader and is responsible for retrieving the BIOS system information, loading the second part of *vmlinuz*, and jumping

to the entry point of the protected-model kernel after switching to the protected mode. By default, the Linux kernel image refers to the protected-mode kernel image. The header of the real-mode kernel image contains detailed Linux version information such as the kernel version, distribution, and build-timestamp, which can be used to track down the corresponding distribution source package.

Besides the exported symbols from *System.map*, kernel-level debugging information such as definitions of variables, structures, and functions, could be embedded into the kernel image during kernel compilation, if the kernel configuration option `CONFIG_DEBUG_INFO` is turned on. Using the GDB interface [62], one can extract these debug symbols from the kernel image.

To avoid memory fragmentation, Linux uses a buddy system memory allocator, which organizes free memory pages into groups of physically contiguous pages, each with a size that is a power of two. The first page of every free memory page group is called a *Buddy page*, which represents the entire group and uses the *private* field of its page descriptor to record the number of physically contiguous pages in its group. For example, if the *private* field of a *Buddy page*'s descriptor is n, there are 2^n contiguous free pages in its group. Therefore, one can uncover free pages in a guest OS by first identifying Buddy pages and then all other pages in their groups.

Linux supports two memory models, each of which corresponds to a different representation for its memory map. The exact memory model used in a kernel is selected at the kernel configuration step, and remains fixed after the kernel is built. In the *flat* memory model, the memory map, whose symbol is `mem_map`, is, like Windows, stored as a physically contiguous array of descriptors, one for each memory page. This memory model is mostly used in 32-bit guests that are allocated with a contiguous guest physical address space. The *sparse* memory model is mainly used in 64-bit guests that are given a larger guest physical address space but with holes. In this model, instead of allocating a descriptor for every guest physical page in the guest OS, a two dimensional array, whose symbol is `mem_section`, is allocated to avoid wasting descriptor space on guest physical pages that are in the holes. After a kernel is built, either `mem_map` or `mem_section` appears in *System.map* depending on the memory model configured.

There are many Linux distributions that are based on the same kernel source. Some distributions, e.g., Ubuntu and Centos, add their own patches or drivers into a vanilla (officially unmodified) Linux kernel, and publish the built kernel image as installation packages, e.g., the *deb* file in Ubuntu and *rpm* file in Centos. The kernel images in these packages usually do not come with any debug information. However, we can recreate the debug information associated

with a kernel distribution by downloading the distribution's corresponding kernel source and building a kernel image from it with the `CONFIG_DEBUG_INFO` option enabled.

Introspecting Linux image is qualitatively different from introspecting VM image, because the Linux kernel is highly configurable, and the same kernel source could be compiled into kernel versions with very different structures and configurations. For example, configuration primitives such as `#ifdef` could conditionally trigger different compiler pre-processing and post the following issues that make it difficult to identify free pages by examining the memory map:

(a) The memory model configuration affects how the memory map is represented, a one-dimensional or two-dimensional array, as illustrated by the following code snippet in the Linux kernel source.

```
#ifdef CONFIG_SPARSEMEM_EXTREME
extern struct mem_section *mem_section[NR_SECTION_ROOTS];
#else
extern struct mem_section mem_section[NR_SECTION_ROOTS]\
[SECTIONS_PER_ROOT];
#endif
```

(b) Size and layout of the page descriptor data structure may vary from one kernel version to another.

(c) Semantics associated with data structure field values required to identify Buddy pages vary from one kernel version to another.

The first two issues could be resolved by using the GDB interface to query the debug information associated with the input kernel image. However, the last issue could not be easily resolved because which data structure field is used to identify Buddy pages changes from one kernel version to another. For example, in Linux versions older than 2.6.18, a page is said to be a Buddy page if the 19th bit of the `flags` field in the page's descriptor is set, but in version 2.6.38, it uses another field called `_mapcount`.

Fortunately, across all Linux kernel versions, there is a standard inline function, `PageBuddy(struct page*)`, that takes a page descriptor structure as the input and returns true if the page is a Buddy page. However, the implementation of this function is different for different kernel versions. One way to solve the Buddy page identification problem is to incorporate the logic of the PageBuddy function for different Linux kernel versions into our engine

as the crash utility [32] so as to identify free memory pages in guest OSs that run different versions of Linux. However, this approach is cumbersome and error-prone.

Instead, we directly leverage the `PageBuddy` function's implementation in each Linux kernel version by calling it from the GMD engine. More concretely, we wrote a stub function, called `GFN_is_Buddy`, which determines whether a given GFN in a guest is a Buddy page or not by calling the guest's `PageBuddy` inline function. The input argument of this stub function is a GFN in a guest, and this interface is the same across all Linux kernel versions. If the GFN corresponds to a Buddy page, the function returns the number of free pages starting from this GFN.

```
/* stub.c: guest kernel module */
int GFN_is_Buddy(unsigned long GFN)
{
    struct page *page;
#ifdef CONFIG_FLATMEM
    /* Flat model: use memory map as 1D array */
    page = (struct page*)mem_map[GFN];
    /* page->private shows the power of two for
     * the free pages in the buddy system */
    if (PageBuddy(page))
        return 1 << page->private;
    else
        return 0;
#else ...
}
```

Given a guest VM, we dynamically compiled this stub function against the source code of the specific Linux kernel version and configuration used by the guest, and produced an ELF file, `stub.o`. Then we linked this `stub.o` file with the GMD engine so that the latter can call the `GFN_is_Buddy` function on every guest page descriptor it examines. The above approach represents a brand new VM introspection technique in that it pioneers the use of kernel version-specific code to interpret kernel version-specific undocumented data structures. All this technique needs is the source code and configuration file of the Linux kernel version used by a guest VM, but not any a priori knowledge of the detailed layout or semantics information associated with any kernel data structures.

Eventually, we decided to separate the stub function from the GMD engine into a separate program for the following two reasons. First, because

the PageBuddy functions in different Linux kernel versions may use identical variable, it is difficult to compose a general stub function that can simultaneously call on multiple kernel version-specific PageBuddy functions. Therefore, it is more feasible to prepare a separate stub function for each Linux kernel version. Second, because the stub function is complied against the source code of a particular Linux kernel version, the resulting stub.o file could be either 32-bit or 64-bit, depending on the Linux kernel version in question. Accordingly, it is impossible to run the stub function in the same address space as the GMD engine, which is a 64-bit user-level program. The following code snippet shows the independent free page check program (FreePageCheck.c) that includes the GFN_is_Buddy function and services requests from the GMD engine. The communications between the GMD engine and the free page check program is based on a shared memory mechanism.

```
/*  FreePageCheck.c: kernel version-specific code
     that checks if a page is a Buddy page */
extern int GFN_is_Buddy(unsigned long);
void check_free_page(int *free_map)
{
  int num_of_free_pages;
  for (gfn = 0; gfn < guest_max_gfn;)
    if (num_of_free_pages = GFN_is_Buddy(gfn))
    {
        ... /* mark free_map accordingly */
        gfn += num_of_free_pages;
    }
    else
        gfn++;
}

int main()
{
  check_free_page(free_map1);
  ... /* nominate free pages */
  check_free_page(free_map2);
  ... /* share free pages after double check */
 }
```

Figure 4.4 shows the steps taken to generate a kernel version-specific stub.o and link it with FreePageCheck.c to form the kernel version-specific free page check program that provides service to the GMD engine. With this

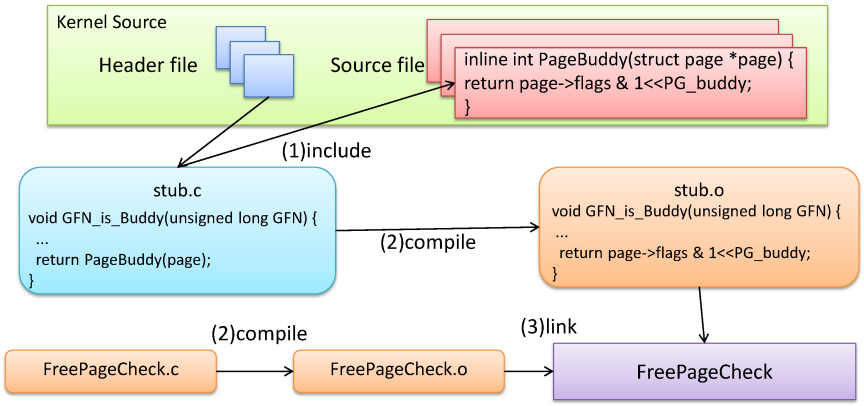

Figure 4.4: *Compiling the* `GFN_is_Buddy` *function against the source code and configuration file of a guest's Linux kernel version to generate a* `stub.o` *file, and linking it with* `FreePageCheck.c` *to form the independent free page check program.*

dynamic code composition mechanism, the memory de-duplication procedure for a Linux guest work as follows:

(1) Searches the real-mode kernel image's header of the Linux guest for the the magic string "HdrS", then extracts the kernel version information, and uses the kernel version information to either retrieve the corresponding free page check program if it exists, or dynamically composes the corresponding free page check program based on the kernel version's source code and configuration file,

(2) Extracts from the `System.map` file associated with the kernel version used in the Linux guest the memory model used and the location of the memory map, and

(3) Traverses every page descriptor in the memory map according to the memory model by calling the free page check program to determine if a page is a Buddy page, and nominates and shares every free memory page using the flow shown in Figure 4.1.

4.2.3 Summary of Bootstrapping VMI Technique

Traditionally, VM introspection relies on manually constructed interpretation programs that convert byte sequences in a guest physical address space into high-level kernel data structures. The process of building these interpretation

programs is time-consuming and error-prone, and sometimes next to impossible, especially for closed-source OS. In the development of the GMD engine, we have developed the following two new techniques that, to a certain extent, automate the construction of the interpretation programs for VM introspection.

1. A technique to programmatically leverage the kernel debugging information, which is either pre-built by the original software publisher or dynamically built on demand from the corresponding kernel source code, to derive the type information of variables and data structures of interest in a guest OS.

2. A technique to programmatically leverage selected functions in a guest OS to derive the semantic meanings (typically un-documented) of specific byte values in the guest physical address space.

The above two techniques still require a priori knowledge of how to link high-level information being sought after (e.g. memory page status) with specific kernel data structures (e.g. memory map), and thus do not remove manual efforts completely. Because a kernel image is a product of the kernel source code, the configuration file, and the compiler, the above two techniques assume that the kernel image used in a guest whose state is being examined is identical to the binary image which these two techniques draw on. In particular, any change to the source code, the configuration file, or the compiler, may potentially render invalid the resulting interpretations.

The second technique is an instance of applying a guest OS's own functions to make sense of its own byte sequences. While conceptually simple, it has two potential pitfalls. First, the guest functions used to interpret the guest physical address space must be self-contained and do not reference other kernel variables or functions that cannot link with a user-level program. Second, as the interpretation program inspects a guest's physical address space, the guest's state must not go through any modification; otherwise the interpretation program may break because of dangling pointers, e.g., a next pointer in a linked list item becomes invalid as a result of guest state modification.

4.3 Evaluation

4.3.1 Methdology and Test Set-up

We used three metrics to evaluate the effectiveness of a memory de-duplication scheme: (1) the number of pages it reclaims, (2) the performance overhead it

itself incurs, and (3) the performance penalty it imposes on guest VMs. The performance overhead of conventional memory de-duplication schemes mainly comes from hash computation and byte-by-byte comparison, whereas that for the proposed introspection-based memory de-duplication approach arises from introspection. Here, (2) and (3) are different for two reasons. First, one could minimize performance impacts on guest VMs by carefully scheduling memory de-duplication operations when the CPUs are less loaded. Second, guest VMs may encounter additional protection faults, context switches and hypercalls (*unshare* in the case of Xen) as a result of writes to pages that are protected by the copy-on-write mechanism. Therefore, a guest VM's run-time performance penalty is dependent on the number of *unshare* calls it triggers.

To isolate the performance benefit of hashing-based and introspection-based memory de-duplication schemes, we compare the following four configurations of the GMD engine, using the **Baseline** configuration as the basis of effectiveness calculation:

- **Baseline**: The GMD engine is totally turned off, no memory pages are de-duplicated and no memory de-duplication overhead is incurred.
- **Intro**: Only the *Introspection* stage of the GMD engine is turned on.
- **Dedup**: Only the *De-duplication* stage of the GMD engine is turned on.
- **IntroDedup**: Both stages of the GMD engine are enabled. Free pages are de-duplicated by the Introspection stage and non-free pages are de-duplicated by the De-duplication stage.

The test machine used in this study contains an Intel Xeon E5640 quad-core processor with VT and EPT enabled, 24 GB physical memory, and a 500 GB hard disk. The host runs Xen-4.1 with CentOS-5.5 as the *Dom0* kernel. All our VMs are configured with 1 virtual CPU and 4 GB memory, which corresponds to something between the *Small Instance* and *Large Instance* classes of Amazon's Elastic Compute Cloud (EC2) service [63] and should be representative for normal server or desktop applications.

We tried guest VMs running 32-bit and 64-bit versions of Windows and Linux, including **Win7-64** (64-bit Windows 7), **WinXP-32** (32-bit Windows XP with Service Pack 2 installed), **Centos-64** (64-bit Centos 5.6 with the 2.6.18 Linux kernel and Sparse memory model configured), and **Debian-32** (32-bit Debian-6.0.2.1-i386 with the 2.6.32 Linux kernel and Flat memory model configured). As for input workloads, we ran the following three benchmarks inside the guest VMs:

- **Video-Creation**, **E-Learning**, and **Office**: These three workloads are from SYSmark2007 [64] and simulate the three different classes of business user behaviors on Windows desktop machines.

- **Banking**, **Ecommerce**, and **Support**: These three workloads are from Specweb2009 [65] and are designed to evaluate web server performance.

- **Specjbb**: Specjbb2005 [66]. A SPEC benchmark that emulates a three-tier client/server system and is designed to evaluate the performance of server-side Java applications.

While Sysmark is a stand-alone benchmark that runs on Windows guests, Specweb is used to generate workloads from simulated clients and require running a web server on a Linux-based guest VM. As for Specjbb, we apply it to both Windows and Linux guest OSs.

In this study, we mainly focused on the memory de-duplication within an individual VM, and ignore inter-VM memory de-duplication, because most of the gains from leveraging free memory pool information come from intra-VM memory de-duplication. As a result, in each experiment, we ran only one VM, on which a particular input application workload is run. Because the GMD engine takes less than one minute to complete one de-duplication round through a VM with 4GB of physical memory, we configured the GMD engine to run once every minute by default. We also varied the invocation frequency of the engine to explore the trade-off between the cost and gain of memory de-duplication.

4.3.2 Correctness of Virtual Machine Introspection

During the process of our bootstrapping introspection, we have collected information from the each of the 4 guest VMs and listed in Table 4.1 and Table 4.2. For Windows OS, our experiment log shows that the kernel base address changes after each system reboot while other values stay unchanged as we have discussed previously. As for the Linux guest, the real-mode kernel stays unchanged for each boot at the same guest physical address. The page descriptor structure information is necessary for Windows but not for Linux because the `PageBuddy` function in the stub is used to identify free pages. Therefore, there is no need to know the Linux page descriptor size and the offset of flags.

In order to show the correctness of our introspection mechanism, we have to prove two things. First, the guest data structures we have seen from the introspection must be the same as the guest OS, e.g., size of the memory

	Win7-64	WinXP-32
Kernel base	0x260b000	0x4d7000
RVA of the `_MmPfnDatabase`	0x2a8220	0x80b48
Memory map	0xfffffa8000000000	0x80c86000
Size of `_MMPFN`	0x2c	0x1c
Offset of `type`	0x1a	0x0d

Table 4.1: Introspection results for locating Windows memory map. The kernel base is in guest physical address while the memory map is in guest virtual address. All addresses or size are measured in bytes.

	Centos-64	Debian-32
Real-mode kernel	0x90000	0x90000
`mem_map`		0xc14b7e00
`mem_section`	0xffffffff80574380	
Memory map	0xffff8100007cb000 0xffff81000098b000 0xffff810000b4b000 0xffff810000d0b000 0xffff810001001000 0xffff8100011c1000 0xffff810001381000 0xffff810001541000	0xc14d1000

Table 4.2: Introspection results for locating Linux memory map. The address of real-mode kernel is in guest physical address while all others are guest virtual addresses. Centos-64 uses `mem_section` for sparse memory model while Debian-32 uses `mem_map` for flat memory model. The memory map of Centos64 contains 8 sections in this case.

descriptor. Second, we remove the free memory of guest OS without contaminating guest states. Without these two facts, the guest could easily crash at run-time especially when the guest is under stress, e.g. benchmark is running. Thus, during the benchmarking of each guest VM we introspect, we always double check the system integrity by looking into the logs of system, applications, and also the benchmark. So far, no error has been found during the experiment, which shows the success of introspection.

4.3.3 Effectiveness of Introspection for Windows

Figure 4.5 shows the comparison of Win7-64 VM among the three GMD configurations, *intro*, *Dedup* and *IntroDedup*, in terms of memory saved, de-duplication overhead, and performance impacts on guest VMs, under four different input workloads. Because the Windows OS zeros out a memory page before putting it in the free memory pool, *Dedup* can de-duplicate any free page that *Intro* can de-duplicate. So in theory, the amount of memory saved by *Dedup* should be larger than that by *Intro*, and is equal to that by *IntroDedup*," as shown in Figure 4.5(a), where the metric is the percentage of the test guest VM's physical memory that is de-duplicated and shared by the GMD engine at the end of each minute during the experiment run. However, for the E-Learning workload, the amount of memory shared by *Dedup* is smaller rather than larger than that by *Intro*, and for the Video-Creation workload, the amount of memory shared by *Dedup* is smaller than rather than equal to that by *IntroDedup*. These anomalies arise mainly because the amount of time and work required to perform one memory de-duplication round through the test VM's physical memory space is different for these three configurations, as shown in Figure 4.5(b). As expected, *Dedup* is the most time-consuming because it needs to perform per-page content hashing and byte-by-byte comparison, *Intro* is the quickest because it only needs to examine specific guest kernel data structures, and *IntroDedup* is between the two extremes because it is a hybrid of *Intro* and *Dedup*.

Figure 4.5(c) shows the average percentage of total memory pages that are *unshared* each minute by the test VM under the four workloads, and mirrors the results in Figure 4.5(a), because the number of pages "unshared" is directly correlated with the number of pages that were previously shared by the GMD engine and later allocated and modified. Two factors affect the performance degradation of the test VM when memory de-duplication runs in the background: (1) the overhead of the GMD engine's own de-duplication operations, e.g., hashing computation and locking of memory pages, and (2) the number of copy-on-write exceptions because of the VM's writes to shared pages. Therefore, the test VM's performance degradations shown in Figure 4.5(d) reflect the

combined effects in Figure 4.5(b) and Figure 4.5(c). Because the differences in the number of unshared pages among the three GMD configurations are small, the performance degradation is influenced more by the de-duplication overhead than by the amount of memory unsharing. Among the three configurations, *Intro* imposes the minimum performance penalty on the test VM. As for WinXP-32, the results are similar as Figure 4.6 shows.

In summary, on the Windows platform, most de-duplicable pages within an individual VM are free memory pages that are zeroed; as a result, *Intro* is able to discover the majority of the de-duplicable pages that *Dedup* can, and the marginal value provided by the vanilla de-duplication stage in *IntroDedup* is relatively minor. On the other hand, the de-duplication overhead of *Intro* is on average four times smaller than *Dedup*'s. In terms of performance impacts on the test VM, *Intro*'s is also significantly smaller than *Dedup*'s. Therefore, as far as intra-VM memory de-duplication for Windows guests is concerned, *Intro* is the clear choice.

4.3.4 Effectiveness of Introspection for Linux

Unlike Windows, the Linux kernel does not zero out a memory page to be freed before putting it in the free memory pool. Consequently, traditional memory de-duplication schemes cannot easily identify these pages and de-duplicate them, whereas the proposed GMD engine can. To ensure that free memory pages are not zero, we wrote a program to allocate as many free memory pages as possible, write random contents to them and then free all of them, before running any experiments. As shown in Figures 4.7(a) and 4.8(a), *Dedup* can barely de-duplicate any memory page, whereas by leveraging free memory map information, *Intro* can still de-duplicate most of the free memory pages. Moreover, the run time of *Intro* is significantly lower than that of *Dedup*, because the latter blindly computes the hash values of all guest physical pages, as shown in Figures 4.7(b) and 4.8(b), For the same reason, the marginal value of the de-duplication stage of *IntroDedup* is also small when compared with *Intro*. Other than the above, the conclusions drawn from Figures 4.7 and 4.8 are similar to those drawn from Figure 4.5. One notable result is that the performance degradation when the test VM runs Specweb is close to zero, as shown in Figures 4.7(d) and 4.8(d). This is because the memory usage of Specweb is pretty static and hence not much memory page unsharing takes place at run time.

4.4 Summary

Traditionally, memory de-duplication is based on page content hashing and byte-by-byte comparison. In this chapter, we demonstrate that it is possible to both increase the amount of memory de-duplicated *and* decrease the de-duplication performance overhead by leveraging kernel state information in guest VMs. In particular, the proposed Generalized Memory De-duplication (GMD) engine exploits virtual machine introspection techniques to identify free memory pages, and de-duplcates them as if they are all-zero pages. We have implemented a fully operational GMD prototype, and successfully tested it against Linux and Windows guest VMs. As far as intra-VM memory de-duplication is concerned[3], when compared with traditional memory de-duplication, the GMD engine on average runs 4 times faster, and is able to de-duplicate a comparable amount of memory on Windows guests and a significantly higher number of memory pages on Linux guests. More concretely, this proposed work makes the following three research contributions:

- Development an bootstrapping VMI mechanism to identify the free memory pools of both Windows and Linux guest OSs,
- Leveraging free memory pool information in memory de-duplication, and
- A thorough study of the three configurations of the GMD engine, *Intro*, *Dedup*, and *IntroDedup*, and an analysis of their de-duplication gains and overheads.

Although the current GMD prototype is built on Xen, we believe it can be easily ported to other hypervsiors, such as KVM [19]. Also, leveraging free memory pool information is one example of applying VM introspection to facilitating memory de-duplication. For example, it is concevable to leverage location information of DLL pages to speed up inter-VM memory de-duplication. Finally, reducing the performance degradation due to memory de-duplication to zero by dynamically tuning the aggressiveness of the GMD engine is another interesting direction to explore.

[3]Note that our mechanism makes no difference between intra-VM or inter-VM de-duplication because the free pages of all VMs can be mapped to a single zero page.

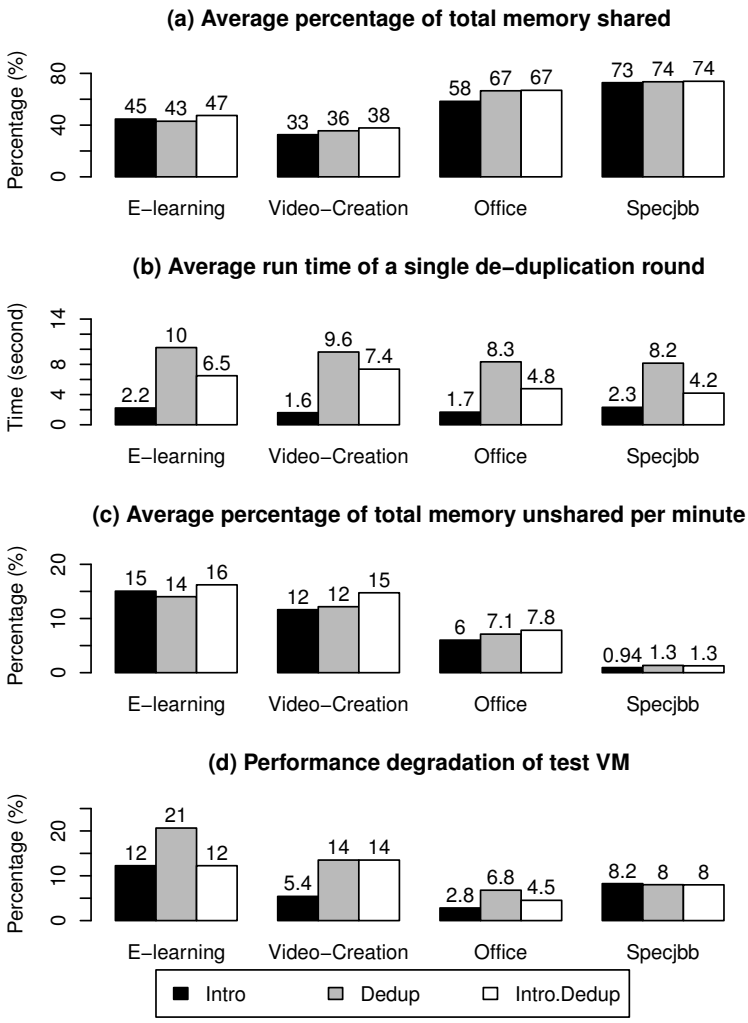

Figure 4.5: *Comparisons among Intro, Dedup and IntroDedup under four different workloads running on a Win7-64 test VM with 4GB physical memory in terms of (a) the average percentage of total memory shared by the GMD engine per minute, (b) the average time required to perform a single memory de-duplication round through the test VM's physical memory space, (c) the average percentage of total memory unshared by the test VM per minute, and (d) the performance penalty experienced by the test VM.*

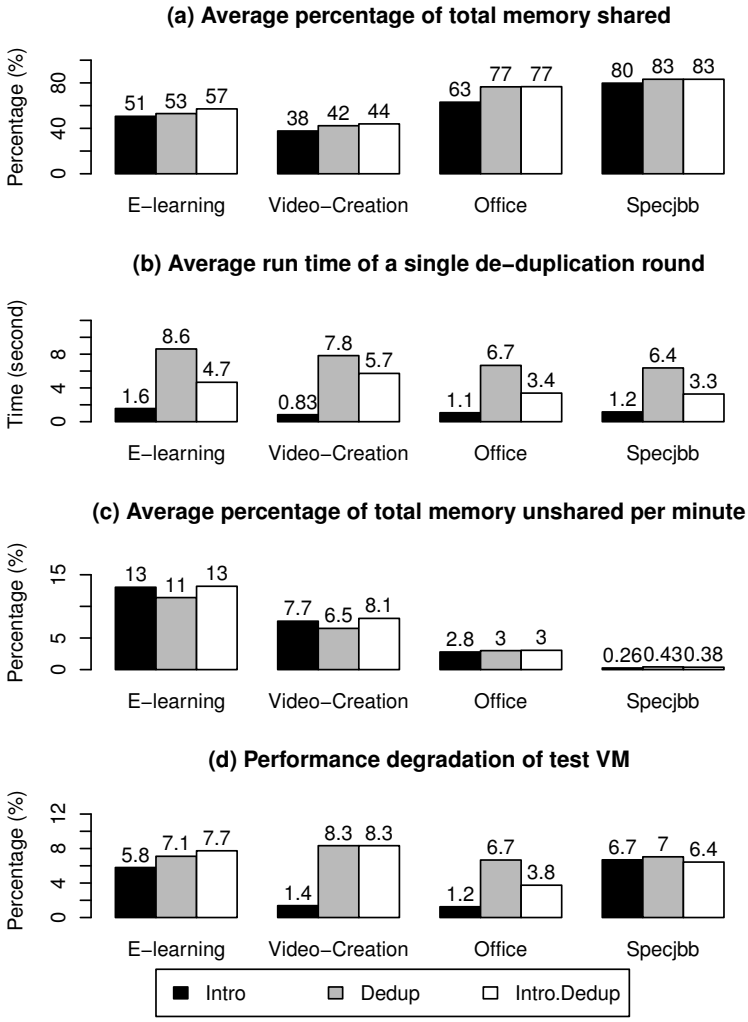

Figure 4.6: *Comparisons among Intro, Dedup and IntroDedup under four different workloads running on a WinXP-32 test VM with 4GB physical memory in terms of the same set of metrics as in Figure 4.5.*

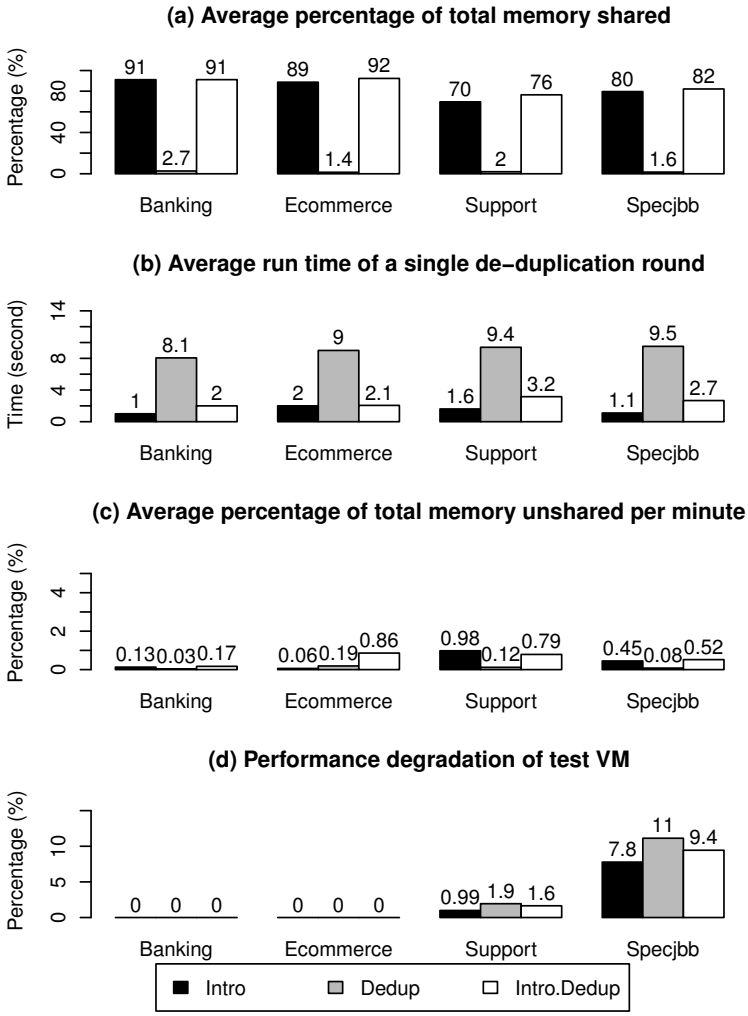

Figure 4.7: *Comparisons among Intro, Dedup and IntroDedup under four different workloads running on a Centos-64 test VM with 4GB physical memory in terms of the same set of metrics as in Figure 4.5.*

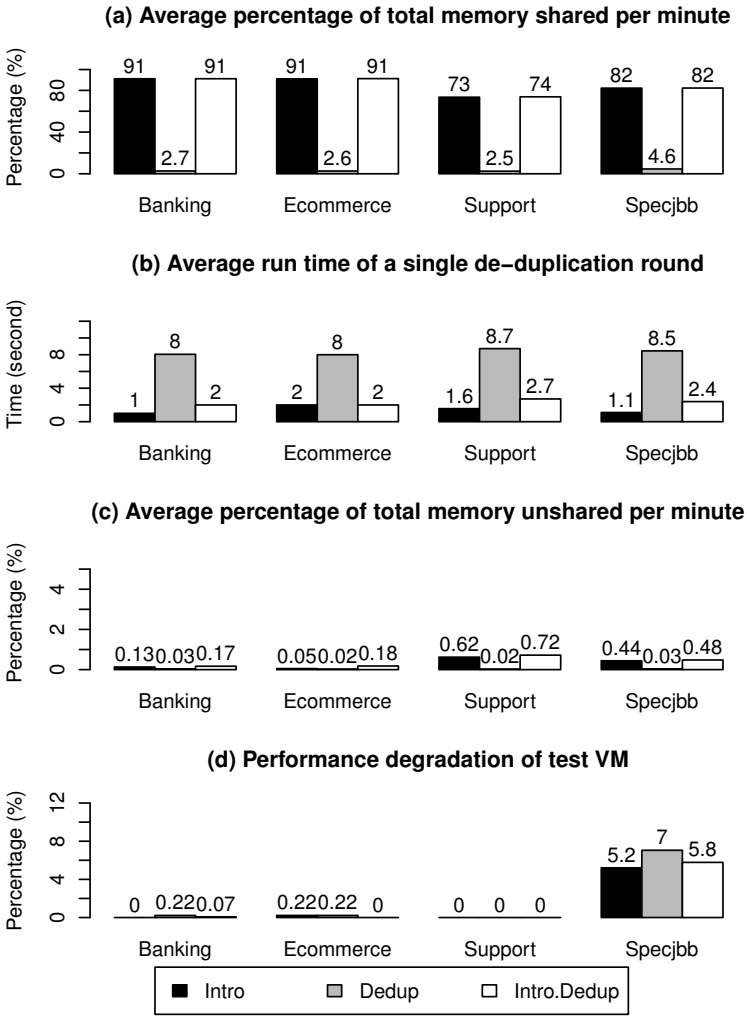

Figure 4.8: *Comparisons among Intro, Dedup and IntroDedup under four different workloads running on a Debian-32 test VM with 4GB physical memory in terms of the same set of metrics as in Figure 4.5.*

Chapter 5

Introspection-assisted Virtual Machine Migration

For resource consolidation or load balancing purposes, modern hypervisors support VM migration, which moves a VM from one physical machine to another. Because the source and target physical machines are assumed to share the same storage server, most of the work involved in VM migration lies in the moving of VM's memory state. In addition to the time consumption of the migration transaction, a major performance metric for VM migration is its impact on the network due to memory state transfer. Because the contents of the free memory pages of a VM are don't-cares, they do not need to be moved when the VM is migrated. Avoiding transferring free memory pages of a migrated VM is thus an effective way to reduce the total time and the network performance impact of a VM migration transaction.

5.1 Skipping Don't-care Pages During VM Migration

When a VM is migrated, its CPU and I/O states are transferred once from the source to the target machine, but its memory state is transferred iteratively using a dirty page tracking mechanism [6]. More concretely, the system first transfers the migrated VM's memory pages to the target machine without stopping the VM. During this period of time, some of the memory pages of the migrated VM may be modified and become dirty. In the second iteration, the system transfers those memory pages that are dirtied in the first iteration; in the third iteration, the system transfers those memory pages that are dirtied in the second iteration, and so on. We applied VM introspection before the first iteration to identify memory pages that do not need to be copied, and thus

reduce the number of memory pages that are transferred in the first iteration.

To transfer the memory state, Xen groups the migrated VM's memory pages into chunks of 1024 pages. In the first iteration, for each chunk, Xen first sends to the target machine a map, *pfn_type*, each entry of which describes the type of each transferred memory page, e.g., *invalid* or *regular* data page, and then sends the contents of all valid pages in the current chunk. To avoid transferring free (don't-care) memory pages to the target machine, we introduced one more type, *free*, to denote pages that are valid but free. By consulting with the *bootstrapping VMI* program described in chapter 4, the Xen hypervisor on the source machine identifies the guest physical pages in a migrated VM that are free, marks the corresponding entries in the *pfn_type* map as *free*, and skips transferring them to the target machine. For all free pages of a migrated VM, as indicated in the received *pfn_type* map, Xen on the target machine deduplicates them to an all-zero page. For the remaining iterations, Xen does not leverage introspection, but focuses only on the transfer of dirtied pages.

	E-learning	Video-Creation	Office	Specjbb
Win7-64	77, 52, 2	75, 40, 4	69, 61, 57	84, 73, 69
WinXP-32	78, 58, 17	77, 43, 0	70, 65, 53	86, 72, 9
	Banking	Ecommerce	Support	Specjbb
Centos-64	93, 92, 90	93, 92, 91	92, 76, 69	91, 81, 77
Debian-32	92, 91, 90	92, 91, 90	93, 75, 68	92, 83, 79

Table 5.1: Percentage of free pages against the test VM's total memory size for four test VMs each under four different workloads where each grid shows the maximum, average, and minimum value.

5.2 Performance Analysis

We used two metrics to evaluate the effectiveness of a VM migration scheme: (1) the amount network traffic injected by a VM migration transaction, and (2) the total VM migration time. Conventionally, the amount of network traffic injected by a VM migration transaction is directly proportional to the total memory size of the migrated VM, i.e., 4GB in our test setup. It takes roughly 40 seconds to transfer this VM's memory state on our Gigabit Ethernet-based testbed, whose TCP throughput is 819.2 Mbps. By leveraging the *bootstrapping VMI* technique to identify free memory pages in a VM that is to be migrated, the hypervisor could skip transferring these free memory pages during

the VM migration transaction, and this significantly cuts down the migration-induced network traffic volume.

As an example, for a Win7-64 VM, Table 5.1 shows that the percentage of available free memory on average is more than 50%; accordingly leveraging the free memory pool information could cut down the associated network traffic volume by more than 50%. In practice, the hypervisor tends to migrate a VM when it is least loaded or the amount of free memory is the highest; therefore, the actual reduction in migration-induced network traffic load is likely to be even higher.

To assess the performance benefit of introspection-based VM migration, we compare the following two VM migration schemes using the above two metrics:

- **BaseMigrate**: The conventional VM migration scheme implemented in Xen.
- **IntroMigrate**: BaseMigrate with the optimization that avoids transferring free memory pages as identified via VM introspection.

In each run, we triggered a migration of the test VM at a randomly chosen time and measured the network traffic load and migration time, and reported the average of the measurements of multiple runs. Due to space constraint, we only present the results of two types of test VMs, Win7-64 and Debian-32, because WinXP-32 and Centos-64 have similar results.

Figure 5.1(a) compares the injected network traffic volumes of *BaseMigrate* and *IntroMigrate* for a Win7-64 VM under four different test workloads. Compared with *BaseMigrate*, *IntroMigrate* reduces the network traffic in the first iteration of memory state transfer, depicted as "1st-Iteration" in the figure, by 48%, 41%, 62%, and 81% for E-learning, Video-Creation, Office, and Specjbb, respectively. As expected, the percentage of network traffic reduction is roughly proportional to the average percentage of free pages shown in Table 5.1, because information extracted by introspection is used only in the first iteration.

For the remaining iterations of memory state transfer, depicted as "Remaining" in the figure[1], surprisingly *IntroMigrate* also cuts down the network traffic volume by 8%, 57%, 75%, and 9% for E-learning, Video-Creation, Office, and Specjbb, respectively, even though no introspection-derived information is used in these iterations. This reduction originates from the fact that when the first iteration is shortened, fewer memory pages are dirtied in the first iteration, the second iteration is also shortened, even fewer pages are dirtied in the second iteration, and so on. The introspection benefit to the remaining

[1]The CPU and I/O states are only a few KBytes and thus ignored in this discussion.

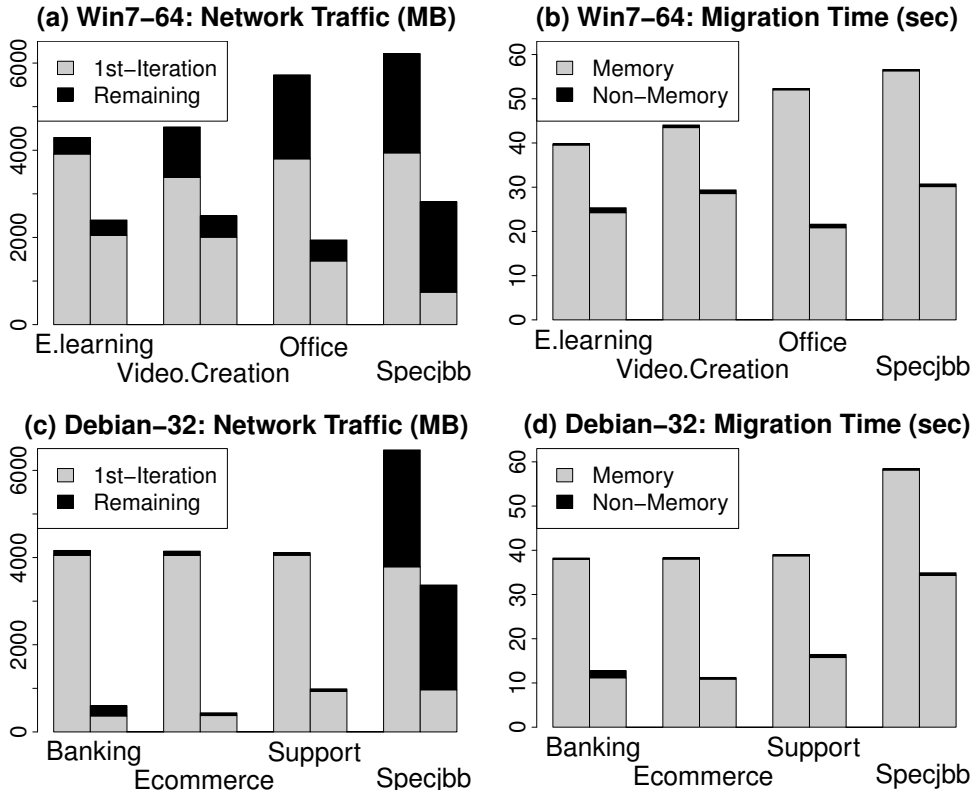

Figure 5.1: *Comparison of injected network traffic volume and total migration time between $BaseMigrate$ and $IntroMigrate$ for four different workloads running on a Win7-64 VM and a Debian-32 VM. For each workload, the left and right bar represent the result of $BaseMigrate$ and $IntroMigrate$ respectively. In subfigure (a) and (c), "1st-Iteration" and "Remaining" correspond to the injected network traffic volume in the first and remaining iterations during a VM migration transaction, respectively. In subfigure (b) and (d), "Memory" and "Non-memory" correspond to the memory state migration time and the migration time for other VM states, respectively.*

iterations is apparent for the Video-Creation and Office workload, but not so obvious for E-learning because the network traffic due to the remaining iterations is small to begin with, and for Specjbb because it is memory-intensive and introduces a large number of dirtied pages in the remaining iterations regardless of the length of the first iteration.

Figure 5.1(b) shows that *IntroMigrate* cuts down the total migration time from 40, 44, 52, and 56 seconds to 25, 29, 21, and 30 seconds, or by 38%, 34%, 60%, and 46% for E-learning, Video-Creation, Office, and Specjbb, respectively. The amount of migration time reduction is proportional to the amount of reduced network traffic or memory state transfer, which accounts for more than 96% of the total migration time. The non-memory portion of the migration time is too small to be noticeable.

Figure 5.1(c) and 5.1(d) show similar benefits for the Debian-32 test VM. When compared with *BaseMigrate*, *IntroMigrate* reduces the network traffic load due to VM migration by 85%, 89%, 76%, and 48%, and the total migration time by 66%, 71%, 59% and 40%, for Banking, Ecommerce, Support, and Specjbb, respectively. Unlike the Win7-64 test VM, introspection does not benefit the remaining iterations much even when it produces a significant benefit in the first iteration for all four test workloads running on the Debian-32 VM. In addition, the reduction percentage in total migration time is not as significant as the reduction percentage in memory state transfer, and the ratio between these reduction percentages is smaller in the Debian-32 VM than in the Win7-64 VM. For example, for the Banking workload running on the Debian-32 VM, the network traffic reduction percentage due to introspection is 85% but the migration time reduction percentage due to introspection is only 66%; for the E-learning workload running on the Win7-64 VM, the network traffic reduction percentage due to introspection is 44% but the migration time reduction percentage due to introspection is only 38%. The reduction percentage ratio is 0.77 (66%/85%) for the Debian-32 VM and 0.86 (38%/44%) for the Win7-64 VM.

In the current prototype implementation, the VM migration module first asks the hypervisor to map all the pages in the migrated VM and then queries the hypervisor for each page's type information. This mapping and query step incurs a fixed overhead, which accounts for the discrepancy between the reduction in amount of memory state transfer and the reduction in total migration time. More specifically, this fixed overhead becomes relatively more significant when the total migration time becomes smaller. Consequently, when the performance benefit of *IntroMigrate* increases, the reduction in the total migration time increases, the relative importance of this fixed overhead increases and finally the discrepancy also increases. Because the performance benefit of

IntroMigrate is more pronounced in the Debian-32 VM than in the Win7-64 VM, the discrepancy between network traffic volume reduction and total migration time reduction is therefore larger in the Debian-32 VM than in the Win7-64 VM.

5.3 Summary

Besides for the memory de-duplication in chapter 4, we have demonstrated the VM migration as another application of the proposed *bootstrapping VMI* technique. The total time and network traffic of VM migration is effectively reduced with more guest knowledge in the hypervisor context.

Chapter 6

Working Set-based Memory Allocation and Compression

6.1 Working Set Estimation

To make memory compression more efficient, it is essential to identify the working set of a guest and compress all its pages outside of the working set. To achieve this, there exist two issues: (1) how to identify a guest's working set and (2) how to determine the *working set size* (WSS) of a guest VM, i.e., the number of pages in the working set.

Intuitively the working set of a guest VM is defined as the amount of memory being actively used by the VM in the recent past [18]. One way to determine the working set of a VM is to intercept memory accesses, which can be achieved by marking the guest memory as not-present in the Extended Page Table (EPT) to trap and record the number of accessed pages. Then, the WSS of the VM at any moment is equal to the number of accessed memory pages of the VM, and the memory that is not accessed, i.e., referred to as *cold* pages, can be reclaimed by the hypervisor and used to satisfy future memory allocation requests. However, this solution is not feasible because the overhead of trapping on every memory read/write is simply too prohibitive to be practically feasible.

The *Self-ballooning* mechanism [67] on top of the Xen hypervisor estimates a guest VM's WSS by using the the `Committed_AS` statistics maintained by Linux,[1] which is the total size of *anonymous memory* consumed by all processes, and automatically reclaims the unused memory of the guest into the hypervisor's free memory pool. The Linux OS identifies a guest VM's

[1] In this chapter, we focus on the Linux guest OS and thus use the terminology used in Linux documents.

`Committed_AS` using the reclamation mechanism described in Chapter 2.6. When the Self-ballooning mechanism is activated in a guest VM, it periodically sets the VM's balloon target to the current `Committed_AS` and reclaims unused memory pages. This approach guarantees that applications consuming anonymous memory will not suffer from any swap-in delay because all their stacks and heaps are likely to stay in memory. However, this approach exhibits two deficiencies.

First, the Self-ballooning mechanism cannot take into account applications' disk I/O activities when computing the working set because `Committed_AS` only focuses on consumed anonymous memory. Unlike anonymous memory, the Linux kernel does not factor page cache need to the working set calculation. The page cache is automatically populated by the *virtual file system* (VFS) layer of the Linux kernel upon a disk access for the first time. If a page cache page belonging to the working set is evicted due to memory reclamation, a *refault* event occurs and thus could be used as a signal that one more page should be added to the working set to accommodate the page cache page. Based on this observation, the TWS-ballooning mechanism we proposed maintains a counter for refault events in the guest, and adjusts the balloon target according to the refault count so that the performance penalty resulting from evicted page cache can be minimized.

Second, `Committed_AS` represents an upper-bound on the anonymous memory a guest VM consumes, but does not necessarily correspond to the VM's working set size. More specifically, `Committed_AS` is incremented upon the first access to each newly allocated anonymous memory page but is decremented only when the owner process explicitly releases the page. For example, if a program allocates and accesses a memory page only once when the program starts but leaves it untouched until the program exits, the Linux kernel cannot exclude this cold page from `Committed_AS` even though it is clearly outside the working set. Our TWS-ballooning algorithm gets around this limitation by actively probing each guest VM's true working set.

6.2 Problem Classification and Design

A generic way to leverage the guest kernel's page reclamation mechanism is to ask the balloon driver running in a guest to lower the balloon target (inflate) i.e., allocate more memory pages via standard memory allocation APIs, and return the allocated memory pages to the hypervisor. When a guest's balloon driver allocates more than the current size of its free memory pool, the guest's page reclamation mechanism is triggered to evict cold pages.

To determine the best balloon target for every guest VM running on a

physical machine, especially when the total amount of physical memory available to them is limited, is a crucial design issue in memory virtualization. To answer these questions, let's review four possible physical memory allocations for a given guest VM, from high to low.

- Baseline (**BASE**): a VM is allocated with its configured memory, e.g., the hypervisor allocates 4GB memory to a VM configured with 4GB memory, which is generally larger than the total memory requirements of the applications running on the VM. The performance and characteristics of the applications running inside such a VM are treated as the baseline for comparison.

- Committed_AS (**CAS**): a VM is allocated with an amount of memory equal to its Committed_AS.

- True Working Set (**TWS**): a VM is allocated with an amount of memory equal to its true working set. A guest VM's TWS is lower than its Committed_AS if the guest VM does not perform significant disk I/Os, which require additional buffer cache pages included in the working set.

- Minimum Memory Requirement (**MMR**): a VM is allocated with an amount of memory equal to the minimum memory requirement, which is generally lower than the TWS. If a guest VM's allocated memory is lower than MMR, the guest VM is likely to encounter *Out-Of-Memory (OOM)* exceptions.

To pack more VMs into one physical host where the host memory is smaller than the sum of the configured memory requirements of all VMs, one must decrease the memory allocation of each VM from its BASE level to lower levels, i.e., CAS, TWS or MMR, while keeping the application performance at the same level.

The *Self-ballooning* mechanism treats a guest VM's CAS as its true working set size, and this estimate is inaccurate because it fails to account for idle anonymous memory pages and page cache needs.

When a guest VM's physical memory allocation is equal to its WSS, the disk access overhead associated with swapin and refault should be close to zero. To probe a VM's true working set, we gradually increase the balloon target of the balloon driver in the VM, and stop until the VM's swapin and refault counts start to become non-zero.

Based on the observation, we define the *estimated working set size* (**EWSS**) of a VM as follows:

$$EWSS = allocated\ memory\ size + overhead\_count \tag{6.1}$$

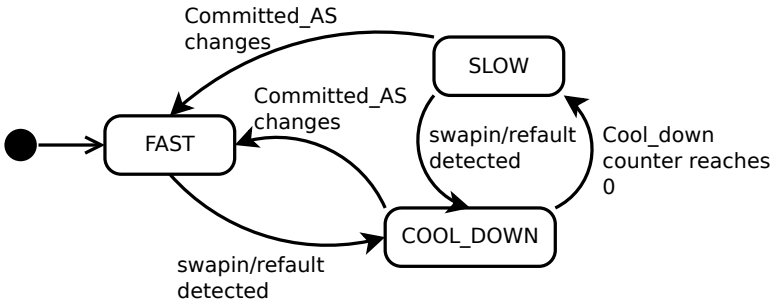

Figure 6.1: *State transition diagram of TWS-ballooning.*

where the *allocated memory size* is the number of memory pages allocated to a VM and *overhead_count* is the number of pages faulted into the VM. Based on this estimate, the TWS-ballooning mechanism is able to cut a VM's allocated memory from CAS to TWS, leaving all pages outside the working set to be evicted.

Because these evicted pages are considered cold, they make good candidates for compression. To compress these cold pages, we intercept the eviction flow of the guest's page reclamation mechanism, and store the evicted pages in the compressed form inside the guest VM. However, compressed memory pages consume memory space and thus put additional pressure on the guest VM, affecting the VM's application performance. Thus, we have designed an on-line memory compression mechanism called *dyn_memlimit* to determine the optimal amount of VM memory that should be compressed to minimize the application performance overhead when the amount of physical memory allocated to a VM is between its TWS and MMR. Moreover, when multiple VMs are running on the same physical machine, our *memory balancing* mechanism properly appropriates the available memory among all VMs so as to equalize their performance overhead due to memory contention.

6.3 True Working Set-based Ballooning

When a VM's allocated memory amount is above TWS, our TWS-ballooning algorithm uses a customized balloon driver, referred to as `zballoond`, to gather swapin and refault counts every second and adjusts the balloon target to probe the VM's true working set. The upper-bound of a VM's balloon target is set to the VM's configured memory size[2] when the VM boots up, but the lower-bound depends on the amount of memory pinned down by the VM's OS.

[2]The size configured by the hypervisor when the VM is started.

More concretely, we leverage the heuristics used in the *Self-ballooning* algorithm to calculate the initial lower-bound value, and add to it memory requirements reserved for system emergency and compressed pages. Without this adjustment, a guest can easily encounter an *Out-Of-Memory (OOM)* exception, when its Committed_AS is low. We incorporate this adjustment into the Self-ballooning mechanism as well.

To better approximate a VM's true working set size, the zballoond of the TWS-ballooning algorithm employs a finite state machine that has three runtime states as shown in Figure 6.1, and adjusts the balloon target adaptively. Starting from the *FAST* state, the zballoond initializes the balloon target to Committed_AS[3], and iteratively lowers it by 5% of its current Committed _AS. Whenever swapin or refault events occur, zballoond raises the balloon target by an amount of memory whose page count is equal to the combined swapin and refault count, because each swapin or refault event suggests the need for an additional free memory page. These swapin and refault events indicate that either the balloon target is approaching the true working set or there is a sudden burst in memory demand from applications; it is thus not wise to further decrease the VM's memory allocation.

Unlike the Self-ballooning algorithm, even when swapin or refault events happen, the TWS-ballooning algorithm actually allows the memory allocation to a VM to exceed its Committed_AS. This flexibility is especially important for VMs running a disk intensive workload where their Committed_AS does not reflect the additional memory demand due to page caching.

Whenever a swapin or refault event occurs, zballoond stops lowering the balloon target by switching its state to *COOL_DOWN*, and initializing a Cool_down counter (now arbitrarily set to 8) and decrementing it every second from then on. When the Cool_down counter reaches zero, zballoond considers the workload burst already gone, and then switches its state to *SLOW*, which applies the same logic used in *FAST* state except that the balloon target is iteratively lowered by 1% of the current Committed_AS.

When a VM's Committed_AS changes, zballoond considers the VM's working set size is going to change significantly, and resets itself by entering *FAST* state. However, there are two things to note at this moment. First, if the balloon target exceeds Committed_AS because of swapin or refault bursts, before entering the *FAST* state, the balloon target is initialized to Committed_AS plus the exceeded amount. Second, when zballoond switches from *COOL_DOWN* to *FAST* state while the Cool_down counter has not reached zero, zballoond enters *FAST* state but continues the count-down until it reaches zero, before resuming the working set probe.

[3]This is because there is no explicit information for page cache at this point.

Pseudo Page Fault (usec)		True Page Fault (usec)	
zram swap-in	zram swap-out	disk swap-in	disk swap-out
86	24	5775	2328

Table 6.1: The comparison of pseudo page fault overhead and true page fault overhead. Note that the actual fault overheads may vary under different workloads and memory pressures.

6.4 TWS-aware Memory Compression

When a VM's allocated memory is between its TWS and MMR, memory compression could substantially mitigate the performance degradation due to memory pressures. To compress evicted pages from a guest VM, we leverage the *zram* [7] driver to intercept the VM's swapinand swapout operations. The zram kernel module is inserted into a guest Linux kernel as a virtual disk device, and configured as the swap disk by the system management tool, i.e., swapon, so that all swapin and swap-out operations enter the zram driver as disk I/O requests. When a swapped-out page arrives at the zram driver, it is compressed into a **sub-page** size by the LZO1X (Lempel-Ziv-Oberhumer) algorithm and stored in a memory area allocated from the guest kernel without being sent to the bare metal hard disk. One exception is zero evicted pages, which zram recognizes based on the *page type* information and skips the compression step. When a swap-in page arrives, zram uncompresses the page and returns it to the process that causes the page fault triggering the swap-in.

Although we include buffer cache pages into a VM's working set, we do not compress evicted buffer cache pages in our current implementation for the following two reasons. First, the lifetime of a process's anonymous pages is the same as that of the process itself because they are released when the owner process dies. However, buffer cache pages are not explicitly owned by any process, because they could be allocated by one process and then used to satisfy disk accesses by another process. Second, compared with anonymous memory pages, the buffer cache is typically backed by a larger disk volume, and thus may require too much memory to compress it. While intercepting swap-in and swap-out of anonymous memory pages is relatively straightforward because it could be done through a well-defined API, the same thing cannot be said about intercepting eviction of buffer cache pages, whose logic is embedded in the VFS layer of the Linux kernel. In the end, we choose not to compress evicted buffer cache pages and focus only on swap-in events associated with anonymous memory.

6.4.1 Pseudo Page Fault versus True Page Fault

For a VM is backed by a zram driver and a swap disk, when a page fault occurs, this missing page could be fetched from the zram driver, in which case the page fault leads to a pseudo page fault, or from the swap disk, in which case the page fault leads to a true page fault. When a page is swapped in from the zram driver, the overhead is mainly due to the time required to decompress the page. When a page is swapped out to the zram driver, the overhead is mainly due to the time required to compress the page. Table 6.1 shows a quantitative comparison between the swap-in and swap-out times associated with a pseudo page fault and a true page fault, and the difference between their overheads is at least a factor of 50.

When a larger portion of a VM's memory is given to its zram driver, less memory is available to the applications running on the VM, and the pseudo page fault rate is increased. However, as the zram driver is given more memory, more pages are held in memory effectively due to compression and fewer page faults result in true page faults, because they are more likely to be satisfied by the compressed pages in the zram driver. Therefore, the amount of memory given to the zram driver represents a trade-off between pseudo page fault rate and true page fault rate.

Suppose the amount of memory allocated to a VM is M, C of which is allocated to the VM's zram driver, and the average compression ratio of the pages stored in the zram driver is X. Then the key design question here is to find the optimal C such that $PPFR(M, C) * Overhead_{PPF} + TPPR(M, C) * Overhead_{TPF}$ is minimized, where $PPFR(M, C)$ is the pseudo page fault rate of a VM when its allocated memory is M and C of which is allocated to the VM's zram driver, and $TPPR(M, C)$ is the true page fault rate of a VM when its allocated memory size is M and C of which is allocated to the VM's zram driver. Of course, the exact variations of $PPFR()$ and $TPFR()$ with their arguments very much depend on applications on the VM in question, and are not at all linear in most cases.

The technical formulation of TWS-aware memory compression becomes as follows: Given a VM, how to automatically deduce the optimal percentage of the VM's allocated memory that should be assigned to its zram driver, and the subset of memory pages evicted to the zram driver that should be sent to the swap disk.

6.4.2 Dynamic Adjustment of Zram Size

The zram driver provides a control parameter, *memlimit*, which specifies the amount of memory it can use to store compressed swapped-out pages, exclud-

ing the zero pages. When the amount of used memory in a VM's zram driver exceeds *memlimit*, it simply directs all future swapped-out pages from the VM to the backing swap disk without attempting to compress them. Initially, we set the *memlimit* parameter of a VM to $balloon\ target - MMR - M_{zram}$, where MMR is the VM's minimum memory requirement and M_{zram} is the basic memory requirement of the zram driver itself.

Suppose there exists a LRU list that orders all the memory pages ever accessed by a VM according to the last access time, where the hottest pages are those that are actively in use and the coldest pages are those that have not been touched for a while. Assume the number of pages available to the VM's applications is $N1$ and the number of pages allocated to the zram driver is K. Then the hottest $N1$ pages in the VM's LRU list should be uncompressed and stay in the VM's memory outside the zram driver. The next hottest $N2$ pages, whose accumulative size after compression is K, should be placed in the VM's zram driver. The remaining pages in the LRU list should be in the VM's swap disk.

If $N2$ is decremented by one so that $N1$ is incremented by one, some of the coldest pages in the zram driver that is stored in a compressed form now have to reside in the swapping disk and be explicitly brought into memory when they are accessed, and at the same time the hottest page in the zram driver could be held in the VM's memory. That is, the pseudo page fault rate is decreased but the true page fault rate is increased. Therefore, it is preferable to decrement the number of memory pages assigned to the zram driver if the reduction in the pseudo page fault overhead out-weighs the increase in the true page fault overhead. Similarly, it is preferable to increment the number of memory pages assigned to the zram driver if the reduction in the true page fault overhead out-weighs the increase in the pseudo page fault overhead.

Assume the $(N1+1)$-th page to the $(N1+N2)$-th page in a VM's LRU list are stored in the VM's zram driver, it is preferable to decrement $N2$ if the condition of the following equation is met.

$$\begin{aligned} &AccessProbability(N1+1) \times Overhead_{PPF} > \\ &\sum_{j=m}^{N1+N2} AccessProbability(j) \times Overhead_{TPF} \end{aligned} \tag{6.2}$$

The $AccessProbability(N)$ is the access probability of the N-th page in the LRU list, and the sum of the sizes of the coldest $N1+N2-m+1$ compressed pages is less than the page size but the sum of the sizes of the coldest $N1+N2-m+2$ compressed pages is larger than the page size. We continue to decrement $N2$ as long as the above inequality holds, until either $N2$ becomes

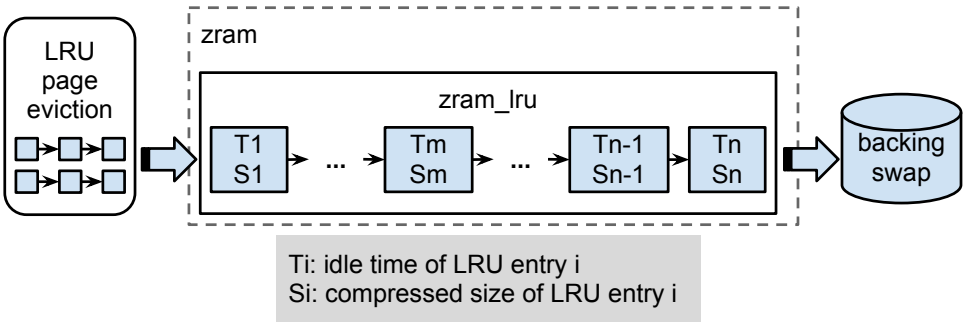

Figure 6.2: *The LRU list maintained in the zram driver, called zram_lru, where hotter pages are on the left and colder ones are one the right side.*

zero. The above algorithm is called *dyn_memlimit* in the following sections.

The LRU list above includes all the memory pages accessed by a VM, and incurs too much performance overhead to build in practice. Instead, we opt to build a local LRU list called *zram_lru* based on the pages swapped out to the zram driver. The access probability of each page in this list is estimated by the inverse of the page's idle time, which is defined as the difference between the time when it is swapped out and the current time, as shown in Figure 6.2. This estimate of a page's idle time is an approximation because it equates a page's swapped-out time as its last access time. When a VM evicts pages more frequently, this approximation is more accurate. When a VM does not evict pages frequently, there is no need to adjust the zram memory dynamically and the fact that this approximation is less accurate does not have much impact.

6.5 Memory Balancing of Virtual Machines

When multiple VMs run on a physical machine and the sum of their TWSs exceed the available physical memory on that machine, the hypervisor should allocate the physical memory among these VMs in a fair way. We call this operation *memory balancing*. After memory balancing computes a memory allocation for the VMs, each VM runs the *TWS-ballooning* algorithm in the *COOL_DOWN* state.

6.5.1 Evolution of Memory Balancing Mechanisms

There are multiple possible definitions of fairness. We explore the following four in this section and see how one mechanism evolves to the next.

- **Bal_equal**: The simplest form of memory balancing is to divide the

available physical memory by the number of VMs, and give each VM an equal amount of memory. However, this mechanism does not take into account the working set size of each VM. The implicit assumption of this approach is every VM is identical, including the applications running on it and their input workloads.

- **Bal_prop**: Similar to the memory allocation scheme used in VMware's ESX server [18], this approach assigns to each VM a percentage of the available physical memory that is proportional to its working set size. The intuition is to give larger memory size to the VM with larger demand on memory resource. With this allocation, the difference between a VM's working set size and its allocated memory, i.e., the *overhead_count* in Equation 6.1, is also proportional to the VM's working set size. This means that the additional page fault penalty when a VM's memory allocation is decreased from its working set size to a fixed percentage of its working set size may be higher for VMs with a larger working set size.

- **Bal_count**: To equalize the additional page fault penalty experienced by each VM when compared with the case when its memory allocation is its working set size, we first quantify the *overhead_count* for each VM, and strive to assign each VM the same overhead_count. Towards this goal, we make one assumption: the overhead of each swapin and refault event is the same across different VMs with different workloads and allocated memory size This assumption has to be true when we quantify the overhead by overhead_count; otherwise, even if we balance the overhead_count of each VM, the true VM overhead is not balanced.

  Given a physical host with memory size M_{Avail} and N VMs where each VM VM_i has working set size WSS_i, we first subtract M_{Avail} from the sum of the VMs' working set sizes, and then divide the subtraction result by the number of VMs. Subtracting the division result from WSS_i produces the memory allocation or balloon target for VM_i.

$$BalloonTarget_i = WSS_i - \frac{(\sum_{i=1}^{N} WSS_i) - M_{Avail}}{N} \tag{6.3}$$

- **Bal_time**: The assumption of Bal_count is not always true because the time cost of each swapin or refault event may vary within one VM or between different VMs. For example, when the allocated memory of a VM is much lower than its true working set, the guest OS's swapping mechanism has to work harder, due to more frequent synchronizations between the Active and Inactive list, more frequent scheduling of the

kswapd kernel daemon, and more modifications on the metadata of the swap subsystem. These collectively slow down each swapin operation.

To remove this assumption, the proposed *Bal_time* mechanism is aimed to balance the swapin and refault time, referred to as ***overhead_time***, among all VMs. Here, the sum of swapin and refault time of each VM_i is referred to as $Overhead\_time_i$. The reduction in the memory allocation of the i-th VM, S_i, is proportional to the inverse of the $Overhead\_time_i$, which is the time spent on swapin and refault, because the higher $Overhead\_time_i$ is, the smaller the cut of the memory allocation of the i-th VM:

$$S_i = ((\sum_{i=1}^{N} WSS_i) - M_{Avail}) * \frac{\frac{1}{Overhead\_time_i}}{\sum_{i=1}^{N} \frac{1}{Overhead\_time_i}} \tag{6.4}$$

Then, the new balloon target of VM_i can be calculated as the following.

$$BalloonTarget_i = WSS_i - S_i \tag{6.5}$$

6.5.2 Analysis of the Memory Balancing Mechanisms

One implicit assumption for all balancing mechanisms who depend on the working set size, except for the Bal_equal, is that the current EWSS of a VM is applicable to predict its EWSS for the next moment when its memory allocation is changed. For example, when a VM is allocated with 100 memory pages and has 100 overhead_count, if we cut 1 page from it, its memory allocation is changed to 99 memory pages, and we like to see 101 overhead_count from the VM.

Given two VMs each with working set size 600 and 300 MB as an example, when the total host memory is 600 MB, we analyze how our memory balancing mechanism evolves to perform the memory allocation. First, we can choose to use Bal_equal mechanism to divide the memory into half. The VM 300 is allocated with 300 MB which just fits to its working set size, and it can perform natively without problem. However, the VM 600 has 300 MB deficient, and thus has significant performance overhead, e.g, 300 MB of swapin and refault events.

Then, we try to consider the working set size of each VM and allocate the memory proportional to it. The VM 600 and 300 will be allocated by 400 MB and 200 MB respectively. But the overhead_count of VM 600 is expected to be larger because it has 200 MB deficient while the VM 300 has only 100 MB deficient, and the overhead is not balanced.

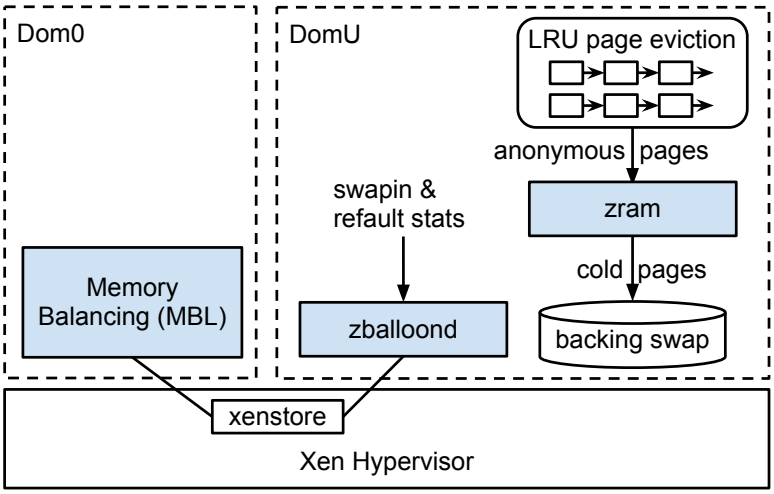

Figure 6.3: *Software architecture of zram, zballoond and MBL components. The modified or newly added components are drawn in shaded blocks.*

Now we try the Bal_count mechanism by dividing into half the deficient amount of total host memory, i.e., 150 MB (divide 300 MB into half), and assigning the same deficient amount to each VM. Then, the VM 600 and 300 will be allocated by 350 MB (150 MB deficient) and 150 MB (150 MB deficient) respectively, and we expect to see them to have the same overhead_count after the memory allocation.

However, the true performance overhead may not be equalized even if we balance the overhead_count of the two VMs. Assume the overhead_time ratio of VM 600 and VM 300 is 2 to 1, we consider to apply the Bal_time algorithm, which assign the deficient amount to VMs with inverse ratio of overhead_time, i.e., 100 MB deficient for VM 600 and 200 MB deficient for VM 300. Then, the final memory allocation is 500 MB for VM 600 and 100 MB for VM 300. We expect the overhead_time of the two VMs are balanced and the VM performance overhead is equalized. In the evaluation section of this chapter, we will compare and analyze the results of these mechanisms when they are used to balance VMs with different working set size.

6.6 Software Architecture and Implementation

Figure 6.3 shows the software architecture of all the software components we have talked about so far, including zram, zballoond and the Memory Balancing (MBL) component in which we also implemented all three balancing mechanisms described in chapter 6.5. Using the Xen hypervisor as the underlying

platform, we describe the implementation and interaction between the above three software components.

- **zram**: the zram device is configured as the default swap disk inside the Linux guest, i.e., the *domU*, which services the swap I/O requests of the evicted anonymous pages from Linux LRU page eviction mechanism. To dynamically adjust the zram memory size as described in chapter 6.4, We modify the original code of Gupta [7] to add LRU list for all compressed memory pages in zram, and record the access time of the LRU entry. When the compressed memory has exceeded the *memlimit* or when we decide to shrink zram size, the cold pages in zram LRU will be evicted to the configured backing swap disk as the figure shows.

- **zballoond**: it is the major component implemented as a guest kernel module with a kernel thread running the main function, which wakes up every second to collect information, change the zballoond state, and make decision of the balloon target. Also, the mechanism to adjust zram size is also triggered by zballoond. To decide the current zballoond state and balloon target, the zballoond requires three kinds of information. The first one is the *Committed_AS*, which can be retrieved from the exported kernel variable[4] `vm_committed_as`.

  The second one is the swapin information including the number of swapin event and the total time cost of them. While the swapin count can be retrieved from the `pswpin` variable of `vmstat` kernel component, which is Linux kernel statistics on its memory subsystem, the swapin time consumption can be retrieved from the `delayacct` component [68], which is a built-in kernel component of per-process delay accounting including the time consumption of page fault. Note that we count for all processes for the swapin time consumption.

  Finally, the zballoond needs the information of refault from disk I/O. The I/O operation of guest disks can be intercepted by modifying the disk driver inside the Linux guest. For our testbed, we configure the Linux guest to use the frontend disk driver, which is open-source from the Xen community. Another alternative is to use the `blktrace` facility in Linux kernel, which provides APIs to add hook point to the disk I/O so that we can perform statistics there. Once the I/O is tracked, we initialize a zero bitmap for each block on the disk and set one bit in the map to one when the corresponding disk block is accessed. A refault

[4]The exported kernel variable in Linux can be accessed by any kernel component or loadable kernel modules.

event is counted when we discover a bit is already set before changing it to one, which means the block has been accessed before, but now requires another access, and we consider this as the performance penalty of evicted page cache, e.g., when the TWS-ballooning moves down the balloon target according to *Committed_AS* which does no account for the page cache usage. All these information are sent to the centralized MBL component every second.

- **MBL**: the *Memory BaLancing* (MBL) component is implemented as a user process running inside the privileged VM, *dom0*, and communicates with the guest zballoond via the `xenstore` component, which is the default event delivery subsystem in Xen hypervisor. The MBL is activated only when it detects the memory of Xen is below 1%, which is our configured lowerbound for emergency memory pool of hypervisor. The MBL component receives the swapin and refault information from the zballoond of each guest VM , decides the working set of each VM, and calculate the new balloon target depending on the which of the four memory balancing mechanism is chosen. In order to collaborate with the TWS-ballooning, the MBL set the *upperbound* of the balloon target into zballoond running inside each guest. When the zballoond discovers the upperbound is lower than the current balloon target, calculated by the algorithm of TWS-ballooning, it overwrites its decision and lowers down the balloon target right away. On the other hand, if the upperbound is higher than the current balloon target, the zballoond does nothing, but depends on the TWS-ballooning mechanism to adjust the balloon target autonomously.

  From the experiment result, we found that the reported swapin and refault sometimes go up and down with certain spikes in the statistics, which makes it difficult for the memory balancing algorithm to converge. Therefore, when the MBL is active, we choose to adjust the VM balloon target every 5 seconds instead of every second, and apply *moving average* on the swapin and refault information to remove the spikes so that the memory balancing can observe the memory growing/shrinking trend of each VM and make the right decision.

6.7 Performance Evaluation

As the discussed four levels of memory allocation to each VM, i.e., BASE, CAS, TWS, and MMR, we evaluate our entire work starting from the scenario with high host free memory to the one with low host free memory. The test

machine used in this study contains an Intel Core i7 quad-core processor with VT and EPT enabled and 16 GB physical memory. The host runs Xen-4.1 with 64-bit vanilla Linux 3.2.6 as the *Dom0* kernel. All our VMs are configured with 1 virtual CPU and run Linux 3.2.6 64-bit kernel with our zram and zballoond components. The test VMs we used here can be configured with the ***memory upperbound***, i.e. the upperbound of balloon target, to confine its memory usage. When there are multiple VMs involved, the memory upperbound of the VMs limit the sum of the balloon target of each VM.

To verify the effectiveness of the TWS-ballooning and dyn_memlimit in the real world, we have compared them with the latest VMware ESXi 5.0 server.[5] When we perform the comparison with the ESXi server, we use two identical test machines where one hosts the Xen hypervisor with our developed memory virtualization mechanisms while the other one hosts the ESXi server. For the ESXi server, except for the memory de-duplication that we have already compared in chapter 4, we use the default options including all other memory reclamation mechanisms, i.e., ballooning, compression, and swapping. To isolate the performance benefit of memory virtualization techniques, the memory upperbound here only confines the memory used by the VM itself plus all memory used to store compressed pages, but has no limitation on the memory used by the hypervisor, i.e., all the rest of host memory are given to hypervisor in both Xen and VMware ESXi server. For example, the VMware ESXi 5.0 server reserves 188 MB for a configured 2GB VM, referred to as *overhead memory*, to store the code and data structures associated with VM resource management, e.g., VM frame buffer and page tables. When we are setting the upperbound of VM memory, the overhead memory is not taken into account.

6.7.1 Effectiveness of TWS-based Ballooning

Advantage of Probing Working Set Proactively

The TWS-ballooning could have a small amount of performance overhead because of the periodical probe of the working set but it has performance benefit from the earlier detection of working set when there is a sudden change of workload on the physical host. To verify this benefit, we compare the TWS-ballooning and the VMware ESXi 5.0 server on different host machines. On each host, we run the following two VMs with different synthetic workloads.

- **300**: this workload initializes by allocating 300 MB of memory, and then all the memory access uniformly falls in the allocated memory,

[5] VMware ESXi 5.0.0 build-623860.

	Baseline	Self-ballooning	TWS-ballooning
Average latency (sec)	0.37	0.37	0.37
Latency degrade (%)	N / A	0	0
Average balloon target (KB)	N / A	263344	263344
Average saved memory to Self-ballooning (%)	N / A	N / A	0

Table 6.2: Comparison of average latency and balloon target of SPECweb Banking benchmark for Baseline, Self-ballooning, and TWS-ballooning mechanisms.

	Baseline	Self-ballooning	TWS-ballooning
Runtime (sec)	682	710	703
Runtime degrade (%)	N / A	4.11	3.08
Average balloon target (KB)	N / A	922607	783570
Average saved memory to Self-ballooning (%)	N / A	N / A	15.07

Table 6.3: Comparison of runtime and average balloon target of SPEC CPU 401 benchmark for Baseline, Self-ballooning, and TWS-ballooning mechanisms.

	Baseline	Self-ballooning	TWS-ballooning
Runtime (sec)	1487	1755	1537
Runtime degrade (%)	N / A	17.99	3.31
Average balloon target (KB)	N / A	328810	350839
Average saved memory to Self-ballooning (%)	N / A	N / A	-6.70

Table 6.4: Comparison of runtime and average balloon target of OLTP benchmark for Baseline, Self-ballooning, and TWS-ballooning mechanisms.

The memory content we use in the two synthetic workloads has 40% compression ratio to the LZO1X compression algorithm in zram.[6]

- **1200**: this workload initializes by allocating 1200 MB of memory, and then all the memory access uniformly falls in the allocated memory, The memory content we use in the two synthetic workloads has 40% compression ratio to the LZO1X compression algorithm in zram.

The memory upperbound of the two VMs is set to 2 GB, which is enough to accommodate the sum of their working sets including the memory used by the guest OS (**200 MB** in our test VM including system daemons). Before the workloads starts, the balloon target of each VM in the TWS-ballooning is at the lowerbound, i.e., 263344 MB, because there is no workload running. On the other hand, the ESXi server allocates equal amount of memory to each VM in the beginning, i.e., 1 GB. We let the two workloads start at the same time and measure the time consumption for both workloads to reach their baseline throughput.

From the result, our TWS-ballooning takes 10 seconds to reach the baseline because it takes time for zballoond to adjust the balloon target to the Committed_AS and there exist certain amount of working set being evicted out to zram when the workload suddenly grows in the beginning. On the contrary, the ESXi server takes 136 seconds because the working set detection is not done beforehand and it takes longer for the ESXi server to sample the VM working set and adjust the memory allocation accordingly. Thus, the TWS-ballooning has benefit to rapidly react to working set changes between multiple VMs but has to pay a small price on the periodic probe of the working set.

Comparing TWS-ballooning with Self-ballooning

Similar to our TWS-ballooning, the Self-ballooning mechanism also have the benefit mentioned above; however, it is not able to accurately probe the working set depending on `Committed_AS`. Here, we compare our TWS-ballooning against the Self-ballooning mechanism running benchmarks with different kinds of characteristics and see how our mechanism probes the true working set to save more host memory while keeping the application performance. The first of the two performance metrics we concern is the application performance degradation against the **Baseline**, i.e., when the VM is configured with memory size larger than the requirement of benchmark, e.g., here we set the VM

[6]From the experiment results of real workload, the compression ratio observed in zram is about 30 to 40%. Thus, 40% is used in synthetic workload to reflect the characteristics of real workload.

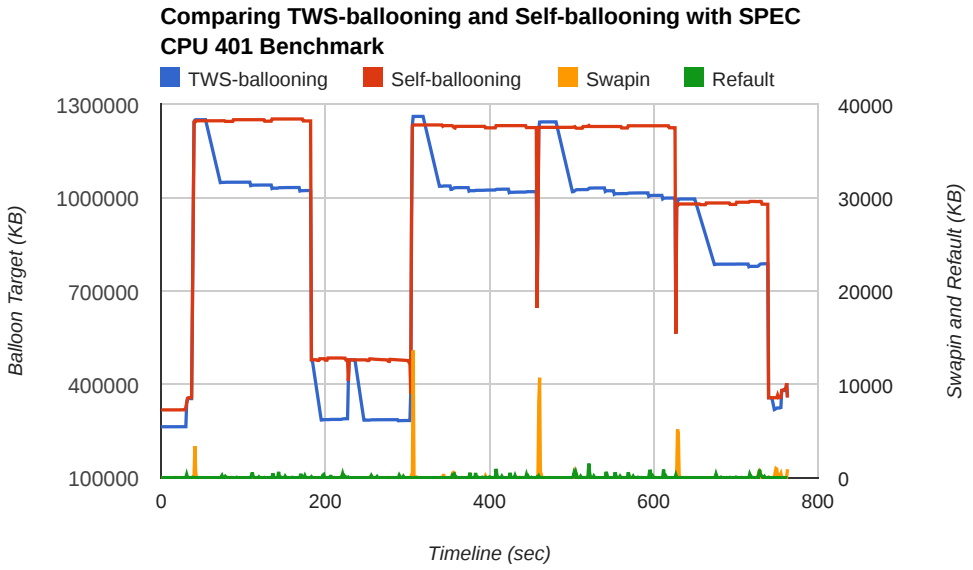

Figure 6.4: *The balloon target, swapin and refault size of Spec CPU 401 workload under TWS-ballooning and Self-ballooning mechanisms.*

memory size of baseline to 2 GB with all our kernel components disabled. The other metric is the amount of saved memory against the Self-ballooning mechanism. While the application performance should be kept without noticeable degradation, we expect the TWS-ballooning to save more host memory so that the hypervisor can use it more efficiently. To perform the comparison, we look at the following workloads.

- SPECweb Banking [65]: SPEC's benchmark to evaluate web server performance. The apache [69] is used to host web server.
- SPEC CPU: a set of SPEC's CPU [70] benchmarks with intensive memory access and allocation of anonymous memory.
- OLTP: an Online transaction processing (OLTP) benchmark from Sysbench [71] test suites to test database server performance, which consumes more page cache from disk access. Here we use MySQL [72] to host the database server.

In Table 6.2, 6.3, and 6.4, we show the performance comparison of these benchmarks running the Self-ballooning and TWS-ballooning mechanisms. For Specweb Banking, its workload is always below the lowerbound, i.e., 263344 KB for our 2GB test VM. Thus, we do not observe any degradation from baseline or saved memory for both Self-ballooning and TWS-ballooning

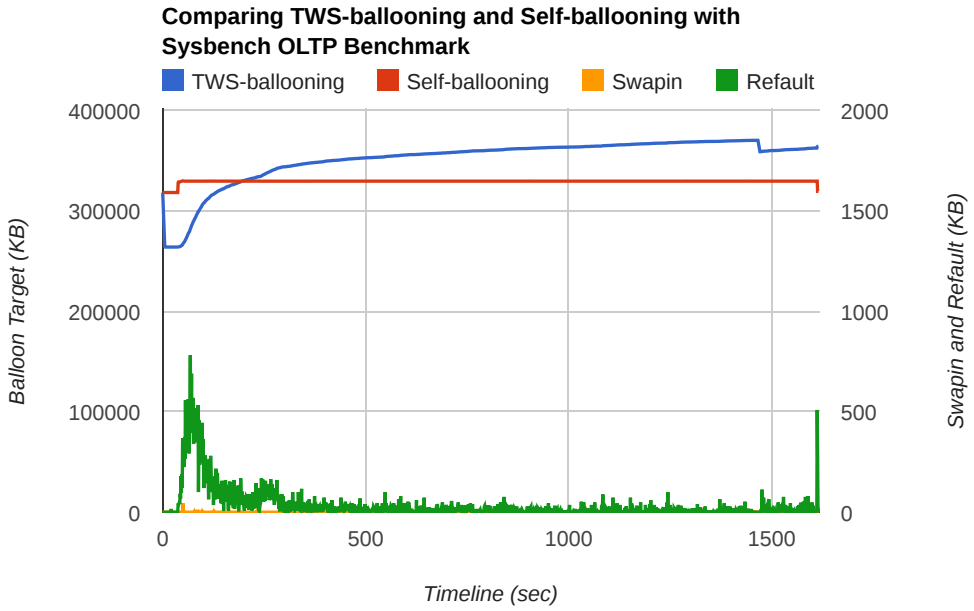

Figure 6.5: *The balloon target, swapin and refault size of Sysbench OLTP benchmark under TWS-ballooning and Self-ballooning mechanisms.*

mechanisms. As for the SPEC CPU 401 benchmark, our TWS-ballooning can further save 15.07% of the memory comparing to the Self-ballooning because we periodically probes down the current working set of the VM as Figure 6.4 shows but the Self-ballooning only watches the Committed_AS, which is not accurate enough for the real working set.

Besides of memory saving, the performance degradation against the baseline is surprisingly reduced from 4.11% of Self-ballooning to 3.08% of TWS-ballooning. One major reason is that the Committed_AS can not reflect the memory demand of burst swapin or refault event. As Figure 6.4 shows, the balloon target of TWS-ballooning has been dynamically adjusted beyond the target of Self-ballooning, i.e., Committed_AS, at certain points of the timeline, which is caused by the sudden growth of swapin and refault event. The performance benefit is not obvious in the SPEC CPU test suites because its benchmarks mostly consume the anonymous memory and can be measured directly by Committed_AS.

But for the OLTP benchmark, the performance benefit will be more obvious because the benchmark has more disk access and thus consumes more page cache, which is outside the scope of Committed_AS. From the result of OLTP benchmark in Table 6.4, the average balloon target of TWS-ballooning is 6.7% higher than the Self-ballooning due to react to the occurred refault events, and thus the TWS-ballooning can reduce the performance degradation against the

baseline from 17.99% of Self-ballooning to 3.31% of TWS-ballooning. From the timeline of balloon target in Figure 6.5, we can observe that the TWS-ballooning detects the refault events and increase the balloon target accordingly so that the performance is more close to the native. As a result, our proposed TWS-ballooning can dynamically adjust the balloon target according to the current working set size and perform better than Self-ballooning for various kinds of workloads discussed in this section.

6.7.2 Effectiveness of TWS-aware Memory Compression

When the host memory is low such that the true working set of the VM has to be evicted out, the memory size used by zram becomes more important for application performance. In this subsection, we evaluate our *dyn_memlimit* mechanism, which dynamically adjusts the zram size or memlimit based on page access probability of zram LRU list mentioned in chapter 6.4. To isolate the performance of dyn_memlimit, we disabled the TWS-ballooning mechanism. The test VM we used here has configured the memory upperbound as 600000 KB (586 MB). The memory used by the workload and the guest OS itself, i.e., 200 MB, has exceeded the upperbound. The questions to answer are (1) whether the *dyn_memlimit* can adjust the memlimit to an optimal size and (2) how well can the *dyn_memlimit* react when the working set changes. The first question is pretty difficult because the optimal size of memlimit changes dynamically when the workload is running and it is impossible to find out from experiment.

Thus, we compare our proposed *dyn_memlimit* mechanism with two other naive implementations: (1) zero_memlimit which consumes no memory in zram by redirecting all swapped-out pages to disk except zero page and (2) max_memlimit which uses as much memory as possible in zram with an upperbound described by Equation 6.2.

Evaluating dyn_memlimit with Synthetic Workloads

We introduce the following two synthetic workloads each has one of the naive implementations as its optimal solution and run for 20 minutes to gather results.

- **700-half**: this workload initializes by allocating 700 MB of memory, and then all the memory access uniformly falls in the first half of the allocated memory, i.e., 350 MB is the true working set. If we number the allocated memory from 0 to 699 with MB as the unit, then the working set we

	zero_memlimit	dyn_memlimit	max_memlimit
Throughput (MB/sec)	8027	8091	74
Zram size (MB)	N / A	0.68	288

Table 6.5: Comparison of average throughput and zram size for zero_memlimit, dyn_memlimit, and max_memlimit running 700-half workload.

are referred to here is number 0 to 349. The memory content we use in the two synthetic workloads has 40% compression ratio to the LZO1X compression algorithm in zram. The working set plus the 200 MB of guest OS, i.e., 550 MB, can just fit into the memory without using zram. Thus, *zero_memlimit* is an best choice among the three mechanisms.

- **95-10**: this workload initializes by allocating 700 MB of memory, and then 95% of the access uniformly falls into the first 10% of the allocated memory while the rest 5% of the access uniformly falls into the rest 90% of the allocate memory. The true working set plus the memory used by guest OS is larger than the total VM memory upperbound, and the ideal case is to put the first 10% of the working set, i.e, 70 MB (referred to as *hot working set*, into memory without compression and compress the rest 630 MB working set (*cold working set*) into 252 MB zram, and all working set plus the 200 MB of guest OS can just fit into the VM memory upperbound. Therefore, *max_memlimit* is the best choice.

Table 6.5 shows the average throughput and zram size of 700-half workload running with the three mechanisms, zero_memlimit, dyn_memlimit, and max_memlimit. As expected, the zero_memlimit gives almost native memory throughput[7] while the max_memlimit suffers because the working set is unnecessarily compressed into zram. Our proposed dyn_memlimit adjusts the memlimit by kicking out all pages in zram to disk, i.e., average zram size is less than 1 MB, and leave free memory space for the true working set. From the table, we have observed that dyn_memlimit performs slightly better than the zero_memlimit by 64 MB/sec. When we look more closely into the experiment logs, we found that during the 20 minutes of experiment, there are some swapin and refault events occasionally. It is due to the system daemons of guest OS, which rise and evict some of the working set. The zero_memlimit has more degradation because evicted working set has to be swapped-in from disk but our dyn_memlimit can use zram to buffer these evicted working set,

[7]The throughput here is **not** the baseline result because the VM memory upperbound is smaller than the workload itself.

	zero_memlimit	dyn_memlimit	max_memlimit
Throughput (MB/sec)	14	619	652
Throughput of hot working set (MB/sec)	6213	2361	2454
Throughput of cold working set (MB/sec)	0.71	42	44
Zram size (MB)	N / A	285	285

Table 6.6: Comparison of average throughput and zram size for zero_memlimit, dyn_memlimit, and max_memlimit running 95-10 workload.

and has small time cost when swapin happens later on. Thus, zram has certain benefit in this case.

Table 6.6 shows the average throughput of 95-10 workload running with the three mechanisms, zero_memlimit, dyn_memlimit, and max_memlimit. Besides for the average throughput, we also measure the throughput of hot working set and cold working set. As expected, max_memlimit gives the best memory throughput while zero_memlimit suffers. The zero_memlimit forbids using zram to store the working set so that the hot working set can be kept mostly in memory and achieve higher throughput, i.e, 6213 MB/sec. Note that this throughput can not be as high as the throughput shown in 700-half experiment, i.e., an extreme case with 50% of the memory completely idle, because the Linux LRU mechanism can not perform 100% accurately by keeping all hot working set in memory. Sometimes, the guest OS may still evict out the pages of hot working set. However, the overall throughput of zero_memlimit suffers because the throughput of cold working set degrades severely upon the disk swapin.

As for our proposed dyn_memlimit mechanism, it adjusts the memlimit to achieve 95% of the throughput of max_memlimit. The 5% degradation could be due to occasional eviction of zram because of the inaccurate estimation of page access probability or overhead_time. The average zram size for both dyn_memlimit and max_memlimit is 285 MB, which is 33 MB higher than our hypothetical 252 MB to compress the 630 MB cold working set. For the extra size of zram, there should exist a portion of the guest OS memory or very few hot working set; otherwise the throughput of the hot working set can not go up to GBs. In short, the dyn_memlimit can adjust the memlimit well in both of the two synthetic workloads.

To see how our dyn_memlimit react when the working set changes, we modify the 700-half workload by shifting the working set 10 seconds after the

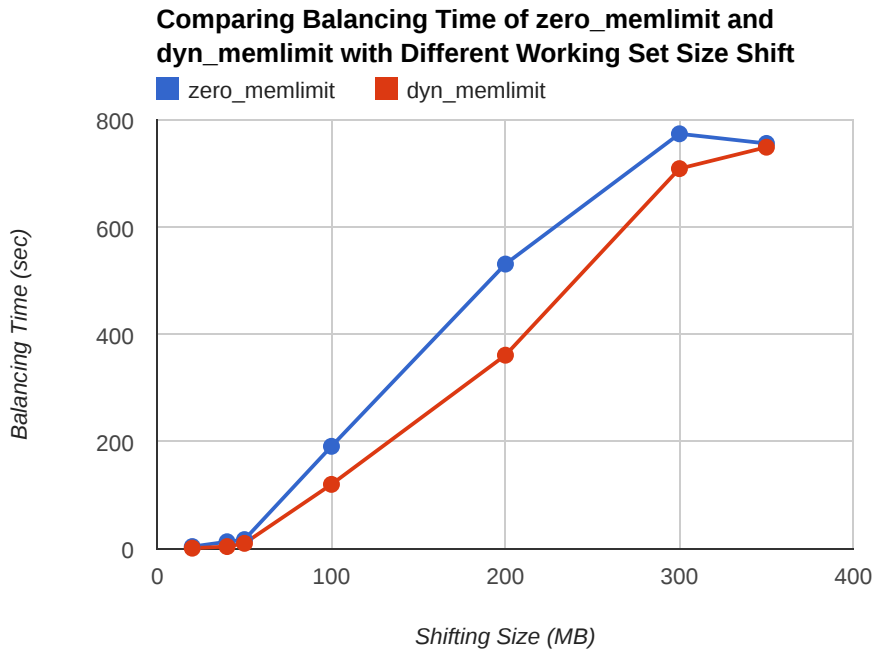

Figure 6.6: *Balancing time comparison of zero_memlimit and dyn_memlimit running 700-half workload with various shifting size of the working set.*

workload has initialized. Note that the new working set is all evicted to disk after the initialization, and it takes time for the exchange the old and new working sets via disk swapin and swapout, referred to as *balancing time*. If our dyn_memlimit can not react fast enough to evict old working set to disk, it will occupy memory space and affect the performance of the new working set. We compare the balancing time of our dyn_memlimit with the zero_memlimit in Figure 6.6 by varying the shifting size of working set. The balancing time increases as the shifting size increases for both of the mechanisms. As the figure shows, the zero_memlimit shows the ideal performance and our dyn_memlimit does not add more overhead to it but further improves the balancing time.

To analyze the reason, we explore the detailed statistics of dyn_memlimit in figure 6.7 when the shifting size is equal to 300 MB. From the figure, the memory throughput, directly bound by the rate of swapin at this moment, keeps climbing from the start of the experiment and eventually to the native throughput. The size of zram, zram_size in the figure, keeps increasing without dropping down before the 600th second. This is because the memory pages residing in zram are mostly cold pages belonging to the old working set, and the difference of page access probability in zram_lru is obvious, and thus the dyn_memlimit does not actively kick cold pages to disk. After 600 second, the new working set is almost swapped-in from disk and the memory space is not

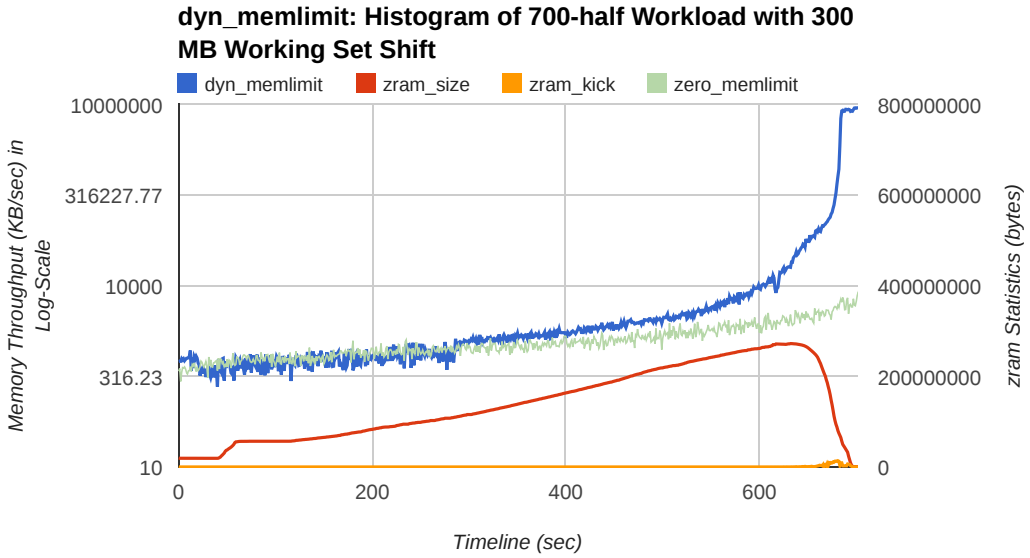

Figure 6.7: *Histogram of memory throughput and zram statistics running 700-half workload and 300 MB shift of working set size with the dyn_memlimit mechanism. Using left axis, the dyn_memlimit line shows the throughput of the workload where the zero_memlimit line is taken from other experiment for comparison purpose. With right axis, the zram_size and zram_kick shows the size of zram and the kicked size of cold memory in bytes.*

enough for both the new working set and the compressed pages of old working set. Thus, some of the hot pages may be evicted into zram, and it triggers the zram to kick out cold pages where the zram_kick shows the number of bytes being kicked to disk per second.

The zero_memlimit line shows the throughput result chopped from other experiment and increases slower than the dyn_memlimit. We consider the reason is because the overlapped operations of swapin and swapout can reduce the performance for each of them. In the case of zero_memlimit, the swapin of new working set and swap-out of old working set have to be performed at the same time, and thus reduce the performance of both disk and swap subsystem. On the contrary, the dyn_memlimit buffered the old working set in zram without sending to disk and let the throughput (swapin rate) climbs faster. As a result, using the zram with dynamically adjusted size does not add extra overhead when the working set changes, but brings extra benefit to it.

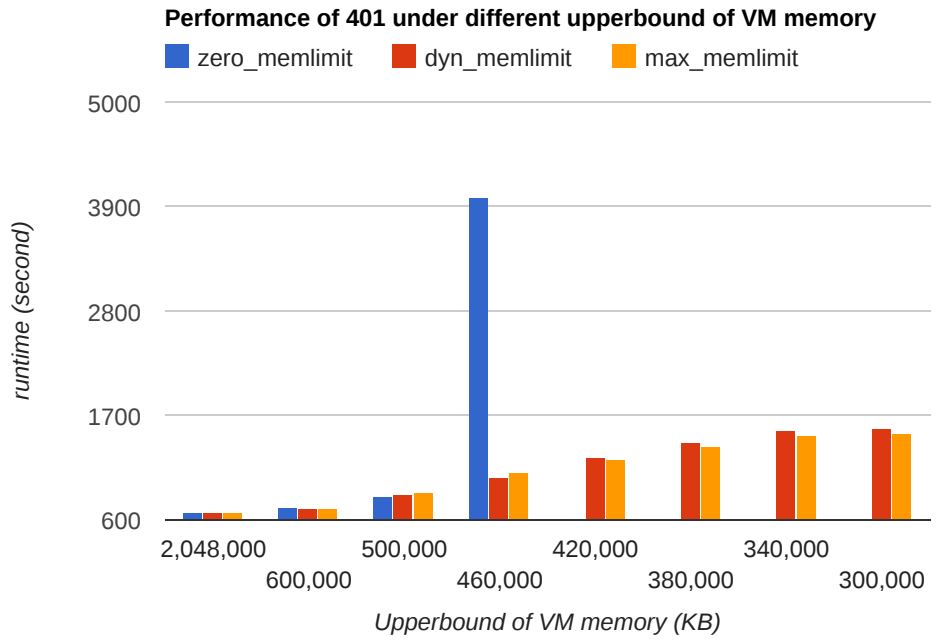

Figure 6.8: *Runtime performance of SPEC CPU 401 bzip2 running zero_memlimit, dyn_memlimit, and max_memlimit with various kinds of VM memory upperbound. The upperbound 2 GB shows the baseline performance.*

Evaluating dyn_memlimit with Real Benchmarks

After the analysis of synthetic workload, we now look into real benchmark and compare the three mechanisms. Figure 6.8 and 6.9 show the runtime performance of the three mechanisms running the two memory intensive benchmarks, SPEC CPU 401 and 481 benchmarks. Here, we vary the memory upperbound of VM, and see how each mechanism performs. For each of the two figures, we put the baseline performance result on the left for comparison, i.e., the upperbound is equal to 2 GB where TWS-ballooning is also disabled. The runtime degrades as the upperbound decreases because of more memory pressure on the benchmark. The results show that the 401 runs better when using the zram but has poor performance when applying the zero_memlimit mechanism. When the upperbound is reduced down to 460,000 KB, the runtime of zero_memlimit is 4 times slower than the rest mechanisms. When the upperbound is smaller than 460,000 KB, we do not show the result of zero_memlimit because the benchmark can not finish within 2 hours. On the contrary, the benchmark 481 runs better when applying zero_memlimit. In both cases, the dyn_memlimit adapts well and shows similar performance to the best mechanism. Especially for the 481 benchmark when the upperbound changes to

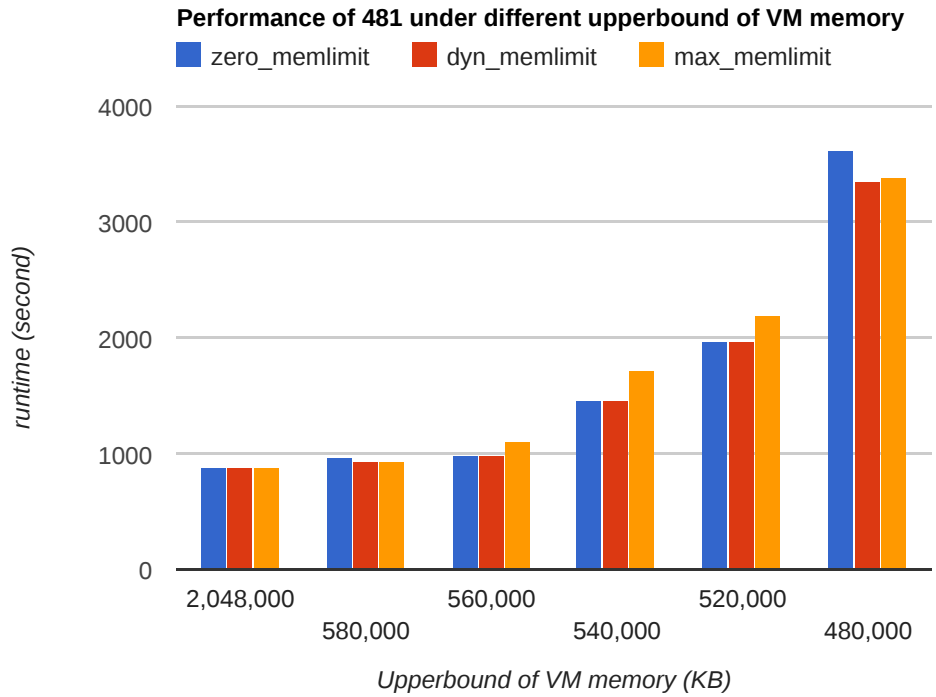

Figure 6.9: *Runtime performance of SPEC CPU 481 wrf running zero_memlimit, dyn_memlimit, and max_memlimit with various kinds of VM memory upperbound. The upperbound 2 GB shows the baseline performance.*

480,000 KB, the zero_memlimit becomes worse than max_memlimit but our dyn_memlimit adapts well and performs the same as the max_memlimit. Similar to the results from synthetic workloads, we have shown that that our dyn_memlimit can also adapt well in real benchmarks.

Comparing the Memory Virtualization Ratio with VMware ESXi Server

Recalling our ultimate goal is to increase the memory utilization or the *memory virtualization ratio*, our TWS-ballooning and dyn_memlimit are shown to squeeze the VM memory below its working set size while maintaining the application performance. Taking the 401 benchmark in Figure 6.4 as an example, it consumes average 1 GB memory while running Self-ballooning mechanism, i.e., the memory size is equal to the Committed_AS plus the memory of guest OS while our TWS-ballooning further suppress it to 830 MB with 3% performance degradation comparing to baseline. As the result of Figure 6.8 shows, With the help of dyn_memlimit, the upperbound of VM memory can further go down to 586 MB (600,000 KB), i.e., around **60%** of the memory size when

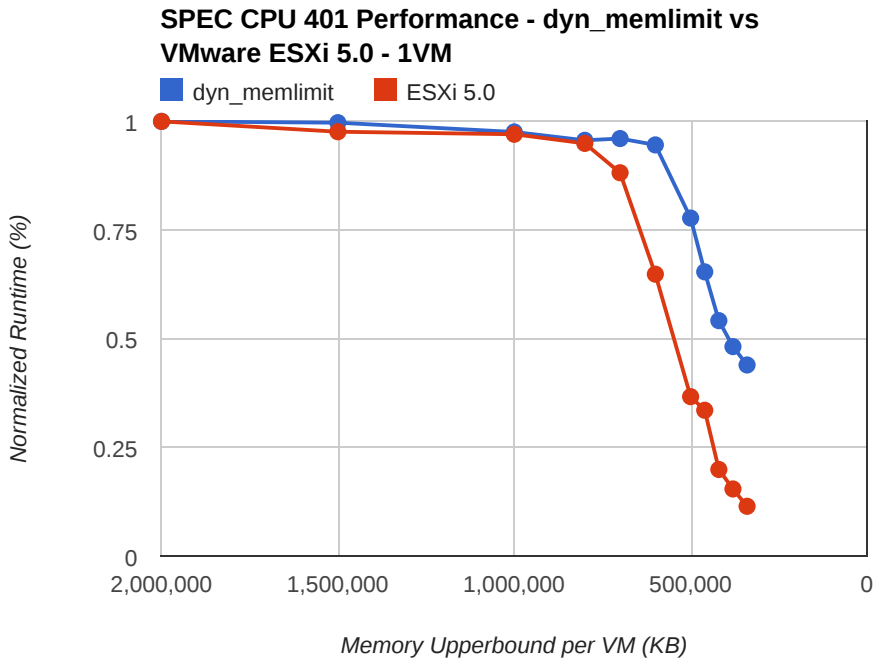

Figure 6.10: *Normalized runtime performance of SPEC CPU 401 bzip2 on 1 VM running dyn_memlimit and VMware ESXi 5.0 with various kinds of upperbound of balloon target.*

the VM is running Self-ballooning, while the performance downgrade is bound at 5.5%.

To verify the effectiveness of our work in the real world, we compare the memory virtualization ratio of our dyn_memlimit with the VMware ESXi 5.0 server. As Figure 6.10 shows, we examine the normalized runtime of the 401 benchmark with dyn_memlimit on Xen hypervisor and the entire suite of reclamation mechanisms on VMware ESXi 5.0 server by lowering down the VM memory upperbound from 2,000,000 KB to 300,000 KB. The performance is normalized to the baseline 682 and 657 seconds for dyn_memlimit and ESXi server respectively, i.e., divide the baseline runtime by the corresponding runtime result. From the figure, the normalized performance of the ESXi server starts to degrade when the memory upperbound is at 800,000 KB while our dyn_memlimit starts to drop at 600,000 KB. When the memory upperbound is at 600,000 KB, the normalized performance of ESXi server has dropped to 65% while our dyn_memlimit keeps at 96%.

As for the 481 workload shown in Figure 6.11, the dyn_memlimit keeps the normalized performance at 91% when the memory upperbound is 560,000 KB while the performance of ESXi server has dropped to 70%. However, our

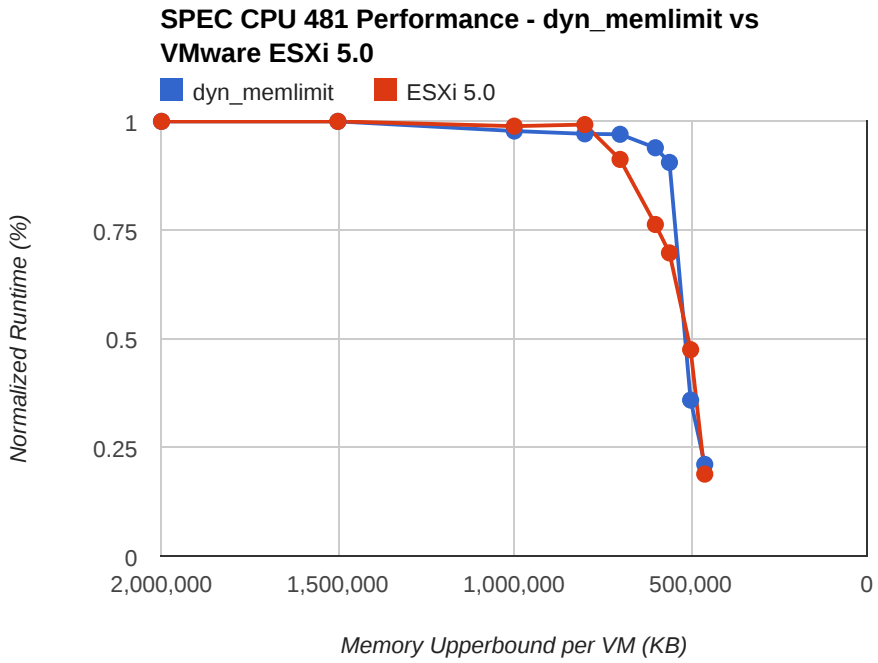

Figure 6.11: *Normalized runtime performance of SPEC CPU 481 wrf running dyn_memlimit and VMware ESXi 5.0 with various kinds of upperbound of balloon target.*

dyn_memlimit does not have too much performance gain against the ESXi server when the memory upperbound keeps decreasing. This is because the 481 workload does not favor zram as we can observe from Figure 6.9 where the zram is kept almost empty and has performance close to the zero_memlimit.

To analyze the performance when multiple VMs are running, we change the experiment of 401 benchmark from 1 VM to 4 VMs where each VM runs the same 401 benchmark. Note that we do not run more than 4 VMs because there only exists 4 CPU cores on each test machine and each VM's virtual CPU is pinned to distinct physical CPU to isolate the performance. In addition, the memory upperbound is equalized among all VMs so that memory balancing mechanism is not necessary. The result is shown in Figure 6.12 where the x-axis shows the memory upperbound per VM.

As the figure shows, the normalized performance of both the dyn_memlimit and ESXi server has degraded comparing to the result of single VM. From the experiment results, the average baseline performance of the 4 VMs has dropped to 748 second and 721 second, i.e., about 10% comparing to the experiment of 1 VM, for the dyn_memlimit and ESXi server respectively. Our explanation is that the hardware resource is multiplexed among VMs especially for (1)

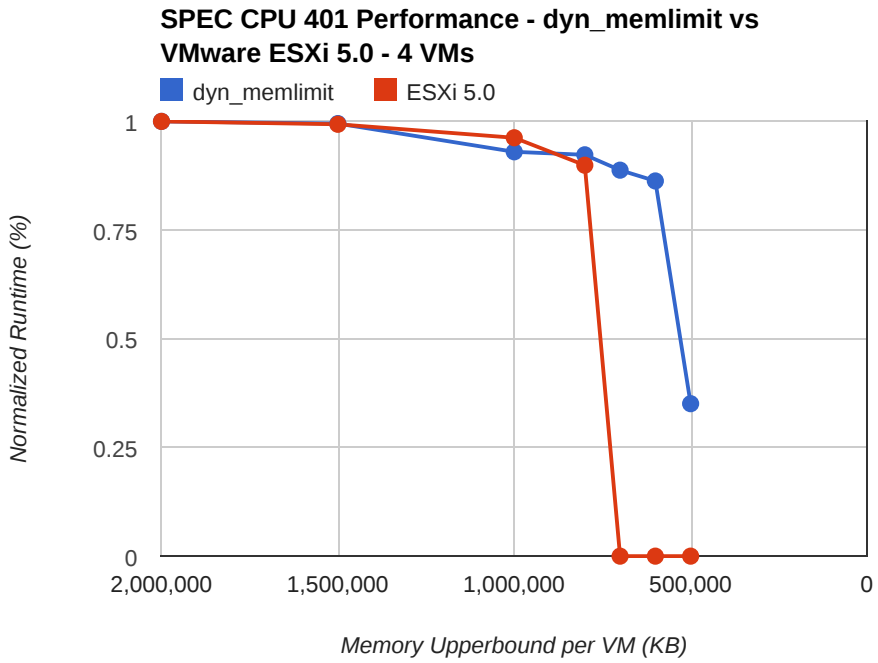

Figure 6.12: *Normalized runtime performance of SPEC CPU 401 bzip2 on 4 VMs running dyn_memlimit and VMware ESXi 5.0 with varied memory upperbound per VM. Note that the memory upperbound is equalized among the 4 VMs.*

memory bandwidth because the 4 VMs are sharing certain memory banks and (2) disk I/O bandwidth because the 4 VMs performing swapin and swapout on the same disk.

While the normalized throughput of ESXi server is at 90% when the memory upperbound per VM is 800,000 KB, all guest OSs panic when the upperbound is dropped to 700,000 KB or less. From the reported message of guest OS, some process and regular disk I/O request have hung for more than 120 seconds. This is possibly due to the heavy disk swapin/swapout inside the guest OS, which slows down the process running and normal disk I/O significantly. On the contrary, our dyn_memlimit has zram running inside each VM to absorb the guest memory pressure and keeps swapin/swapout performance. Thus, the normalized throughput of dyn_memlimit still keeps at 86% when the upperbound drops to 600,000 KB. When the upperbound goes down to 500,000 KB, even the normalized performance is dropped to 35%, but the guest OS is still running properly instead of crashing as the ESXi server.

Workload \ Mechanism	dyn_memlimit	VMware ESXi
700-half	95.12%	0.85%
95-10	7.98%	0.99%

Table 6.7: Comparing the normalized average throughput of dyn_memlimit and VMware ESXi server running the 700-half and 95-10 workload where the memory upperbound is 600,000KB.

Analyzing Performance Bottleneck of VMware ESXi Server

There could be two problems for the compression mechanism of VMware ESXi server.

- By default, the maximum compression cache of VMware is arbitrarily limited to 10% of the VM memory size. In the current experiment, the size is 204 MB for the configured 2GB of VM memory size. We consider the reason is that VMware currently does not have a good mechanism to adjust the memory size of compressed memory so they choose the percentage from experience or experiment results.
- The VMware does not carefully choose the pages for compression, but randomly pickup the candidate page in the guest address space, which can easily touches the pages of the true working set.

On the contrary, we carefully choose the candidate page outside of the working set and adjust the zram memory size dynamically so that the VM performance is kept better.

To analyze the first problem of ESXi server mentioned above, we apply the 700-half and 95-10 synthetic workloads from the previous section to both the dyn_memlimit and ESXi server. The result in Table 6.7[8] shows the average throughput of 700-half and 95-10 *normalized* to baseline, i.e., divide the average throughput by the corresponding baseline result, for dyn_memlimit and the ESXi server respectively. While dyn_memlimit maintains the baseline throughput in 700-half and 7.98% (619 MB/sec) in 95-10, the throughput of ESXi server is low in both cases, i.e., 0.85% (70 MB/sec) and 0.99% (76 MB/sec) for 700-half and 95-10 respectively. As we observe the system statistics of the ESXi server via the console, the compressed memory is always kept at 10% for both workloads.

[8]The results of dyn_memlimit are borrowed from Table 6.5 and 6.6.

	dyn_memlimit	VMware ESXi
Normalized throughput	39.34%	1.47%
Normalized throughput of hot working set	93.16%	50.81%
Normalized throughput of cold working set	3.46%	0.09%

Table 6.8: Normalized average throughput for dyn_memlimit and VMware ESXi server running 95-10 workload where the memory content is zero page and the VM memory upperbound is 700,000 KB.

For the 700-half workload, we expect the compressed memory to be almost zero, but the ESXi server consumes 10% of the VM memory, i.e., 204MB, for compression and thus the working set can not fully reside in memory. As for the 95-10 workload, our dyn_memlimit uses 285 MB for compressed memory, i.e., at least 252 MB to accommodate the cold working, referring to the discussion in chapter 6.7.2, but the ESXi server has only 204 MB maximally, which is insufficient in this case.

To analyze the second problem of VMware ESXi server, we use 95-10 workload with the following two changes.

- Memory content: change the memory content of the workload from 40% compression ratio to zero page[9] because here we only want to analyze the penalty of randomly selecting compressed pages. But if we use 40% compression ratio as the memory content, the compressed memory will occupy VM memory space and further trigger the dynamic zram adjustment in dyn_memlimit and host swapping in VMware ESXi server, which is difficult to isolate the performance problem.

- Memory upperbound: change the VM memory upperbound to 700,000 KB so that the hot working set can be squeezed less and performs better. In this case, if the compressed pages is randomly chosen, the performance hit on the hot working set will be more obvious and easy to observe.

As the result in Table 6.8, our dyn_memlimit shows the normalized throughput of hot working set, i.e., 93.16%, can be very close to the baseline but the ESXi server can only achieve 50.81%. This is because the dyn_memlimit relies on the Linux guest OS to evict cold pages for compression but not random selection

[9]The experiment result shows that VMware does use zero page optimization in compression.

as the ESXi server. The difference of cold working set is not obvious because the working set is larger than the given VM memory upperbound and the cold working set is evicted severely. Finally, the overall throughput also reflects the penalty of random selection mechanism in the ESXi server. In short, the dyn_memlimit utilizes the memory compression mechanism to the maximum extent because it leverages the guest OS reclamation mechanism and carefully adjusts the zram size.

6.7.3 Effectiveness of Memory Balancing

Given VMs with different working set size, we evaluate our proposed memory balancing mechanisms by comparing the difference of the performance overhead of each VM. For simplicity in this section, when we refer to the name of workload or benchmark, we refer to one VM with the corresponding workload or benchmark running on top of it. First, we use the following configuration of two VMs, referred to as **400-700**, each running different synthetic workload to analyze the characteristics of each balancing mechanism.

- **400**: the synthetic workload running inside the VM allocates 400 MB memory as its working set with memory access normally distributed among the working set where the standard deviation is equal to the working set size divided by three. Here, each memory access zeros out the page content with `memset`.

- **700**: the synthetic workload running inside the VM allocates 700 MB memory as its working set with memory access normally distributed among the working set where the standard deviation is equal to the working set size divided by three. Here, each memory access zeros out the page content with `memset`.

Each of the workload is given a target amount of memory to access, and stops execution when the amount of memory is reached. Therefore, if the two workloads suffer from the same performance overhead, both of them should stop at the same time, i.e., have the same runtime degradation. Because each workload may have different runtime in reality, we force the memory resource contention among all VMs to start in the beginning of experiment and ends when the workload with the shortest runtime stops, i.e., after the shortest benchmark stops, its released memory is enough for all the rest VMs to run natively without memory balancer involved.

For a fixed configuration of VMs and memory balancing mechanism, we gather the following performance statistics from each VM.

- **Normalized Runtime Degradation**: find the runtime degradation in seconds against the baseline,[10] referred to as $T_{degrade}$, normalize it by the runtime of the shortest benchmark, $T_{shortest}$, as the following equation,

$$Normalized\ runtime\ degradation\ (\%) = \frac{T_{degrade} \times 100}{T_{shortest}} \tag{6.6}$$

  One can reason this as the runtime degradation percentage for each second during the $T_{shortest}$ period.

- **Average overhead_count**: for the duration of $T_{shortest}$, report the average overhead_count per second.

- **Average overhead_time**: for the duration of $T_{shortest}$, report the average overhead_time per second.

After gathering the three kinds of statistics of all VMs, for each kind of statistics, we report their standard deviation in order to see how each VM differs from each other. For the standard deviation of each statistics, we refer them as ***S_degrade***, ***S_count***, and ***S_time*** for normalized runtime degradation, average over_count, and average overhead_time respectively. Because our ultimate goal is to equalize the overhead of all VMs, we use the *S_degrade* as the performance metric and expect it to be as small as possible so that the overheads of VMs are close to each other and equalized.

As for *S_count*, we expect the Bal_count mechanism to have the smallest number of it because its goal is to balance the overhead_count. On the other hand, the Bal_prop tries to let the VM with larger working set to suffer more, and thus its *S_count* should be larger than Bal_count mechanism. For the *S_time* statistics, because our Bal_time mechanism use the overhead_time to approximate the true overhead of VM performance, we expect to see the *S_degrade* to go up/down with the same trend as *S_time*, and the Bal_time to have the smallest value of it.

In Figure 6.13, we compare the *S_degrade* of our four proposed memory balancing mechanisms using the 400-700 VM configuration while the memory upperbound of all VMs is varied from 1700 MB to 1000 MB. The corresponding results of *S_count* and *S_time* are shown in Figure 6.14 and 6.15 respectively. Looking at the results of *S_degrade* and *S_time*, if we fix the memory upperbound, then the *S_degrade* of each mechanism has the same trend as *S_time*, i.e., if a mechanism has smaller *S_time* then its *S_degrade* is smaller, and thus achieves better result in terms of overhead balancing.

[10] It is gathered when all workloads run with 2 GB and zram disabled.

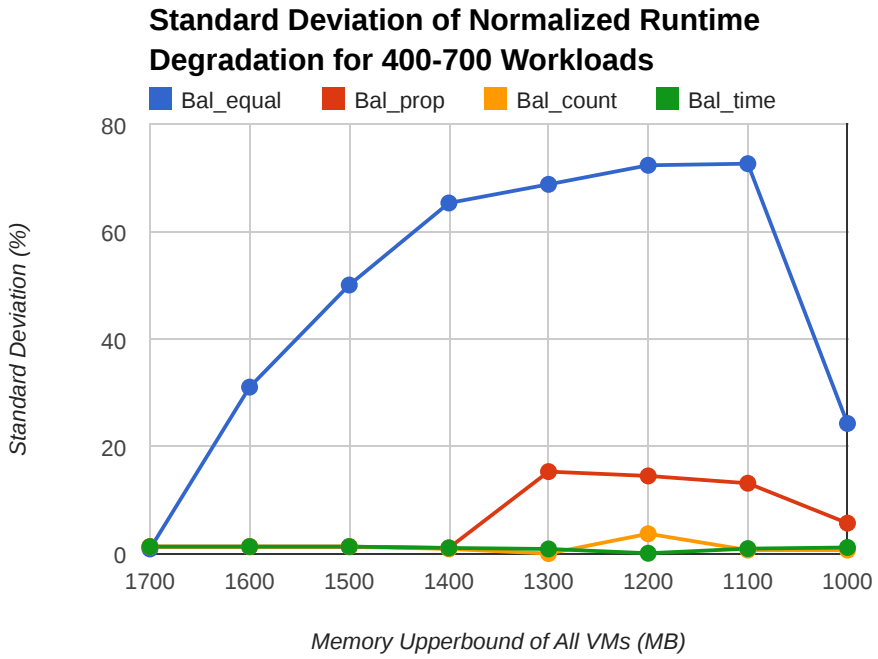

Figure 6.13: *Standard deviation of normalized runtime degradation of different memory balancing mechanisms, Bal_equal, Bal_prop, Bal_count, and Bal_time with 400-700 VM configuration. The memory upperbound of all VMs is varied from 1700 to 1000 MB.*

As for the result of *S_count*, it is completely contributed by swapin but not refault events because the workload does not contain disk access. In addition, the result of *S_count* has very similar trend as *S_time*. We consider it is because the synthetic workloads contain all zero pages, and the time cost of each swapin is zero-optimized, which is small and similar among all VMs. Thus, for our synthetic workloads, if the overhead_count is balanced (smaller *S_count*), then the overhead_time is likely to be balanced (smaller *S_time*), and so is the runtime degradation (smaller *S_degrade*). One exception is when the memory upperbound is 1200 MB, the Bal_count mechanism has the smallest *S_count* but higher *S_time* result than Bal_time, and thus Bal_count performs worse than Bal_time in terms of *S_degrade*. This is also an evidence to show that the overhead_time is a better indicator to measure the VM performance overhead and guide the memory balancing mechanism.

Now we put focus on the comparison of different balancing mechanisms. When applying the Bal_equal mechanism, the workload 400 can enjoy the native throughput when the memory upperbound is above 1100 MB because it consumes around 600 MB of memory, i.e., 400 MB workload plus 200 MB guest OS, and can fit into half of the memory upperbound. However, the

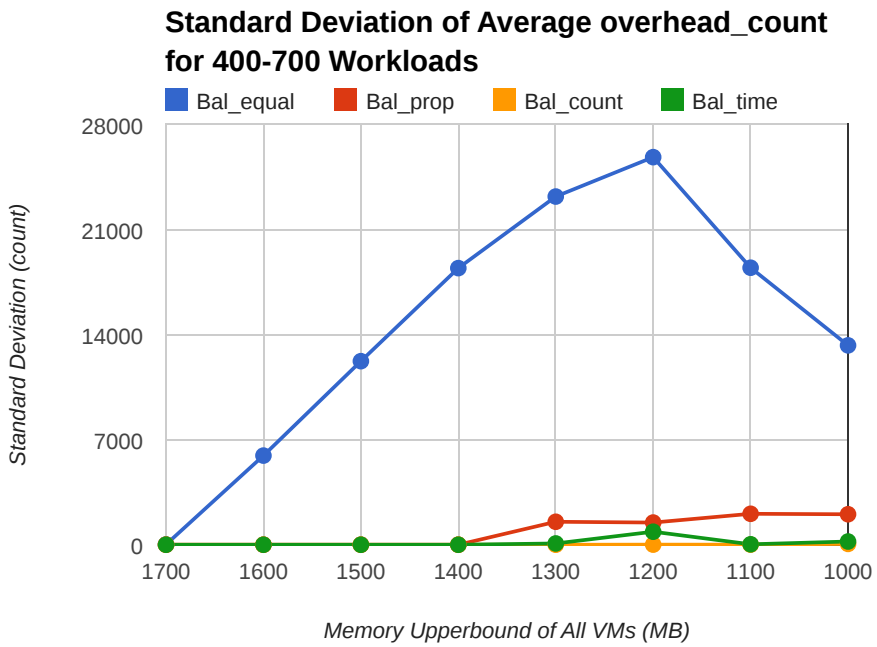

Figure 6.14: *Standard deviation of average overhead_count of different memory balancing mechanisms, Bal_equal, Bal_prop, Bal_count, and Bal_time with 400-700 VM configuration. The memory upperbound of all VMs is varied from 1700 to 1000 MB.*

workload 700 degrades severely as the memory upperbound decreases, and thus the performance gap between workload 400 and 700 keeps increasing as observed from the *S_degrade* results. After the memory upperbound drops below 1100 MB, the working set of workload 400 begins to be eaten, and the *S_degrade* becomes smaller. Because this mechanism does not consider the working set size of VMs, its statistics are the largest among all mechanisms.

As for the Bal_prop mechanism, it tries to give more overhead_count to workload 700 than workload 400 while Bal_count tries to equalize it. With the assumption that *S_count* and *S_time* have similar trend, the *S_degrade* of Bal_count is smaller than Bal_prop and thus achieve better result of overhead balancing. Finally, the Bal_time aims to equalize the overhead_time among all VMs and thus has the best result of *S_degrade* among all mechanisms.

In the following, we further analyze the balancing mechanisms on real workloads with the following configurations. Here, we do not concern about the Bal_equal mechanism but focus on the comparison of other three ones with each set of VM configurations.

- **470-481**: two VMs each running the SPEC CPU benchmarks 470 and 481 respectively. The memory upperbound of all VMs is fixed at 1100

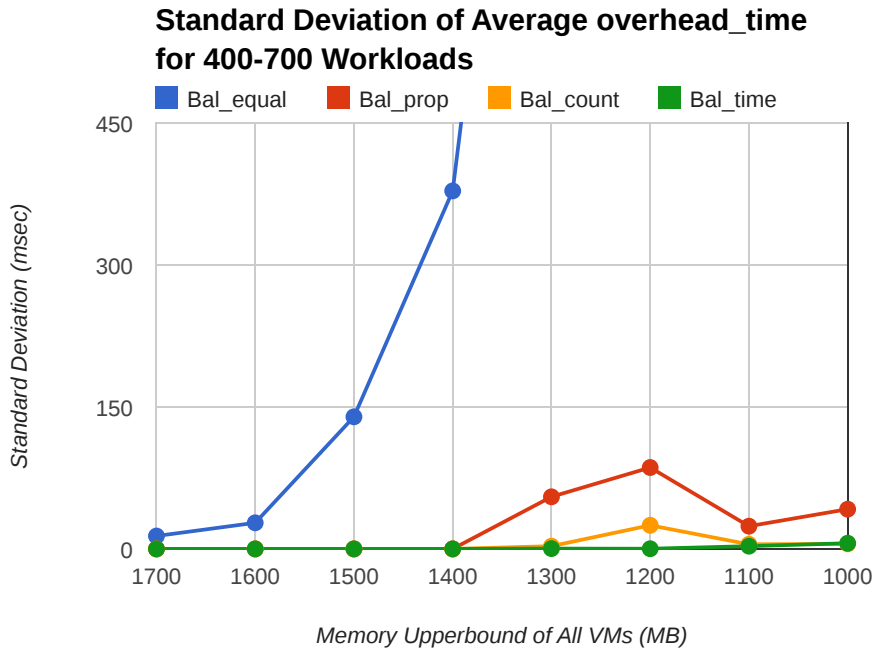

Figure 6.15: *Standard deviation of average overhead_time of different memory balancing mechanisms, Bal_equal, Bal_prop, Bal_count, and Bal_time with 400-700 VM configuration. The memory upperbound of all VMs is varied from 1700 to 1000 MB. Note that the result of Bal_equal is too large to fit into the figure after the memory upperbound drops to 1300 MB, and thus is omitted.*

MB which is smaller the sum of the true working set of benchmark 470 and 481..

- **436-459-470-481**: four VMs each running SPEC CPU benchmarks 436, 459, 470, and 481 respectively. The memory upperbound of all VMs is fixed at 2600 MB which is smaller the sum of the true working set of all four benchmarks.

Similar to the synthetic workloads, we apply the same experiment methodology to gather the three performance statistics described above. In Figure 6.16 and 6.17, we show the performance statistics for the two VM configurations, 470-481 and 436-459-470-481, respectively. From the results, the S_time still holds the same trend as $S\_degrade$, and Bal_time performs the best as it has the smallest S_time among all three mechanisms. On the other hand, the Bal_count mechanism does the best job for balancing the overhead_count, and thus has the smallest S_count among all mechanisms. However, even if the overhead_count of Bal_count mechanism is balanced, its S_time is comparatively larger, and performs the same as the S_prop because these two

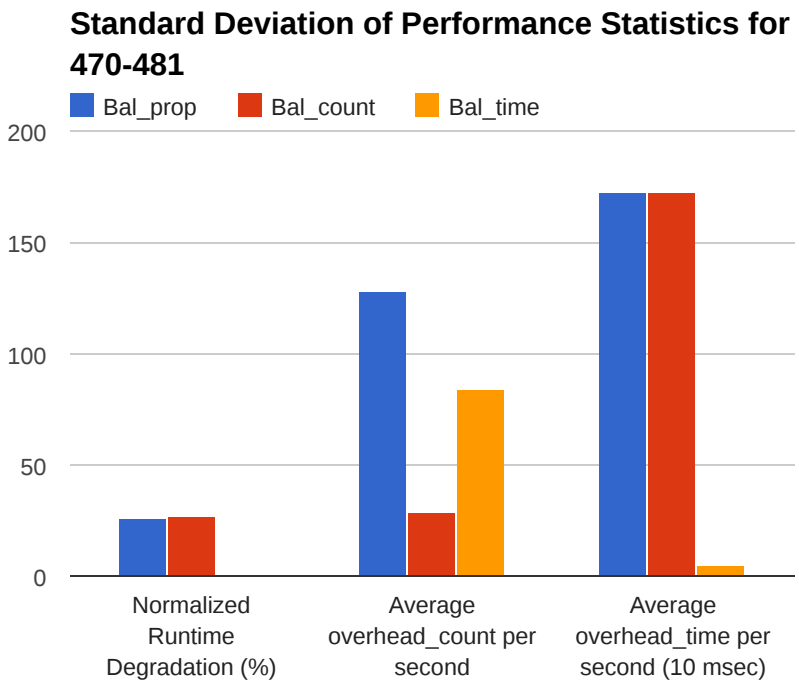

Figure 6.16: *The standard deviation of the all performance statistics of the 470-481 VM configuration Each group of bars represents one statistics and then the three bars within the group represents the results of the three memory balancing mechanisms Bal_prop, Bal_count, and Bal_time respectively.*

mechanisms have similar *S_time* among the two sets of VM configurations.

In short, to balance the VM performance overhead, it is not wise to omit the working set size of VM as Bal_equal and one has to take it into account the Bal_prop mechanism. However, because the performance overhead is driven by the swapin and refault events, the Bal_count can perform better by balancing the overhead_count of VMs. Furthermore, when the time cost of each swapin and refault differs among different VM workloads, it is more important to take the overhead_time into account as the Bal_time mechanism, which is proved from the above results of both synthetic workloads and real benchmarks.

6.7.4 Potential Problem of the Memory Balancing Mechanism

When the memory allocation to VM is comparatively lower than its true working set, the swapin or refault operation will be slowed down due to internal memory pressure of guest. It is possible that the swapin and refault are too slow and can not reflect the true working set of VM. For example, if the VM can

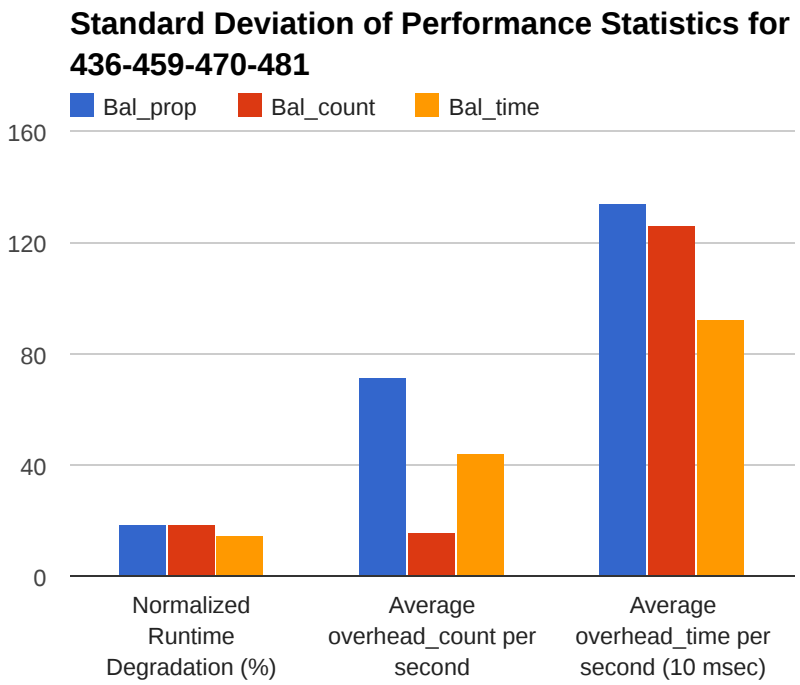

Figure 6.17: *The standard deviation of the all performance statistics of the 436-459-470-481 VM configuration Each group of bars represents one statistics and then the three bars within the group represents the results of the three memory balancing mechanisms Bal_prop, Bal_count, and Bal_time respectively.*

access its deficient part of memory for R MB/sec, the real overhead_count may not be fast enough to react and thus we may gather some value smaller than R, and thus under-estimate the VM working set. One symptom of this scenario is that the CPU utilization of the VM is pretty low at this moment because most of the CPU cycles are consumed in swapin and refault operations. To estimate the working set correctly, one should also take into account the CPU utilization inside the guest. For the Bal_time mechanism, the overhead_time of each VM should be weighted according to the guest CPU utilization, i.e., divide the overhead_time by the guest CPU utilization percentage. Similarly, the overhead_count should also be weighted in the Bal_count mechanism. This issue is an interesting topic to be explored in the future work.

6.8 Applying Memory Compression to Windows Guest

In this section, we discuss how to apply the memory compression technique to the Windows guest OS. In Windows OS, the memory allocated using generic guest API, not including the disk I/O operation, corresponds to the anonymous memory in Linux. Similar to Linux reclamation, the Windows OS also uses the hardware access bit to look for pages outside of VM working set and evicts idle pages [73]. To apply the memory compression technique to the Windows OS, we have to implement the *zram* and *zballoond* components for it.

First, we could implement the zram component and provide it as faked disk device inside the Windows guest. One small difference is that the Windows OS uses the swap file instead of swap disk for swap in/out operations. To complete the goal, we can simply store the swap file into the zram disk, so that every file operation to the swap file is intercepted by the zram component. While the swapin statistics can be gathered in the zram component, the refault statistics needs to be collected by intercepting the regular disk I/O operations, which can be done by modifying the disk driver in guest. However, if there is no open source for the disk driver, we have to intercept the operation in the hypervisor context.[11]

As for the zballoond component, same as Linux guest, there already exists open-source balloon driver for windows VM. Based on the balloon driver, we can try to implement the zballoond component to automatically adjust the balloon target of Windows guest, i.e., the TWS-ballooning. However, one missing thing for Windows OS is that there is no explicit kernel parameter such as Committed_AS to initialize the balloon target and detect the big change of working set. One possible solution is to write a system program to collect the memory usage information of all processes, e.g., using the `GetProcessMemoryInfo` Windows API and iterating all processes.

6.9 Summary

The memory compression is often used as the secondary resort to increase memory utilization in the virtualized environment mostly due to its high overhead of decompression when guest VMs access the compressed pages, i.e., when COA mechanism is triggered. In this chapter, we have leveraged the memory reclamation mechanism in Linux guest OS to find out cold pages, i.e.,

[11] The I/O operations are intercepted by hypervisor and mostly handled by the privileged VM, dom0.

pages outside of the working set, as candidates for compression.

Comparing to the on-demand working set detection of VMware ESXi server, the periodic working set probing of TWS-ballooning can react better to the sudden workload changes among VMs but with small amount of performance overhead as the trade-off. As to compare with the Self-ballooning mechanism, the TWS-ballooning is able to probe the *true working set* and reclaim unnecessary guest cold pages back to the hypervisor memory pool, e.g., we have saved 15% more memory for the SPEC CPU 401 benchmark with 1% improvement, i.e., from 4.11% to 3.08%, on the performance degradation against the baseline. Beyond the Committed_AS used in Self-ballooning, the TWS-ballooning can detect the working set of disk-intensive workload which consumes more page cache, e.g., the performance degradation of OLTP benchmark has been reduced from 17.99% of Self-ballooning to 3.31%.

While the TWS-ballooning split the cold working set out into the zram, the dyn_memlimit is further used to dynamically adjust the zram size by kicking out the cold pages in zram to the swap disk. This is especially important when the host memory goes lower than the true working set of VMs and dyn_memlimit adjusts the zram size so that the application performance is kept. From the experiment result, we are able to pack the 401 benchmark into 586 MB (600,000 KB) of VM memory with 5.5% performance degradation whereas the VMware ESXi 5.0 server requires 781 MB (800,000 KB) of VM memory to achieve the same performance, which shows our improvement on the memory virtualization ratio.

Finally, when the host memory can not support the true working sets of all VMs, we balance the performance overhead of each VM so that they can degrade gracefully. We have found that the *overhead_time* of each VM is an accurate estimation of the VM performance overhead, and can be used well on the memory balancing mechanism. From the experiment results, we have shown that the Bal_time mechanism has achieved the best for our goal.

Chapter 7

Fast and Light-weight Virtual Machine Cloning

7.1 Use Case and Requirements

From the use cases that we have mentioned in chapter 1.7, the software architecture of our cloning mechanism requires the interaction between the applications running inside the guest VM and the underlying hypervisor. For example, when the VM user starts/opens a suspicious program, the antivirus software in guest can choose to create a cloning from the current VM, execute and scan the suspicious program in the cloned VM, and gather the running result to analyze the program behavior.

Using the above use case as an example, the software architecture we proposed in Figure 7.1 requires a manager running inside hypervisor context to manage the cloning operations for the physical host, and a special and secure agent running inside each VM to communicate with the manager to perform the corresponding actions. The manager can be implemented as a user-space program running inside the Dom0 of Xen, and each agent can communicate with the manager by pre-configured network socket. For the cloning operation, we apply the following procedures.

- Similar to process fork in traditional OSs, each VM will be assigned a unique identifier, `VMID`, so that the agent can identify itself and perform the right actions. As the Figure 7.1 shows, the source VM is assigned `VMID_P` while the cloned VM will be assigned `VMID_C` after cloning.

- The agent in source VM triggers the cloning operation by sending a message to the manager, which performs the cloning operation. After cloning, each agent receives its own `VMID` from the manager and performs

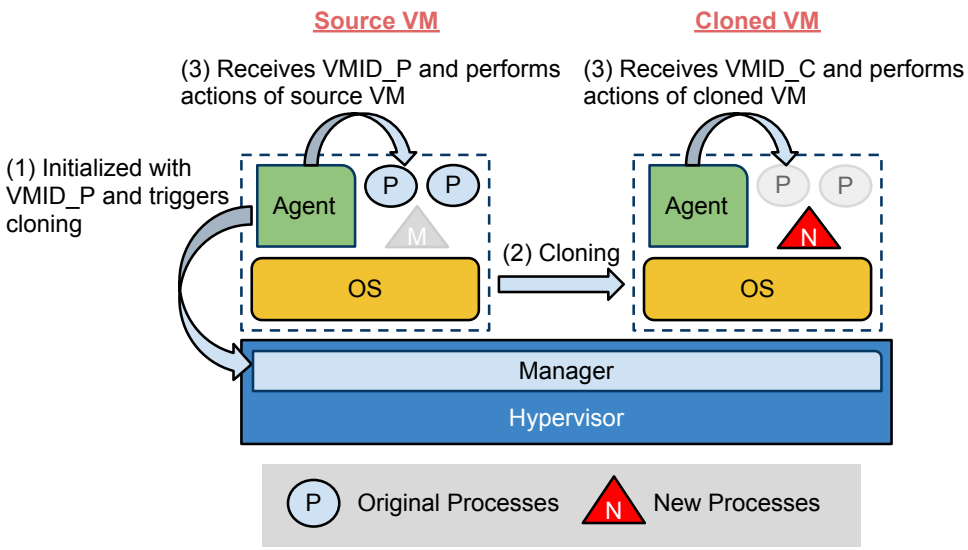

Figure 7.1: *Software architecture of Virtual Machine Cloning where the cloned VM is acting as a sandbox with same kind of execution environment as the source VM. The agent is responsible to invoke the cloning operation, suspend unnecessary processes and start new processes, e.g, suspicious programs for malware detection or security purposes.*

its own operations. Taking antivirus software as an example, the agent in cloned VM may choose to suspend or remove some original processes to cleanup the execution environment, and then invoke the suspicious program as new process to analyze its behavior. As for how to identify which process is necessary for a clean OS boot environment, the application can leverage the session or desktop manager concept from both Windows and Linux where the manager can record/invoke/kill processes such that the system can be brought up as the antivirus software wants. Similarly, the agent can open a DRM document or install new software patches and updates.

The good side of cloning is the cloned VM has exactly the same state as the source so that we do not have to boot the guest OS, load and run the application, or copy extra memory state in order to performs tasks. However, referring to the article of VMware knowledge base [74], there are problems due to the duplicate state that needs to be taken care by applications and system software. In the following, we list these problems and discuss the solutions.

- **MAC and IP address**: After cloning, the MAC and IP address of the cloned VM are all the same as the source. The first question is how the

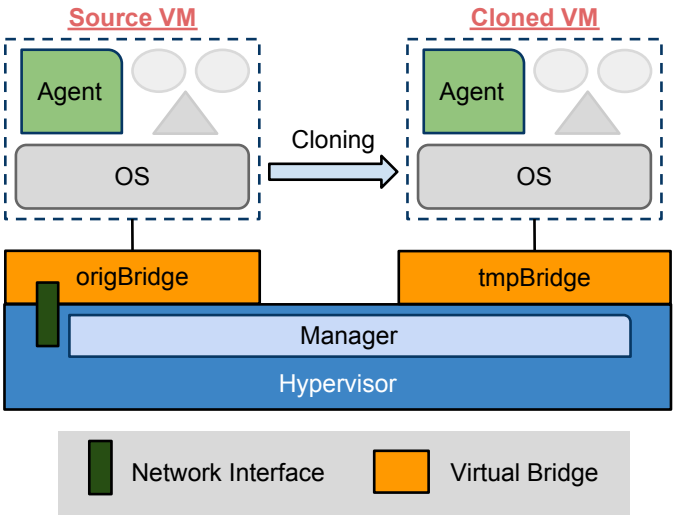

Figure 7.2: *Network configuration of VM cloning. The source VM is connected to a the hardware network interface via a virtual bridge, origBridge, for outgoing network while the cloned VM is connected to a temporary bridge, tmpBridge, without any network interface attached. Both the two bridges are presented as virtual network interfaces in hypervisor so that the hypervisor can talk to each VM.*

manager sends a unique message, e.g., the `VMID`, to each VM so that each agent knows the next step to perform. If this issue is solved, the manager can simply send new MAC and IP address to the cloned VM to reconfigure it if the cloned VM requires network connection to the outside world. As Figure 7.2 shows, the source VM is connected to the hardware network interface via a virtual network bridge, *origBridge*, setup by the manager. After cloning, the cloned VM is connected to a temporary virtual bridge, *tmpBridge*, without any network interface attached, and thus all the network traffic of cloned VM is confined within the host.

Both the two virtual bridges are presented as virtual network interfaces inside hypervisor. To send message to the source VM, the manager configures the default route to *origBridge* for the IP address of source VM[1]. On the other hand, the manager changes the default route to *tmpBridge* when it wants to send message to the cloned VM. Note that the change of route does not affect the external network connection of the source VM because there is no concern of IP routing in the level of

[1]Same as the cloned VM at this moment.

virtual bridge. Once each VM receives its `VMID`, it can make a decision to continue its operation depending on different use cases. To re-configure the MAC and IP address of cloned VM, manager can send a new MAC and IP address as message to its agent via *tmpBridge*. After the agent set the new MAC and IP address, the manager can connect the cloned VM to *origBridge* so that it can reach the external network. Of course, it is not the case for opening DRM document where the network of cloned VM could have been shutdown for security reasons.

- **Software Identifiers**: Some applications or software require unique identifier for each VM, and the cloned VM has to recreate a new one. For example, Microsoft use Windows Security Identifiers (SIDs) to represent machine identity and its security principals including user accounts and security groups. Thus, the cloned VM has to use Sysinternal utilities [75] to generate a new SID right after cloning. Another example is the UUID (Universally unique identifier) which is commonly stored in the Bios of motherboard to identify the machine itself. In the current virtualized environment, the manager has to generate another UUID for the cloned VM.

- **Software Transactions**: If the running application on source VM has certain memory state related to transactions, e.g., database server of online banking, then the cloned VM can not simply use the same software state as the source VM. The application has to take care of it by canceling or reseting the application state inside the cloned VM, which can be implemented by the software architecture of manager and agent we mentioned above.

After the above discussion, now the entire problem has been reduced to how to perform the HAL-based VM cloning efficiently.

7.2 HAL-based Virtual Machine Cloning

A naive implementation is to consider VM cloning as a special case of VM migration by migrating a VM from source to target without killing the source VM after migration. Note that the migration here refers to the non-live version, i.e., VM is paused during the entire migration, because the live migration takes longer time for the whole procedure to complete while we need it to be done as soon as possible. Then, the target will have the snapshot of source VM with same virtual CPU, memory, and device I/O states. However, there are two problems left for (1) the disk state and (2) the memory state.

First, the migration facility does not copy the disk state but assumes the disk is shared between source and destination VMs. After regular VM migration, only the destination VM is assumed to access the disk; otherwise, the disk state will be in-consistent. In the current implementation of Xen hypervisor, there exists two basic types of disk provisioning for VMs.

One type is to directly assign a raw disk partition to the VM, referred to as data disk, and the other type is to use a file image[2] to host the entire disk of VM, referred to as system disk. To use the data disk provisioning, we have explored the LVM (Logical Volume Manager) disk snapshot support [76] to perform snapshot of the source disk volume at the time of cloning, and assign the snapshot version to the cloned VM. After the disk snapshot, both of the VMs access their own disks with COW mechanism. In the following, we give a time breakdown for the naive implementation, referred to as **BasicClone** mechanism.

- Prepare cloned VM environment: each VM has certain kinds of metadata and structures maintained by hypervisor, e.g., virtual CPU and virtual device data structures, which costs around 100 milliseconds to create them.
- Snapshot disk state: using the LVM snapshot costs 1.1 second.
- Copy VM state: the time cost is linearly increased with VM memory size, e.g., copying 1 GB VM memory costs 403 milliseconds while 2 GB VM memory costs 824 milliseconds.
- Restore CPU, I/O states of cloned VM: this step is to restore the states into the virtual devices and initialize the device model of the VM, which costs 600 milliseconds for Linux VM.

However, the LVM snapshot feature costs too much time and we also noticed that the performance varies as the VM memory size changes. Thus, we choose to use the system disk provisioning and utilize the file system snapshot mechanism supported by *btrfs* [77] in Dom0. As for the general problem of data disk snapshotting, it remains unsolved and is left as the future work.

As we have discussed in the chapter 1.4, the majority of VM states is the memory, which can not be copied page-by-page directly because the size can go up to a few GBs and make it non-scalable for the entire cloning mechanism. In order to reduce the cloning time of memory state, one alternative is to copy the mapping of GFNs to MFNs, i.e., the P2M table or the EPT table in Intel

[2]The entire VM disk is represented as a regular file in the filesystem of Dom0, btrfs in our case, mounted as loopback device, and assigned to the VM.

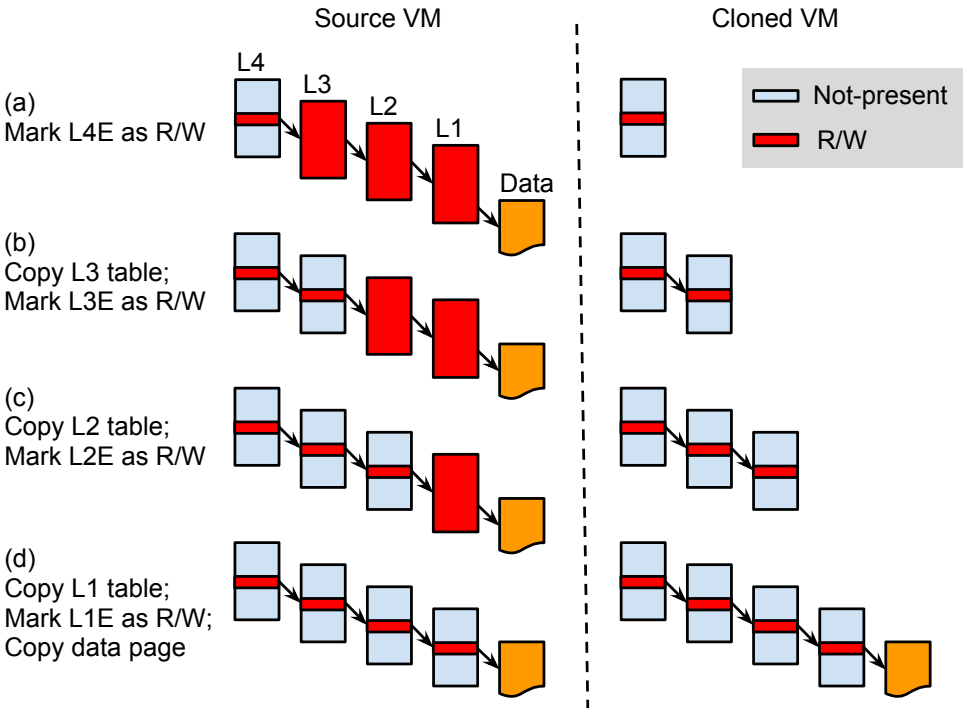

Figure 7.3: *The steps of lazily cloning EPT table upon the first memory access after cloning from either the source or cloned VM. L4, L3, L2, and L1 represents Level 4, 3, 2, and 1 EPT page table while L4E, L3E, L2E, and L1E shows the page table entry in L4, L3, L2, L1 page table respectively. While traversing down from L4 to L1 of EPT, the corresponding page table is synchronized to the cloned VM, and the page table entry in each level corresponding to the data page will be marked as R/W gradually. Finally, a new memory page for cloned VM is allocated, and the content is copied from the date page of source.*

architecture, from source to the cloned VM, and mark all memory pages as write-protected, which is the same as the COW mechanism of memory sharing in chapter 4. However, as memory size of VM increases, traversing the entire page table entries becomes time-consuming and the total time can not be bound to sub-second. Thus, we proposed to perform the lazy copying of the P2M table in the following section.

7.3 Lazy Memory State Cloning

In Intel architecture, the target platform in our work, the EPT page table is organized as a four-level hierarchy. During the cloning, we performed the following steps.

- Mark all highest-level entries of source EPT table as not-present.
- Copy the highest-level of EPT table from the source to cloned VM.

As Figure 7.3 shows, when either the source or the cloned VM accesses its memory pages first time after cloning, the VM execution will be trapped into hypervisor as the COA mechanism. We copy the EPT page table content related to the *address translation of the accessed page*, from the highest level L4 to the lowest level L1 to the cloned VM. All related page table entries (PTEs) i.e., L4E, L3E, L2E, and L1E in the figure, will be changed from *not-present* to *R/W* (Read-writable). A new memory page is allocated to the cloned VM with copied content from the source VM. From now on, the source and cloned VM can access or modify this recently accessed memory page without affecting each other. With this kind of lazy approach, we can skip the effort of copying the entire table at the moment of cloning and bound the entire processing time within one second. As for implementation, now we focus on cloning a VM on the same physical host running the Xen hypervisor.

7.4 Performance Evaluation

To evaluate the effectiveness of VM cloning, we used the cloning time as the metric to compare the following three mechanisms.

- **MigrateClone**: Clone the VM with migration facility while the disk of cloned VM is the btrfs snapshot from source VM.
- **TableClone**: Same as the *MigrateClone* mechanism except that memory pages are not copied and only the EPT page tables are copied with level 1 entries marked as read-only. The memory pages are synchronized between the source and cloned VM using the COW mechanism.
- **LazyClone**: Instead of copying the EPT page tables as *TableClone*, the page tables are lazily synchronized from the source VM to the cloned one using the COA mechanism described in chapter 7.3.

For the testbed setup, we use the same test machine and VM images as described in chapter 4.3.1.

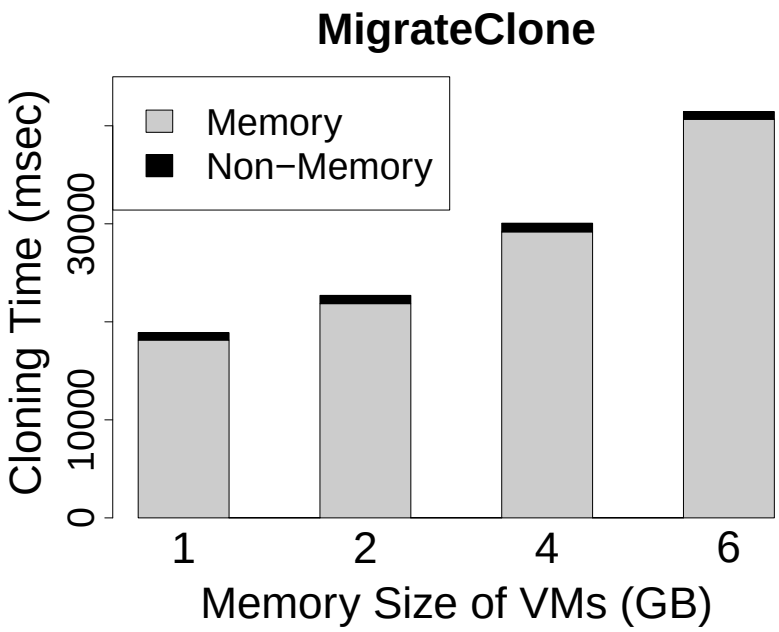

Figure 7.4: *The cloning time of Centos-64 VM with different memory size using the MigrateClone mechanism. The "Memory" part refers to time cost of cloning the memory state while "Non-memory" refers to other time consumptions including CPU and I/O states and the snapshot time of btrfs filesystem.*

Figure 7.4 shows the cloning of Centos-64 VM with memory size 1, 2, 4, and 6 GB using *MigrateClone* mechanism. The legend "Memory" shows the time of cloning memory state while the "Non-memory" part shows the cloning time of CPU and I/O states plus the time for btrfs to perform file system snapshot on source disk. From the result, the time of memory state cloning has dominated the entire procedure, and the total cloning time increases from 19, 23, 30 to 41 seconds as the VM memory size increases from 1, 2, 4, to 6 GB. In short, the *MigrateClone* mechanism is inefficient and non-scalable to the memory size of VM.

As for the *TableClone* and *LazyClone* mechanisms, we compare their cloning time in Figure 7.5. For each memory size of the VM, the *LazyClone* mechanism, the right bar, only costs 1 millisecond to perform the memory state cloning, i.e., to mark and copy the top level of EPT entries, which is bound to 512 entries in Intel architecture. Thus, only the "Non-memory" part of cloning contributes to the total time of cloning in the *LazyClone* mechanism, which is fixed among all results, i.e., 706 millisecond on average, because the states of CPU and I/O are small and do not change as the memory size changes, and the time cost of file system snapshot is also kept the same.

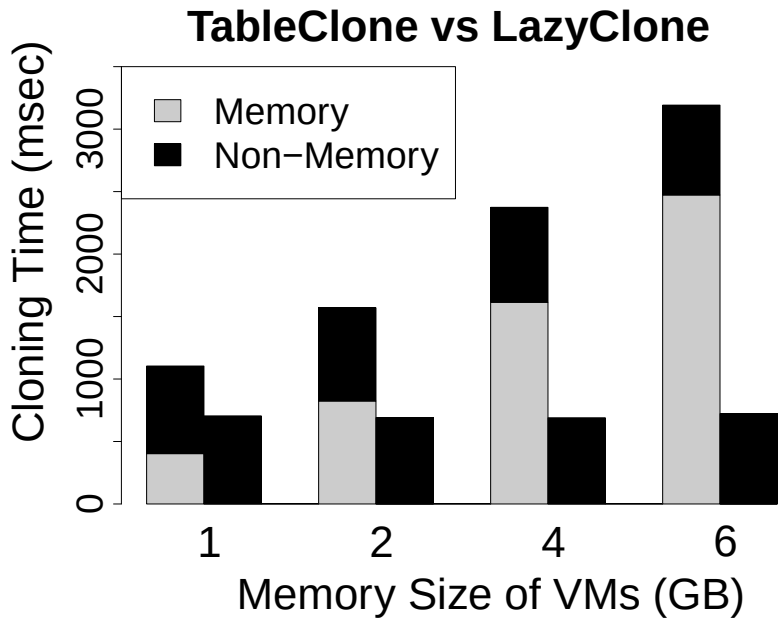

Figure 7.5: *Cloning time comparison of Centos-64 VM with different memory size 1, 2, 4, and 6 GB. For each memory size, the left bar represents the TableClone mechanism while the other one shows the result of LazyClone. The "Memory" part refers to time cost of cloning the memory state while "Non-memory" refers to other time consumptions including CPU and I/O states and the snapshot time of btrfs filesystem.*

On the other hand, the time of memory state cloning for the *TableClone* mechanism, left bar for each memory size, keeps growing from 403, 824, 1614, to 2473 milliseconds when the VM memory size increases from 1, 2, 4, to 6 GB. The growing time of memory state cloning is just proportional to the increase of VM memory size because the number of EPT entries also increases proportionally and becomes more time-consuming to iterate through them during the cloning process. As a result, the total cloning time of *TableClone* mechanism increases from 1104, 1570, 2374, to 3192 milliseconds for each VM memory size respectively. As for the Windows VM, we have performed the same experiment for the Win7-64 VM where the Non-memory part of cloning time has increased for 100 milliseconds among all VM memory size. The reason is due to an extra step for restoring the virtual sound device for the cloned VM. Eventually, the cloning time of *LazyClone* is still bound within sub-second, i.e., 807 milliseconds. To summarize, our proposed *LazyClone* approach scales well when the memory size of VM goes up and retains the sub-second performance.

7.5 Optimizing Cloning Time by Prefabricating the Cloned VM Environment

Unlike the VM migration, right before we perform the VM cloning, we already know the virtual hardware profile of the cloned VM, e.g., the number of virtual CPU and the device model of virtual disk. As chapter 1.4 describes, the cloned VM environment is constructed after the VM states are received at the destination side, which takes a while for hypervisor to build it up before the VM states can be restored.

Because the hardware profile of VM can be seen as an invariant for the cloning operation, we can pre-build the environment of cloned VM, referred to as *pre-cloning* mechanism, and thus save the total time of VM cloning. From our empirical result, we are able to further reduce the cloning time by 100 milliseconds, i.e., the cloning of Centos-64 and Win7-64 test VMs can be done in 607 and 707 milliseconds respectively.

7.6 Summary

While the VM cloning in OS-level virtualization can be easily achieved as the process forking in traditional OS, we have designed and implemented the corresponding HAL-based solution in this chapter. As the VM memory size increases, our *Lazy memory state cloning* keeps the cloning time constant by lazily synchronizing the EPT page tables between the source and the cloned VMs. To be concrete, we have demonstrated that the VM cloning time can be bound within *sub-second* for both Windows and Linux VMs on top of the latest Xen hypervisor. The cloning time does not change at all when the VM memory size rises up from 1 to 6 GB because of our lazy approach on memory state cloning. In addition, the *pre-cloning* mechanism pre-constructs the hardware environment for cloned VM and further save the entire cloning time by 100 milliseconds. With the sub-second performance, the HAL-based VM cloning can be easily used as a sandbox for various purposes such as DRM document reading, virus scanning, and software testing environment.

Chapter 8

Conclusion

8.1 Memory Virtualization-based Applications

As the data center size grows, the resource management becomes more important to the VM performance. The memory resource is currently the bottleneck of resource utilization, and it is challenging in terms of performance for various virtualization applications such as VM migration, VM de-duplication and compression, and VM cloning. In the following, we summarize our findings for each of them.

8.1.1 VM Migration

To consolidate system resource among PMs, VM migration is the primitive in the virtualized environment. It can be triggered when a physical host is going to run out of hardware resource or to be shutdown due to maintenance. At this moment, the system administrator or the system administration program will pickup certain VMs on the host and then choose a candidate physical host as the target of migration, e.g., the machine with the fewest VMs or the lowest CPU utilization. If the VM migration takes a long time, it consumes both the system resource and network bandwidth of the source and target physical hosts. Even worse, the VMs on the source PM may start to starve on hardware resource.

Thus, it is time critical to move the VMs to the target PMs where reducing the memory state size of migration is the most effective way. Using the automated VMI, the don't-care pages can be instantly identified inside the hypervisor space without causing any side-effect such as the overhead of COA from compressing guest VM's memory. On the other hand, as the resource management of VMs becomes more important, there are more applications

related to system administration running inside the hypervisor space. Thus, during the system maintenance, it is also necessary to keep the entire PM state without interrupting these application services. Therefore, our developed PMSM becomes important at this moment to migrate all applications or VM states on a PM to the target physical host.

8.1.2 VM Deduplication

To relieve the bound of virtualization ratio on top of a physical machine, it is important that one can effectively and efficiently increase the memory utilization ratio among VMs. From our study, most of the pages that can be de-duplicated are just zero pages. For a page with randomized content, it is unlikely to find another page with exactly the same content. Some researches try to find duplicate pages among similar VMs, e.g., both running Windows XP Service Pack 1; however, it is not always true to assume this case on a single physical machine. Thus, the traditional memory de-duplication could spend a great effort to compute hash value and compare content of memory pages, eve if the page is don't-care, i.e., the free memory pages of guest OSs. With the automated VMI technique, we are able to generalize the de-duplication process from page content to page type, which is able to de-duplicate non-zero don't-care pages, and speeds up the de-duplication engine by four times from empirical results.

8.1.3 VM Compression

The memory compression is more expensive than the de-duplication mechanism because for each read operation of a compressed page, the COA will be triggered to uncompress the page. In addition, while de-duplicating a page can reclaim one page back to hypervisor, it usually takes memory compression to compress a few pages to reclaim a single page. Thus, it is reasonable to use it as the secondary resort to increase memory utilization. In order to reduce the overhead of this mechanism, we have tried lots of effort to detect the cold pages of VM, i.e., pages outside of its working set, and use them as the candidate pages for compression.

In the beginning of the compression project, for the transparency purpose to VMs, we try to use the automated VMI technique to identify the page type of Linux guest VM and find cold pages from the page type, e.g., the page descriptor with `PageReferenced` field turned on represents that the page has been accessed recently. However, the Linux OS only performs the reclamation when the kernel is running out of memory, and the `PageReferenced` field is only set or cleared when the reclamation mechanism is active. Thus, it is not

possible to utilize this information outside of guest VM when there is no memory pressure internally. In the end, we found that the zram component [7] tries to compress the swapped-out pages in order to reduce the I/O delay of swapin, and then realize it could be used as part of our compression mechanism. Thus, we choose to implement the component inside the guest OS, and utilize the TWS-ballooning mechanism to trigger the guest reclamation, which identifies the guest working set with hardware reference bit on page table entry, and evicts candidate pages into zram for compression. Using the access probability of pages, we solved the problem of the original zram component and VMware when deciding the memory size used to hold the compressed pages.

The identified guest working set size is also important when memory balancing issue arises among multiple VMs. Instead of trapping the memory access, we want to utilize the guest reclamation knowledge which already exists, and do not want to do extra work on it. While our goal is to equalize the overhead of each VM, some other researches choose to assign the VM proportional to the VM working set size. Assume there are two customer paying the same amount of money for their VMs, the VM which consumes more memory should suffer more performance overhead, and then the Bal_prop mechanism can make sense in this scenario. There is no single answer for the memory balancing mechanism, and all depends on the policy and goals of the system administration in the data center.

8.1.4 VM Cloning

The live VM cloning has certain kinds of limitation in its usage as we have discussed in chapter 7.1. In our developed work, we do not expect to run a full-blown benchmark or workload inside the cloned VM. As the name of our project suggests, the cloning is done in a light-weight fashion and the application running inside VM will not have huge memory footprint and the cloned VM state can even be discarded afterwards. As a result, the current implementation with sub-second performance is very flexible to be applied on our proposed use cases.

8.2 Future Directions

From the experience of our developed works, we consider that the following topics could be interesting problems in the future.

8.2.1 PMSM with Heterogeneous Hardware

Although we have not support the PMSM across machines with different hardware profiles, we discuss the possible solutions when facing with heterogeneous CPU, memory, and I/O devices on the source and target machines.

The CPU compatibility is critical in migrating a physical machine's state to another physical machine, and has two aspects: the Instruction Set Architecture (ISA), and the number of CPU cores. VMSM [12, 78–80] ensures ISA-level compatibility by presenting to VMs the least common denominator of the ISAs of all CPUs participating in VM migration. Both VMotion and XenMotion support VM migration between different generations of CPUs from the same vendor, either Intel or AMD [12, 78]. KVM can even migrate between CPUs from different vendors [79]. In both cases, the ISA is the same between the source and destination CPUs. PMSM imposes the same requirement. More specifically, PMSM can leverage the CPU flag masking capability [81] to ensure that the destination CPU has the same ISA as the source CPU.

As for the number of cores in the source and destination CPUs, VMSM [12, 78] leverages the CPU hotplug capability [82] to dynamically adjust the number of virtual CPUs in a VM when it migrates between CPUs with a different number of cores. For example, VMotion can add/remove virtual CPUs to a VM when the VM is migrated if the VM's kernel features CPU hotplug. Similarly, PMSM can use the CPUID instruction [81] to detect new/missing CPU cores and then takes advantage of Linux's hot-plug feature to add or delete cores.

As for memory, PMSM requires that the destination machine have at least the same amount of memory as the source machine, but the additional memory may not be available to the migrated kernel. If the OS supports the memory hotplug feature [82], PMSM can leverage it to allow the migrated kernel to use all the memory on the destination machine.

To accommodate the I/O device differences between the source and destination machines, a simple solution for PMSM is to unload the I/O device drivers and detach the corresponding devices on the source machine, and automatically probe all available I/O devices on the destination machine and load the corresponding device drivers. However, this solution is not ideal because some kernel data structures may get reset by driver unloading and loading. For example, when a PCI-based Ethernet NIC's driver is unloaded and loaded, all high-level network connections are torn down.

One way to solve this problem is to hide the device driver unloading and loading events from the kernel. For example, the Shadow driver [41] addresses this problem by intercepting the interactions between native device drivers and the kernel's I/O subsystems, and presents to the kernel's I/O subsystems

the illusion that the device drivers are never unloaded on the source machine. On the destination machine, the Shadow driver loads the new device drivers in a such a way to preserve the same illusion to the kernel's I/O subsystems. This approach requires device-specific adaption to bridge the gap between the illusion and reality. However, it does not deal with the management issues for migration; for example, when the number of NICs on the source and destination machines differs, the migrated kernel may not be able to fully accommodate this difference. Nevertheless, PMSM can leverage the Shadow driver technology as a migration primitive to further minimize the disruption visible to user applications. Furthermore, Shadow driver has the potential to enable V2P and P2V migration because it can hide the differences of underlying I/O devices.

8.2.2 Resource Management and Analysis with Guest OS Information

One can leverage the guest information with introspection mechanism or generic guest API, and then helps on the following applications.

- Switch the DRAM DIMM modules corresponding to inactive pages to low-power mode, which is important to large data center because the energy consumption is the major expense of the cloud operators.
- Figure out the guest performance bottleneck so that the system administrator can identify the VM performance problem right before any disaster happens to the customers who run important applications inside their VMs. This issue becomes crucial to the cloud operator due to the difficulty to analyze performance problem in large-scale data center.

8.2.3 Hardware-assisted Working Set Size Estimation

From the new Intel architecture documentation [1], the company tries to support the access bit in the EPT page table entry in its future product, which can be used to detect page access of VMs and then determine the working set and the size of it as the Linux reclamation mechanism. Although there is no existing CPU product for this feature currently, it could be interesting to explore this feature in the future and then compare with the trap-based and our ballooning-based (TWS-ballooning) solutions. One benefit of using this new feature is that the hypervisor is able to detect the guest working set transparently outside the VMs, which can be more easily to apply to different guest OSs.

8.2.4 Extend the HAL-based VM Cloning Technique

Although our current implementation of VM cloning only works for 1-to-1 VM cloning, the lazy synchronization approach of EPT is not restricted to this scenario. It is also possible to apply the lazy approach to support 1-to-N or clone-of-clone VM cloning depending on the application usage. From example, the system administrator can promptly spawn multiple VMs for web server as the snowflock [37], so that the web service can be scaled up as fast as possible when a burst of web requests comes in. Also, to reduce the future traps on the memory access, one can proactively synchronize the page tables from the source to the cloned VM in the background before the memory pages are actually accessed, which is similar to the *post-copy* mechanism [25].

8.3 Summary of the Dissertation

To summarize, we have made the following contribution.

- Developed *PMSM* mechanism to generalize the machine migration to PMs. The prototype has been proved in the mainstream Linux OS and can migrate the PM state between identical machines within 6 seconds under MySQL server workload.
- Developed *bootstrapping VMI* technique to gather memory state information of un-modified Windows and Linux guest OSs.
- Developed Introspection-assisted Memory De-duplication to de-duplicate VM memory, and our proposed *Generalized Memory De-duplication* can on average runs 4 times faster than the traditional content-based de-duplication approach while incurring negligible CPU overhead.
- Developed *Introspection-assisted Virtual Machine Migration* to remove redundant memory states transfer to reduce the overall network impact and the total time of the live migration.
- Developed *Memory Compression with Working Set Estimation* to compress memory outside of the working set so that the memory utilization is increased while preserving application performance.
- Developed *Fast and Light-weight Virtual Machine Cloning* as sandbox that can be used as a cloud tool for security. And the lazy approach of copying P2M page table for memory state is proposed to confine the VM cloning time to sub-second.

- The proposed bootstrapping VMI technique has lots of foreseeable applications where the VM migration and de-duplication in this dissertation are just witnesses. Also, we expect it can be used for other management or debugging tools in the cloud environment.

With our proposed works, the physical memory in the cloud is virtualized as a way that we are able to perform almost the same operations as regular files on the physical memory, i.e. we are able to easily **move** (Introspection-assisted VM migration and PMSM), **share** (GMD engine), **compress**, and **clone** (VM cloning) physical memory state in the cloud. As a whole, the management of physical memory and VMs becomes more flexible and the entire memory utilization of the cloud is increased with low overhead.

Bibliography

[1] *Intel Virtualization Technology*. URL http://www.intel.com/technology/virtualization/.

[2] Inc. Intalio. Cloud computing is memory bound, 2010. URL http://www.intalio.com/cloud-computing-is-memory-bound.

[3] Clive Cook. Memory: The real data center bottleneck, 2009. URL http://virtualization.sys-con.com/node/1061473.

[4] Dan Williams, Hani Jamjoom, Yew-Huey Liu, and Hakim Weatherspoon. Overdriver: handling memory overload in an oversubscribed cloud. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '11, pages 205–216, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0687-4.

[5] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, and E. Zeidner. Internet Small Computer Systems Interface (iSCSI), 2004. URL RFC3720.

[6] Christopher Clark, Keir Fraser, Steven H, Jakob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live Migration of Virtual Machines. In *Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 273–286, 2005.

[7] Nintin Gupta. Compcache: in-memory compressed swapping. now renamed as zram. URL http://lwn.net/Articles/334649/.

[8] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, SOSP '05, pages 148–162, New York, NY, USA, 2005. ACM. ISBN 1-59593-079-5. doi: 10.1145/1095810.1095825. URL http://doi.acm.org/10.1145/1095810.1095825.

[9] Poul henning Kamp and Robert N. M. Watson. Jails: Confining the omnipotent root. In *In Proc. 2nd Intl. SANE Conference*, 2000.

[10] Yang Yu, Fanglu Guo, Susanta Nanda, Lap-chung Lam, and Tzi-cker Chiueh. A feather-weight virtual machine for windows applications. In *Proceedings of the 2nd ACM/USENIX Conference on Virtual Execution Environments (VEE'06)*, 2006.

[11] Citrix Systems Inc. Xen hypervisor, open source industry standard for virtualization, 2007. URL http://www.xen.org/.

[12] VMware Inc. VMotion, Migrate Virtual Machines with Zero Downtime, . URL http://www.vmware.com/products/vmotion/.

[13] Jacob G. Hansen and Asger K. Henriksen. Nomadic operating systems. Master's thesis, University of Copenhagen, Denmark, 2002.

[14] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, and Sebastian Schönberg. The performance of microkernel-based systems. In *16th ACM Symposium on Operating System Principles*, 1997.

[15] Jacob Gorm Hansen and Eric Jul. Self-migration of Operating Systems. In *Proceedings of the 11th workshop on ACM SIGOPS European workshop*, 2004.

[16] Michael A. Kozuch, Michael Kaminsky, and Michael P. Ryan. Migration without Virtualization. In *12th Workshop on Hot Topics in Operating Systems*, 2009.

[17] H. Kaminaga. Improving Linux startup time using software resume. In *Proceedings of the Linux Symposium, Volume Two*, 2006.

[18] Carl A. Waldspurger. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev.*, 36:181–194, December 2002. ISSN 0163-5980.

[19] Andrea Arcangeli, Izik Eidus, and Chris Wright. *Increasing memory density by using KSM*, pages 19–28. Linux Symposium, 2009.

[20] Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C Snoeren, George Varghese, Geoffrey M Voelker, and Amin Vahdat. Difference engine: Harnessing memory redundancy in virtual machines. OSDI '08, 2008.

[21] Derek G. Murray, Steven H, and Michael A. Fetterman. Satori: Enlightened page sharing. ATEC '09, 2009.

[22] Martin Schwidefsky, Ray Mansell, Damian Osisek, Hubertus Franke, Himanshu Raj, and Jong Hyuk Choi. Collaborative memory management in hosted linux environments. In *OLS06*, pages 313–331, 2006.

[23] Dan Magenheimer. *Transcendent Memory on Xen*, page 3. XenSummit, February 2009.

[24] Joel H Schopp, Keir Fraser, and Martine J Silbermann. Resizing memory with balloons and hotplug. *Linux Symposium*, 2:313319, 2006.

[25] Michael R. Hines and Kartik Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. VEE '09, pages 51–60, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-375-4. doi: 10.1145/1508293.1508301. URL http://doi.acm.org/10.1145/1508293.1508301.

[26] Jonas Pfoh, Christian Schneider, and Claudia Eckert. *Exploiting the x86 Architecture to Derive Virtual Machine State Information*. IEEE Computer Society, 2010.

[27] *AMD Secure Virtual Machine*. URL http://sites.amd.com/us/business/it-solutions/virtualization/Pages/virtualization.aspx.

[28] Xuxian Jiang and Xinyuan Wang. Out-of-the-box monitoring of vm-based high-interaction honeypots. *Lecture Notes in Computer Science*, pp:198–218, 2007.

[29] Fabrice Bellard. Qemu, a fast and portable dynamic translator. ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[30] Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. SP '11, pages 297–312, Washington, DC, USA, 2011. ISBN 978-0-7695-4402-1. doi: 10.1109/SP.2011.11. URL http://dx.doi.org/10.1109/SP.2011.11.

[31] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *NDSS*, pages 191–206, 2003.

[32] Crash, linux crash dump analysis tool. URL http://people.redhat.com/anderson/.

[33] Bryan D Payne, Martim D P De A Carbone, and Wenke Lee. Secure and flexible monitoring of virtual machines. *ACSAC 2007*, pages 385–397, 2007.

[34] Roy Bryant, Alexey Tumanov, Olga Irzak, Adin Scannell, Kaustubh Joshi, Matti Hiltunen, Andres Lagar-Cavilla, and Eyal de Lara. Kaleidoscope: cloud micro-elasticity via vm state coloring. EuroSys '11, pages 273–286, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0634-8. doi: 10.1145/1966445.1966471. URL http://doi.acm.org/10.1145/1966445.1966471.

[35] Pin Lu and Kai Shen. Virtual machine memory access tracing with hypervisor exclusive cache. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, ATC'07, pages 3:1–3:15, Berkeley, CA, USA, 2007. USENIX Association. ISBN 999-8888-77-6. URL http://dl.acm.org/citation.cfm?id=1364385.1364388.

[36] Weiming Zhao and Zhenlin Wang. Dynamic memory balancing for virtual machines. In *In Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual Execution Environments*, 2009.

[37] H. Andres Lagar-Cavilla, Joseph Whitney, Adin Scannell, Philip Patchin, Stephen M. Rumble, Eyal de Lara, Michael Brudno, and M. Satyanarayanan. Snowflock: Rapid virtual machine cloning for cloud computing. In *3rd European Conference on Computer Systems (Eurosys)*, Nuremberg, Germany, April 2009.

[38] Yifeng Sun, Yingwei Luo, Xiaolin Wang, Zhenlin Wang, Binbin Zhang, Haogang Chen, and Xiaoming Li. Fast live cloning of virtual machine based on xen. In *HPCC*, pages 392–399, 2009.

[39] Aberdeen Group, Inc. Virtual vigilance: Managing application performance in virtual environments, 12/31/2008. URL http://www.aberdeen.com/.

[40] CDW. CDW Server Virtualization Life Cycle Report: Medium and Large Businesses. URL http://newsroom.cdw.com/features/feature-01-11-10.html.

[41] Asim Kadav and Michael M. Swift. Live migration of direct-access devices. *SIGOPS Oper. Syst. Rev.*, 43(3):95–104, 2009. ISSN 0163-5980. doi: http://doi.acm.org/10.1145/1618525.1618536.

[42] Bernard B. Tuxonice, Linux equivalent of Windows' hibernate functionality. URL http://www.tuxonice.net/.

[43] Apache Benchmark. The Apache Software Foundation, ab - Apache HTTP server benchmarking tool, 2009. URL http://httpd.apache.org/docs/2.0/programs/ab.html.

[44] SPEC. Standard Performance Evaluation Corporation, SFS (System File Server) Benchmark 2008, 2008. URL http://www.spec.org/sfs2008/.

[45] Vadim Tkachenko. Tools and Utilities Used by Percona, 2008. URL https://code.launchpad.net/~percona-dev/perconatools/tpcc-mysql.

[46] T. J. Kowalski. Fsck—the unix file system check program. In *UNIX Vol. II: research system (10th ed.)*, pages 581–592, 1990. ISBN 0-03-047529-5.

[47] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers*. O'Reilly Media, 2005.

[48] Daniel P. Bovet and Marco Cesati Ph.D. *Understanding the Linux Kernel, Third Edition*. O'Reilly Media, 2005.

[49] Robert Love. *Linux Kernel Development, Second Edition*. Novell Press, 2005.

[50] Mel Gorman. *Understanding the Linux Virtual Memory Manager*. Prentice Hall, 2004.

[51] Christian Benvenuti. *Understanding Linux Network Internals*. O'Reilly Media, 2005.

[52] Daniel Muller. *The Linux Networking Architecture, Design and Implementation of Network Protocols in the Linux Kernel*. Prentice Hall, 2004. ISBN B0032VDV8M.

[53] VMware Inc. Virtual Machine to Physical Machine Migration, . URL http://www.vmware.com/support/v2p/doc/V2P_TechNote.pdf.

[54] Jon Oberheide, Evan Cooke, and Farnam Jahanian. Empirical exploitation of live virtual machine migration, 2007. URL http://www.eecs.umich.edu/techreports/cse/2007/CSE-TR-539-07.pdf.

[55] Mark E. Russinovich and David A. Solomon. *Microsoft Windows Internals, 4th Edition: Microsoft Windows Server 2003, Windows XP, and Windows 2000*. Microsoft Press, Redmond, WA, USA, 2004. ISBN 0735619174.

[56] Microsoft portable executable and common object file format specification. *ReVision*, page 97, 2010.

[57] Microsoft debug interface access sdk. URL http://msdn.microsoft.com/en-us/library/x93ctkx8.aspx.

[58] Mark Russinovich, David A. Solomon, and Alex Ionescu. *Microsoft Windows Internals: Including Windows Server 2008 and Windows Vista, 5th Edition*. Microsoft Press, Redmond, WA, USA, 2009. ISBN 0735625301.

[59] Mark Russinovich, David A. Solomon, and Alex Ionescu. *Windows Internals: Covering Windows Server 2008 R2 and Windows 7*. Microsoft Press, Redmond, WA, USA, 2011. ISBN 0735648735.

[60] R Fielding, J Gettys, J Mogul, H Frystyk, L Masinter, P Leach, and Tim Berners-Lee. Hypertext transfer protocol http/1. 1. *rfc2817*, 2010.

[61] Curl, command line tool for transferring data with url syntax. URL http://curl.haxx.se/.

[62] Gdb: The gnu project debugger. URL http://www.gnu.org/s/gdb/.

[63] Amazon ec2. amazon's web service for virtual machine privision. URL http://aws.amazon.com/ec2/.

[64] Sysmark2007, . URL http://www.bapco.com/products/sysmark2007preview/.

[65] Specweb2009, . URL http://www.spec.org/web2009/.

[66] Specjbb2005, . URL http://www.spec.org/jbb2005/.

[67] D. MAGENHEIMER. Add self-ballooning to balloon driver. discussion on xen development mailing list and personal communication, april 2008.

[68] Shailabh Nagar. Per-task delay accounting. URL http://www.kernel.org/doc/Documentation/accounting/delay-accounting.txt.

[69] Apache http server project. URL http://httpd.apache.org/.

[70] Spec cpu2006 benchmarks suites, . URL http://www.spec.org/cpu/.

[71] Sysbench: a system performance benchmark, . URL http://sysbench.sourceforge.net/.

[72] Mysql: open source database server. URL http://www.mysql.com/.

[73] Mark B. Friedman. Windows nt page replacement policies. In *Int. CMG Conference'99*, pages 234–244, 1999.

[74] Vmware knowledge base. cloning virtual machines in vmware vcenter and virtualcenter. URL http://kb.vmware.com/kb/1027865.

[75] Microsoft sysinternals advanced system utilities., . URL http://technet.microsoft.com/en-US/sysinternals.

[76] Logical volume manager in linux. URL http://sourceware.org/lvm2/.

[77] B-tree file system. URL https://btrfs.wiki.kernel.org/.

[78] Citrix Systems Inc. Citrix XenServer is now free. URL http://www.virtualization.info/2009/02/citrix-xenserver-is-now-free-xencenter.html.

[79] Red Hat Enterprise. Kernel Based Virtual Machine. URL http://www.linux-kvm.org.

[80] Microsoft Corporation. Windows Server 2008 R2 Virtualization with Hyper-V. URL http://www.microsoft.com/hyper-v-server/en/us/default.aspx.

[81] Intel Corporation. Maximize Data Center Flexibility Today and Tomorrow with Intel Virtualization Technology (Intel VT) FlexMigration Assist. URL http://download.intel.com/business/resources/whitepapers/flexmigration.pdf.

[82] Linux Hotplug Project. Hotplug Support for Distributions of GNU/Linux, 2001. URL http://linux-hotplug.sourceforge.net/.