

2012

Understanding and Leveraging Virtualization Technology in Commodity Computing Systems

Duy Le

College of William & Mary - Arts & Sciences

Follow this and additional works at: <https://scholarworks.wm.edu/etd>



Part of the [Computer Sciences Commons](#)

Recommended Citation

Le, Duy, "Understanding and Leveraging Virtualization Technology in Commodity Computing Systems" (2012). *Dissertations, Theses, and Masters Projects*. Paper 1539623603.

<https://dx.doi.org/doi:10.21220/s2-x1j5-p244>

This Dissertation is brought to you for free and open access by the Theses, Dissertations, & Master Projects at W&M ScholarWorks. It has been accepted for inclusion in Dissertations, Theses, and Masters Projects by an authorized administrator of W&M ScholarWorks. For more information, please contact scholarworks@wm.edu.

**Understanding and Leveraging Virtualization Technology
in Commodity Computing Systems**

Duy Le

Hanoi, Vietnam

Master of Science, The Francophone Institute of Computer Science, 2003

Bachelor of Science, Hanoi University of Technology, 2001

**A Dissertation presented to the Graduate Faculty
of the College of William and Mary in Candidacy for the Degree of
Doctor of Philosophy**

Department of Computer Science

**The College of William and Mary
August 2012**


Copyright © Duy Le

All Rights Reserved

APPROVAL PAGE


This Dissertation is submitted in partial fulfillment of
the requirements for the degree of


Doctor of Philosophy




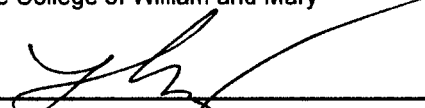
Duy Le

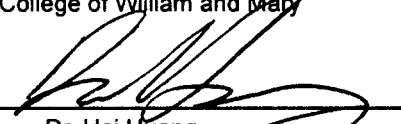
Approved by the Committee, June 2012



Committee Chair
Associate Professor Haining Wang, Computer Science
The College of William and Mary

Professor Weizhen Mao, Computer Science
The College of William and Mary

Associate Professor Qun Li, Computer Science
The College of William and Mary

Assistant Professor Gang Zhou, Computer Science
The College of William and Mary

Dr. Hai Huang
IBM T.J. Watson Research Lab

ABSTRACT PAGE

Commodity computing platforms are imperfect, requiring various enhancements for performance and security purposes. In the past decade, virtualization technology has emerged as a promising trend for commodity computing platforms, ushering many opportunities to optimize the allocation of hardware resources. However, many abstractions offered by virtualization not only make enhancements more challenging, but also complicate the proper understanding of virtualized systems. The current understanding and analysis of these abstractions are far from being satisfactory. This dissertation aims to tackle this problem from a holistic view, by systematically studying the system behaviors. The focus of our work lies in performance implication and security vulnerabilities of a virtualized system.

We start with the first abstraction—an intensive memory multiplexing for I/O of Virtual Machines (VMs)—and present a new technique, called Batmem, to effectively reduce the memory multiplexing overhead of VMs and emulated devices by optimizing the operations of the conventional emulated Memory Mapped I/O in hypervisors. Then we analyze another particular abstraction—a nested file system—and attempt to both quantify and understand the crucial aspects of performance in a variety of settings. Our investigation demonstrates that the choice of a file system at both the guest and hypervisor levels has significant impact upon I/O performance.

Finally, leveraging utilities to manage VM disk images, we present a new patch management framework, called Shadow Patching, to achieve effective software updates. This framework allows system administrators to still take the offline patching approach but retain most of the benefits of live patching by using commonly available virtualization techniques. To demonstrate the effectiveness of the approach, we conduct a series of experiments applying a wide variety of software patches. Our results show that our framework incurs only small overhead in running systems, but can significantly reduce maintenance window.

Contents

Dedication	v
Acknowledgments	vi
List of Tables	vii
List of Figures	ix
1 Introduction	1
1.1 Challenges	2
1.2 Goals	3
1.3 Dissertation Summary	4
1.3.1 Batmem: A Memory Optimization Mechanism	4
1.3.2 Performance Implications of Nested File System	5
1.3.3 Shadow Patching: Minimizing Maintenance Window	6
1.4 Dissertation Organization	6
2 Batmem: A Memory Optimization Mechanism	7

2.1	Introduction	7
2.2	Virtual Device Taxonomy	9
2.2.1	Virtual devices and MMIO optimization	9
2.2.2	User behavior classification	12
2.3	System Design and Implementation	12
2.3.1	System design	13
2.3.2	Implementation	16
2.4	Experimentation	20
2.4.1	High-end Systems	21
2.4.2	Low-end Systems	26
2.5	Discussion	29
2.6	Summary	30
3	Performance Implications of Nested File Systems	31
3.1	Introduction	31
3.2	Background	34
3.3	Macro-benchmark Results	36
3.3.1	Experimental Setup	36
3.3.2	Benchmarks	39
3.3.3	Macro-benchmark Results	40
3.4	Micro-benchmarks Results	49
3.4.1	Benchmark	49

3.4.2	Experimental Results	49
3.4.3	I/O Analysis	53
3.4.4	Disk Image Formats	63
3.5	Discussion	66
3.6	Summary	69
4	Shadow Patching: Minimizing Maintenance Window	71
4.1	Introduction	71
4.2	Related Work	74
4.3	Technical Background	77
4.3.1	Software patching	77
4.3.2	Virtual disk image and VM cloning	78
4.3.3	Disk cloning approach	79
4.4	Shadow Patching Framework	80
4.4.1	Patching scenario	80
4.4.2	Software component replacement	82
4.4.3	Prototype of Shadow Patching	85
4.5	Experimentation	86
4.5.1	Experimental setup	86
4.5.2	When Patch Succeeds	87
4.5.3	When Patch Fails	92
4.6	Discussion	97

4.6.1	Pros and Cons	97
4.6.2	Shadow Patching Enhancements	98
4.7	Summary	101
5	Conclusion and Future Work	102
5.1	Conclusion	102
5.2	Future Work	103
5.2.1	Memory Optimization in Virtualization	103
5.2.2	Virtualization Storage	105
5.2.3	Shadow Patching Enhancements	109
	Bibliography	113
	Vita	118

For Duc-Le and Ben Phong

For Hang Minh

For Parents and In-Laws

ACKNOWLEDGMENTS

Pursuing a Ph.D. is an opportunity to enjoy various stages of hard-working experience. With so many kind help I have been receiving from people, I understand that I would not finish my Ph.D. without their help. Though it will not be enough to express my gratitude in words to all people who supported and encouraged me, I would still like to thank to many people who made this dissertation possible.

First, I cannot overstate my appreciation to my Ph.D. advisor, Dr. Haining Wang. I would like to give my sincere thanks to him, who kindly accepted me as his Ph.D. student back in 2007. His continued encouragement, invaluable advice, and patient supervision, have always guided me in the right direction. He has not only been a strong and supportive advisor to me throughout my graduate school career, but he has always given me the freedom to pursue excellent work. To the end, I would think that without his help, I could not have finished my dissertation successfully.

My second debt of gratitude must go to my mentor and research collaborator, Dr. Hai Huang, who gave me the opportunity to extend upon my research work on file systems and storage. He also supported me patiently and guided me through the research projects during my Ph.D. experience. Most of all, I would like to thank Dr. Huang for his unflagging encouragement and for serving as a model of conducting scrupulous research.

Furthermore, a very special thanks goes to Dr. Evginia Smirni. Without her motivation and guidance, I would not have found an appreciation for the beauty of data analysis and performance evaluation, which have become the core of my Ph.D. work.

My sincere thanks go to my committee members, Dr. Weizhen Mao, Dr. Qun Li, and Dr. Gang Zhou for their valuable support and helpful suggestions. Their critical comments enabled me to explore corner cases of my dissertation work and make the necessary improvements. What has resulted is a tried and tested body of research.

I am indebted to my many student colleagues for not only collaborating in my Ph.D. research, but also for providing a stimulating and fun environment in which to learn and grow. I am especially grateful to Adam Wu, Chuan Yue, Tyler Smart, and Jidong Xiao for their kind assistance with conducting experiments, wise advice, helping with various applications, and so on. Their involvements in my Ph.D. research have served me well, and I owe them my appreciation.

I would like to thank my family for all of their love and encouragement: my parents and my parents-in-law who fully supported me in all of my pursuits; for my sister Hang-Minh, a mother of two, who encouraged me, helped me look after my parents when I was far away from home. I also thank my entire extended family for providing a loving environment for me – my uncles, aunts, and cousins have been extremely supportive. My family's understanding and love have encouraged me to work hard and continue pursuing the Ph.D. studies. Continually letting me know that they are proud of me has motivated me to work hard and do my best.

Lastly, and most importantly, I would like to convey my heartfelt thanks to my loving, supportive, encouraging, and patient wife Duc-Le Nguyen and my son Ben-Phong who form the backbone and origin of my happiness. Without their faithful and full support during the various stages of this Ph.D., I would not have finished. I owe my every achievement to both of them, and it is to them that I dedicate this dissertation.

List of Tables

2.1	VGA Bandwidth	23
2.2	Windows UHCI Bandwidth	24
2.3	Linux UHCI Bandwidth	24
2.4	VMBR Module Size	27
3.1	Testbed Setup	37
3.2	Physical and logical disk partitions	38
3.3	Parameters for Filebench workloads	40
3.4	Best and Worst Throuput	43
3.5	Best and Worst Latency	47
3.6	FIO Benchmarking	52
4.1	Time base comparison between files/directories	84
4.2	Testbed setup	87
4.3	Application services and benchmarks	89
4.4	Rule-based activities in merging deltas	91

4.5 Upgrading Ubuntu Distribution	93
---	----

List of Figures

2.1	Batmem overview	13
2.2	MMIO Optimization in Batmem	15
2.3	Partition Memory Registration	17
2.4	MMIO overhead	18
2.5	NIC Virtual Capacity	21
2.6	UDP Packet Delay	23
2.7	Browser Response Times	28
3.1	Nested File System Scenario	31
3.2	Experimental Testbed	37
3.3	File Server and Web Server Results of Throughput	41
3.4	Mail Server and Database Server Results of Throughput	42
3.5	CV of Throughput	44
3.6	Total I/O Size	44
3.7	File Server and Web Server Results of Latency	45
3.8	Mail Server and Database Server Results of Latency	46

3.9	CV of Latency	48
3.10	I/O Throughput in Reading	50
3.11	I/O Throughput in Writing	51
3.12	Sequential Disk I/Os	54
3.13	Cache Hit Ratio	56
3.14	Sequential Read I/O Times	57
3.15	Sequential Write Extra I/Os	57
3.16	I/O Characteristics JFS/ReiserFS	58
3.17	I/O Characteristics ReiserFS/XFS	60
3.18	Extra Written Data under JFS	62
3.19	Throughput different disk images	63
3.20	Latency under different disk images	64
3.21	Extra Data Written Into Disk	66
3.22	Experiments with other hypervisors	69
4.1	Scenario of Shadow Patching session	79
4.2	Upgrading process comparison	88
4.3	Individual upgrading application services	88
4.4	Upgrading different Ubuntu versions	95
4.5	Upgrading different Linux Distributions	96

Chapter 1

Introduction

Innovations in Virtualization Technology (VT) have significantly improved the utilization of hardware resources and have also enabled a wide array of products and services being offered. As a platform-virtualization software solution, hypervisor, known as a virtual machine monitor, has been widely used for supporting a diverse set of hardware devices and monitoring information between a host machine and multiple guest operating systems (OSes). For high-end systems, virtualization is attractive for server consolidation due to its strong resource and fault isolation guarantees. For example, in cloud computing environment, cloud vendors can quickly provide ready-to-use infrastructures, platforms, and software to customers in a low-cost virtualized environment. For low-end systems, such as mobile netbooks, laptops, or client desktops, virtualization provides a high-level OS interface for application programming via traditional real-time APIs, allowing programs to be performed on different OS platforms. However, virtualization is a double-edged sword. Along with many benefits that it brings, virtualized systems are also more complex, and thus, more difficult to understand, measure, and manage. This is often caused by layers of abstraction that virtualization introduces.

Unfortunately, current understanding and analysis of this abstraction are far from being sat-

isfactory. Current techniques, such as NormanSandbox [19], Anubis [2], Ether [46], and Paranorama [89], dynamically analyze the behavior of virtualized systems by monitoring system calls or API calls when malicious programs are performed. However, these analyzes have serious limitations: 1) they do not provide a performance implication of a virtualized system in a fine-grained manner under various configurations. 2) they do not uncover the hidden behavior of particular components inside a virtualized system. and 3) they do not reason the inner-working mechanism of hypervisor that causes abnormal behaviors on different running virtualized systems. Some research efforts [75, 41, 58] have been made to consider some of these limitations or to analyze a particular abstraction. However, none of them specifically addresses the problem of abstraction from a holistic view, and thus cannot provide a full understanding of the behavior of virtualized systems.

In this dissertation, we aim to tackle this problem of the abstraction from a holistic view, by systematically considering its challenges and goals, and thus fully understand the behavior of a virtualized system. Then, we propose a research direction to effectively leverage VT to minimize the maintenance window time for updating software in virtualized enterprise environments.

1.1 Challenges

Understanding and analyzing the abstraction offered by virtualization is a very challenging task. To determine an efficient approach for the analysis, we have to address the following two challenges: various configuration and fine-grained behavior.

- **Various Configurations:** We can see that virtualization is an ideal solution for high-end computing platforms due to its capability to leverage services. However, virtualization can

also be well employed in low-end systems. As a result, to conduct a thorough investigation on various host systems, not only a virtualization platform, but also emulated components are carefully selected, making it difficult to effectively differentiate the behavior of virtualized systems. Here, to facilitate the investigation, we focus on a memory multiplexing of virtualized systems and examine its behavior on both high-end and low-end host systems.

- **Fine-grained Behavior:** Virtualization provides a capability to monitor the behavior of a regular program or a particular I/O, which usually interacts with the system environment through system calls or API calls. However, system call monitoring is too fine-grained, resulting in much induced overhead. Therefore, to fully understand the system behavior at one particular type of abstraction—the nested file system—without imposing much overhead, our analysis needs to be conducted at user and kernel levels of both host and VM systems.

1.2 Goals

To fully understand the system behavior and performance implication in virtualized environment, our analysis should meet the following goals:

- **Addressing Challenges:** The challenges we discussed above are noteworthy in terms of virtualized system behavior analysis. A lack of addressing any of them may cause incompleteness or inaccuracy of the analysis, and the value of its experimental results will be greatly diminished.
- **Reasoning Inner-working:** Based on observed behaviors, it should be able to interpret at different levels of details, such as what exactly this behavior is and how this behavior happens.

For example in nested file systems, we want to know the dependency between using files and using physical disks to represent logical block devices at the guest VM.

- **Maintaining Efficiency:** In many cases, an effective analysis of system behavior strongly depends on various configuration, resulting in various running time and precision for experimentation. Therefore, it is very important to employ effective analysis methods that balance between accuracy and complexity.

1.3 Dissertation Summary

In order to effectively leverage VT, it is important for system designers to fully understand the behavior of virtualized systems. However, the current understanding and analysis of abstractions brought by VT are far from being satisfactory. In this dissertation, we first focus on a holistic view of the virtualized system behavior analysis and present the two projects on (1) memory optimization and (2) performance implications of nested file systems. Furthermore, we leverage storage utility in a virtualized environment to perform the third project of (3) a new patch management framework. The summaries of these projects are described as follows.

1.3.1 Batmem: A Memory Optimization Mechanism

Utilizing the popular VT, users can benefit from server consolidation on high-end systems and flexible programming interfaces on low-end systems. In these virtualization environments, the intensive memory multiplexing for I/O of VMs significantly degrades system performance. In this project, we present a new technique, called Batmem, to effectively reduce the memory multiplexing overhead

of VMs and emulated devices by optimizing the operations of the conventional emulated Memory Mapped I/O in Virtual Machine Monitor (VMM)/hypervisor. To demonstrate the feasibility of Batmem, we conduct a detailed taxonomy of the memory optimization on selected virtual devices. We evaluate the effectiveness of Batmem in Windows and Linux systems. Our experimental results show that: (1) For high-end systems, Batmem operates as a component of the hypervisor and significantly improves the performance of the virtual environment. (2) For low-end systems, Batmem could be exploited as a component of the VM-based malware/rootkit (VMBR) and cloak malicious activities from users' awareness.

1.3.2 Performance Implications of Nested File System

Virtualization allows computing resources to be utilized much more efficiently than those in traditional systems, and it is a strong driving force behind commoditizing computing infrastructure for providing cloud services. Unfortunately, the multiple layers of abstraction that virtualization introduces also complicate the proper understanding, accurate measurement, and effective management of such an environment. In this project, we focus on one particular layer: storage virtualization, which enables a host system to map a guest VM's file system to almost any storage media. A flat file in the host file system is commonly used for this purpose. However, as we will show, when one file system (guest) runs on top of another file system (host), their nested interactions can have unexpected and significant performance implications (as much as 67% degradation). From performing experiments on 42 different combinations of guest and host file systems, we give advice on how to and how not to nest file systems.

1.3.3 Shadow Patching: Minimizing Maintenance Window

Keeping software up-to-date is a fact of life in any IT environment. Although sophisticated live patching techniques have been available for many years, conventional offline methods are almost always used in practice. This is due to online methods being usually very OS or application specific and cannot be applied as generally as offline methods. In this project, we present a patch management framework that allows system administrators to still use the offline patching methods while retaining most of the benefits of live patching by leveraging commonly available virtualization techniques. To demonstrate the effectiveness of the approach, we conduct a series of experiments applying a wide variety of software patches. Our results show that our framework incurs only small overheads in running systems but can significantly reduce maintenance window.

1.4 Dissertation Organization

The remainder of this dissertation is structured as follows. In Chapter 2, we present Batmem as an effective memory optimization mechanism for hypervisor. In Chapter 3, we detail our investigation of nested file systems to explore their dependency at different levels of details. In Chapter 4, we present Shadow Patching as a novel and effective software updating mechanism for virtualized cloud environments. Finally, in Chapter 5, we conclude and discuss challenges of potential future work.

Chapter 2

Batmem: A Memory Optimization

Mechanism

2.1 Introduction

In high-end systems, virtualization allows its main memory to be shared and to be monitored between VMs [85, 69, 67]. The bottleneck of virtualized systems lies in VM multiplexing, which is dependent on system capacity features, including memory slot availability, additional power consumption, and the memory-sharing mechanism of hypervisor. Here the memory-sharing mechanism is known as page sharing, memory compression, or memory I/O multiplexing. Thus, the memory usage inside VMs and the memory-sharing mechanism in VM multiplexing are critical to a host's performance. As a result, commodity hypervisors require an effective memory-sharing mechanism between VMs and their host, such as optimizing frequent paging and memory that is mapped for virtual I/O devices.

However, due to the small capacity of a low-end system, multiple VMs cannot be installed on a single host. Thus, in terms of performance, memory sharing is not a critical issue for low-end

systems. Nevertheless, in terms of security, malware may exploit virtualization techniques including memory sharing to completely control VMs on low-end systems. SubVirt [60], BluePill [77], and Cloaker [45] are typical examples of Virtual Machine Based Rootkit (VMBR) that attempt to append a thin hypervisor as a middleware between a running OS and hardware devices. The success of VMBR relies on two factors: compromising devices and hiding malicious behaviors. More specifically, VMBR requires virtual devices to intercept the I/O operations of a victim OS, and then VMBR must cloak its malicious behaviors, which could include system modification violation or performance degradation. Therefore, reducing the overhead in memory multiplexing of VMs will not only improve the performance of high-end systems, but also help us understand the possibility of cloaking malicious VMBR behaviors in low-end systems.

We present *Batmem*, an effective technique to improve the performance of the Memory Mapped I/O (MMIO)—a conventional memory exchange mechanism—by reducing the overhead and redundant memory regions during the multiplexing of VMs. The key component of *Batmem* is a dynamic circular buffer that coalesces memory partitions to be written into the reserved memory areas of virtual devices. We also employ a compression algorithm to reduce the allocated memory regions used in such I/O writing. For either high-end or low-end systems, *Batmem* is applied on virtual devices, such as the Video Graphics Array (VGA), Network Interface Controller (NIC), and Universal Host Controller Interface (UHCI). In particular, for high-end systems, we use selective micro benchmarks to evaluate system performance at the device level. For low-end systems, *Batmem* functions as a VMBR component. To validate its effectiveness in concealing VMBR activities from users' observations, we evaluate the performance of selected user applications while maintaining

two malicious services: keylogger and data transmission.

With VT support, we implement Batmem as a prototype based on Kernel Virtual Machine (KVM) [61] and conduct experiments on emulated devices for both Windows and Linux systems. Even though other open-source hypervisors such as Xen [36] or lguest [13] support fault containment and performance isolation by partitioning physical memory among multiple VMs and allows unmodified guest OSes to run on a VT-x supported host, its particular domain-based architecture makes it impossible to compare with other light-weight hypervisor architectures, in terms of driver domain model and performance. In contrast, KVM inherits lguest's flexibility and turns a Linux kernel into an in-kernel hypervisor, in which OSes can directly run on the hardware and take advantage of VT-x.

2.2 Virtual Device Taxonomy

We first categorize virtual device emulations by analyzing three devices – Video Graphic Adapter, Network Interface Card, and Universal Host Controller Interface – and discuss possibilities for optimizing MMIO on such devices. Then, we classify end-user behavior regarding such optimization.

2.2.1 Virtual devices and MMIO optimization

Video Graphic Adapter (VGA): The complexity of VGA architecture highly depends on the various modes and modification capabilities of VGA hardware. In a virtual environment, its performance can be easily degraded due to instruction translations by the VGA emulator. The VGA emulator translates guest instructions into load/store instructions on the host. Identifying emulated

instructions that are directed to VGA hardware is difficult because the emulated VGA module may replace the load/store instruction with a branch to support the general-purpose functionalities of graphical operations. Consequently, the performance of the emulator module is adversely affected. To improve the performance of the emulated VGA, then, we should optimize the emulator module by improving either the speed of store/load instructions mapped to main memory or the accuracy of differentiating such instructions with in-line code generations. Because in-line code generations depend on hardware specifications, our work focuses on memory-mapped operations.

Network Interface Card (NIC): NIC information exchange requires two important communication features: highly sustained throughput and low latency. An emulated NIC is supported in two modes: bridge and virtual host. The bridge mode is more flexible and functional than the virtual host mode in that a VM is considered as an independent system on a LAN. Thus, our work focuses on the bridge mode. Moreover, virtual NIC functions are built as NIC modules. To allow the NIC to function effectively, VMM needs to make a trade-off between the flexibility and complexity of such modules. Recent research found that the majority of overhead is due to a non-optimization of the I/O exchange between a host and VMM [54]. More specifically, in each VMM-host context switch, the overhead is caused by asynchronous data mapping between the processor and memory address spaces of the virtual NIC. Therefore, to reduce the overhead of the virtual NIC, we consider optimizing its memory-mapping mechanism.

Universal Host Controller Interface (UHCI): In the USB architecture, UHCI consists of two main functions: building a data structure for device-to-application communication and providing

a register-level hardware interface for a compatible software driver. In virtual systems, UHCI and USB drivers are emulated as a host USB to take control of real USB devices attached to the host. The host USB emulates the USB buses and devices connected to them so that external USB devices appear and function properly under the guest OS. To monitor data transmissions between USB devices and an application, the host USB also emulates a root hub. In general, USB data transmissions are conducted in one of four modes: isochronous, interrupt, control, and bulk. The bulk mode is more commonly used to transfer a large amount of data under relaxed latency requirements than the others. Therefore, we focus on the bulk mode and consider optimizing its data transmission on UHCI.

MMIO optimization: MMIO uses the same address bus to address both memory and I/O devices, and the CPU instructions used to access the memory are also used for accessing I/O devices. In virtual systems, VMM emulates a context switch module to enable a guest OS using MMIO. When a device driver on the guest OS requires an MMIO on a particular emulated device, it issues a writing request and sends it to the context switch module. Then, according to the received writing request, the context switch module and the virtual CPU establish MMIO on the main memory and the device. In general, this MMIO establishment is monitored by a mapping module in VMM. Fortunately, the mapping module is accessible and can be modified as a regular kernel module. Therefore, we consider improving MMIO by optimizing such a mapping module. Moreover, not only VGA and NIC, but also UHCI and other emulated devices are considered as generic PCI devices in virtual systems, thereby enabling them to participate in the memory-mapping process. Page protection-based solutions decrease virtual/real timing characteristic differences but induce

virtualization overhead. Thus, we attempt to reduce such overhead by solving the MMIO latency problem.

2.2.2 User behavior classification

In accordance with user-perceived performance, we classify user behaviors into three different groups: screen-based, net-based, and file system-based. Note that the performance metrics are chosen according to their acceptable validity and reliability in previous studies [49, 70].

- Screen-based behavior is a variation of the end-user screen interaction, which is described by an instantly estimated frame-per-second (FPS) metric. The higher the FPS values, the closer the matching between a real VGA and an emulated VGA.
- Net-based behavior is a variation of the end-user network activity. We use performance metrics such as virtual capacity and packet delay of a NIC to quantify the variation. Here virtual capacity is defined as the maximum data transfer rate over the virtual NIC, specifically between the guest OS and outside networks.
- File system-based behavior affects user interactions on virtual file systems, such as emulated USB storage. The I/O activities of end-users will be affected by the bandwidth of file systems.

2.3 System Design and Implementation

In this section, we first detail the system design of Batmem. In particular, Batmem improves the speed of MMIO write by using (1) a dynamic circular buffer to group write requests and (2) a

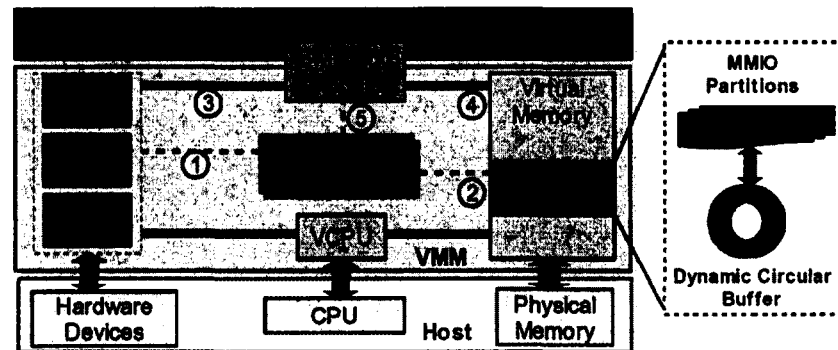


Figure 2.1: A virtual environment overview with Batmem

compression to minimize written memory partitions into the reserved memory. Then, we describe its implementation and related malicious services on KVM.

2.3.1 System design

In our design, Batmem participates in writing MMIO with other virtual components, such as the main memory, devices, and context switch. As shown in Figure 2.1, a virtual CPU (VCPUs) takes control of reading data from the device and writing MMIO data to memory. Batmem intercepts such an exchange by monitoring virtual device status (1) and I/O device operations (2) to control the MMIO writing on the reserved memory area. The context switch, known as a switching module, controls I/O requests sent from guest OS device drivers to virtual devices (3). With VCPUs, the switching module conducts the MMIO writing to virtual memory (4). To allow Batmem to properly participate in the MMIO writing, we need to establish a connection between Batmem and the switching module (5). This connection registers Batmem into a contact list of the switching module, making Batmem capable of monitoring I/O requests. These I/O requests are issued from the guest OS device driver controller, passing through the context switch in each MMIO session.

In each MMIO session, the issued MMIO requests are executed by I/O instructions at the switching module and VCPU. Such requests hold the address and size of the shared page of memory for each MMIO writing. When the Batmem/switching module connection is established, Batmem exchanges with the switching module to obtain information about the MMIO session, including the addresses belonging to MMIO page and reserved memory partitions. Each partition holds information about its size, address, and status (registered or unregistered). The registered partition introduces an occupied memory area, which is allocated and used by Batmem. The unregistered partition presents an extensible unoccupied memory area. The unregistered partitions are extended and used when current allocated memory partitions for Batmem are overrun by numerous arrivals of writing requests. Through Batmem, these partitions are registered or unregistered with the switching module. Note that such partitions are associated with another assigned memory area known as a batched buffer. To increase the MMIO writing speed from the reserved memory for a device to the main memory, we create the batched buffer as a dynamic circular buffer to store all MMIO partitions as a batch for each writing session.

The dynamic circular buffer structure is built on an ordinary circular buffer to prevent buffer underruns when devices perform numerous write-backs to the main memory. As shown in Figure 2.2-A, the dynamic circular buffer is a list of memory regions, where each element can be freed or ready to be filled upon receiving a writing request. To batch MMIO partitions, each buffer element needs to record the physical address and size of the mapped memory. Upon receiving notification from the switching module at session completion, Batmem informs the dynamic circular buffer to group all current partitions in the buffer. Instead of sequentially writing into the reserved memory,

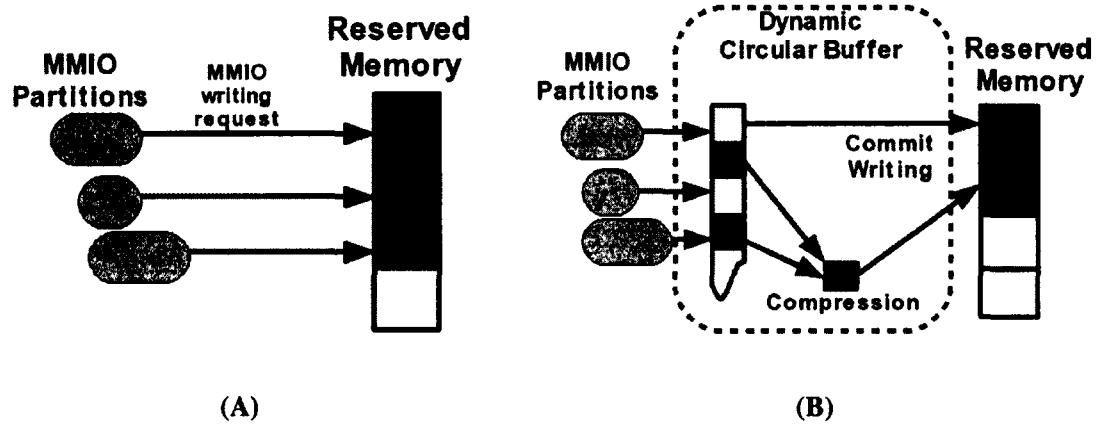


Figure 2.2: (A) Regular MMIO partitions written into reserved memory; (B) Batmem enhances writing speed by using dynamic circular buffer with compression

which is time consuming and may slow other devices's I/O on the main memory, Batmem simply completes an MMIO by copying the available buffer to the reserved memory. Since the buffer to be copied is in a mapped memory area that lies on the same main memory area, this copying is obviously much less expensive than the regular sequential MMIO writing. In order to eliminate buffer underruns when the requests of other devices fall behind, Batmem adjusts the size of the dynamic circular buffer by appending a number of free elements. Note that although a circular buffer has been widely used in sharing memory mechanisms, our improvement goes beyond the design of data structure by dynamically associating its functionalities with a memory compression in each writing session.

To reduce the memory footprint in the reserved memory, as shown in Figure 2.2-B, Batmem employs RLE to compress the batched memory regions in the dynamic circular buffer. Because such a compression is useful only when the compression ratio is high, we need to determine the regions to compress. At the beginning, instead of compressing an entire region, we just compress the first half of a region. Batmem defines a threshold to compare with the compression ratio of the first half

region's. If the measured compression ratio is higher than the given threshold, the compression is effective. The rest of the region is compressed and then is written into the reserved memory. Otherwise, the entire uncompressed batched region is committed to writing into the reserved memory as usual.

In the reserved memory, Batmem marks the compressed regions to differentiate them from others. When a read request from the VM accesses a compressed region, Batmem automatically decompresses and returns the region. Note that a compressed region and its mark remain intact until it is discarded or overwritten by new regions.

Batmem groups and compresses MMIO partitions without affecting the MMIO session. After the partitions to be grouped are successfully registered, the switching module notifies both Batmem and the devices to activate the MMIO batching. Using the device status provided by the switching module, Batmem can differentiate the device and its registered memory partitions from other devices that are not actively executed. Note that when VMM is initialized, to monitor the virtualized main memory, VCPU needs to map all first pages of device memory structures to the main memory. Since the regular size of the mapped memory is given and specified by VMM, Batmem maps the offset of this dynamic circular buffer to the first page of the main memory to easily locate the buffer in each MMIO session.

2.3.2 Implementation

We implement Batmem on KVM, an open-source based VMM/hypervisor, which operates as a subsystem leveraging the virtualization extension. The recent KVM version works as a Linux kernel module running under a VT-x supported host. As a benefit of the Linux kernel architecture, KVM

```
switch (ioctl)

case BATCHING_REG:

    bkvm_reg(kvm, &partition, reg_flag,...);

case BATCHING_UNREG:

    bkvm_unreg(kvm, &partition, unreg_flag,...);
```

Figure 2.3: Partition Memory Registration

can perform or schedule the OS as a Linux process. For high-end systems, Batmem is added into KVM as a module that maintains a capability to monitor multiple running VMs. For low-end systems, Batmem is implemented as a VMBR component that attempts to conceal the presence of running malicious services from the end-user. The implementation on KVM includes two main parts: Batmem and malicious services.

A. Batmem: Batmem takes advantage of the standard `ioctl()` functions under the Linux kernel to allocate, register, and unregister memory partitions. Such functions are immediately initialized with the KVM core module when the host is started. To protect allocated partitions from other processes that do not involve the MMIO writing session, the KVM core must be secure before and after each use. To secure the KVM core and schedule legitimate processes, we use a semaphore. As shown in Figure 2.3, we monitor a memory partition by using `flag` and `side` values that represent the results and side effects of the current batching process. The results consist of the registered partition information, including device identifications, the reserved memory size, and the circular dynamic buffer address. The side effects are considered to be either memory allocation latencies or

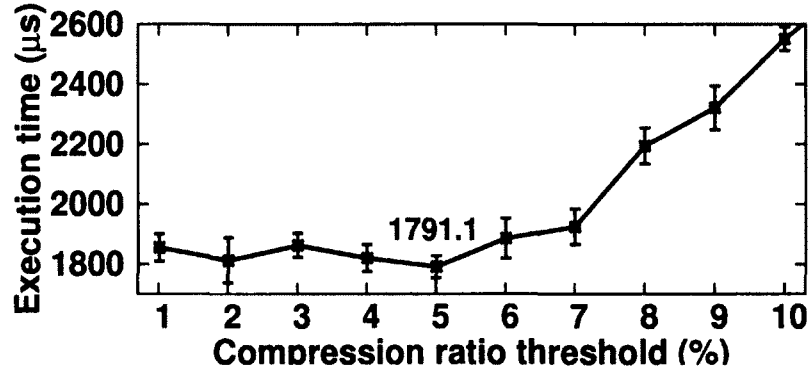


Figure 2.4: Average of total MMIO functions' overhead

buffer overrun circumstances. For the dynamic circular buffer, we start with a default size of 100 elements. If the number of partitions being used reaches the current buffer size, the buffer size is incremented by 10 elements. We choose these numbers to strike a balance between system memory usage and buffer overrun circumstances.

We need to minimize the overhead produced by compression/decompression operations. The overhead is measured in terms of the execution times of various functions involved in MMIO/Batmem, where we enable each function in isolation and evaluate its execution time. Figure 2.4 shows the average of total overhead imposed by major Batmem/MMIO operations, corresponding to the different compression ratio thresholds. As expected, the overhead grows with the increase of the compression threshold value. The overhead growth is primarily due to the increased number of batched regions that are available for compression. More specifically, we conduct multiple experiments using different compression ratio values as integers in a given range with an estimated error of the standard deviation. In each experiment, we maintain consistent batched memory regions as input for the compression. As a result, the ratio threshold of 5% is selected as the default value, with which the compression/decompression module only adds 1-1.5% overhead to the entire system.

For VGA and Rtl8193 NIC, we intercept their original registration functions to monitor both device information and writing processes. The interception directly points original registrations to our new registration routines. Therefore, we can perform batching on the mapped memory partitions of these devices upon receipt of their statuses. Since Batmem conducts the batching, this interception makes our new routines transparent to the guest OS device drivers.

For UHCI, we modify its registration and create its reserved memory for MMIO. To have the KVM UHCI function as a regular PCI device, we modify UHCI registration based on the core registration of a standard PCI device. However, on the KVM, UHCI is emulated without a reserved memory area. We create a reserved memory for UHCI on the main memory and add it to the contact list of the switching module. To allow UHCI to operate MMIO, we modify the UHCI initialization by directly assigning the destinations of its I/O operations to the new reserved memory. Note that such modifications do not affect the fundamental UHCI architecture.

B. Malicious services: We implement two malicious services as parts of VMBR: keylogger and data transmission between malware.

First, using the kernel keylogger concept [25], we implement the keylogger to compromise both the data buffers and I/O functions of the emulated keyboard controller. Since the emulated keyboard controller is operated as a kernel module within KVM, we need to recompile KVM with the keylogger to activate the service. To hijack a keystroke data buffer, the keylogger first checks the buffer availability, then performs its own read/write functions to copy the keystroke data to its buffer. The checking is executed via generated interrupts at an emulated serial port of KVM. We implement a small module to store the copied keystroke data as readable log files under the host.

Although the keylogger is not fully functional, such as encrypting keystroke data or sending it out to networks, we believe that its interception precisely represents a regular VMBR's keylogger service.

Second, to illustrate a data exchange between two pieces of malware, we implement a data transmission service for exchanging data between the user level of the guest OS and the kernel level of the host. The exchanged data is guest OS sensitive information, such as a Windows registry structure or a Linux file system map. An inside-the-guest malware functions at the user level of the guest OS. Another out-of-the-box malware operates at the kernel level of the host, more specifically, inside KVM. These two pieces of malware attempt to periodically send and receive data to each other by using implicit communication methods, such as interrupts, ports, or devices. The inside-the-guest malware cannot modify device drivers, and using interrupt-based or port-based communications is more challenging than a device-based method. We implement a simple protocol based on TCP/IP that allows both malware to send and receive data packets via an emulated NIC. The emulated NIC is initialized when the system is started with activated network services. VMBR can immediately perform this data transmission service afterwards.

2.4 Experimentation

We use benchmarks and sample payloads to evaluate the effectiveness of Batmem. First, for high-end systems, we conduct experiments with KVM/Batmem for three types of guest OSes: Windows XP, Ubuntu 7.10 Linux Kernel (LK) 2.6.21, and Fedora 8 LK 2.6.22. Each type has two guest OSes, for a total of six guest OSes. We run these guest OSes on a Tank GT20 server that includes a quad-core 2.0 GHz Intel Xeon processor and 4 GB RAM. Each guest OS uses 512 MB shared

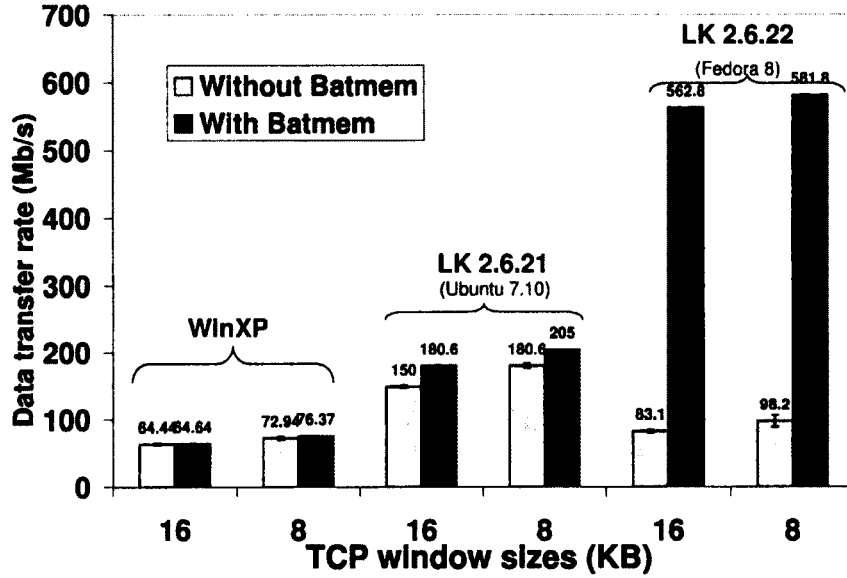


Figure 2.5: NIC virtual capacity (larger is better)

memory. We use benchmarks to examine the operations of intercepted devices. Second, for low-end systems, our evaluation includes two parts: analyzing the modules of Batmem based on their sizes and complexities, and measuring the varied run-time application behaviors when malicious services are activated. The running host consists of Intel 2.0 GHz and 1 GB memory, in which a shared 512 MB is for a guest OS.

2.4.1 High-end Systems

We use selected device level benchmarks to verify the effectiveness of Batmem on NIC, VGA, and UHCI. Our experiments are conducted in two scenarios: with and without Batmem. Each experimental result is an average of eight independent measurements along with an error estimate specified by the sample standard deviation. Due to the different running services involved and differences between UDP and TCP in terms of reliability and weight, we use Iperf [18] to measure the two parameters of virtual NIC, i.e., virtual capacity and UDP packet delay, which correspond to the

net-based behaviors. The measurements are conducted under different benchmark configurations. We cluster the performance results into different groups based on the running OS.

Figure 2.5 shows that Batmem works more effectively in Linux than in Windows. The results show that Batmem increases the virtual capacity of Windows by only 0.05%. However, these virtual capacity values are varied in Linux. In the LK 2.6.22, the virtual capacity is significantly improved by 490%, but only by 16.5% in the LK 2.6.21. These improvements are due to MMIO partitions, which belong to MMIO requests of the virtual NIC and are completely grouped by Batmem. Such grouping increases the data written into the main memory, and thus increases the virtual capacity. Note that without Batmem, the virtual capacity of the vanilla LK 2.6.22 is even less than that of the LK 2.6.21. The reason is that the vanilla LK 2.6.22 system applies some modifications on the TCP congestion control of the LK 2.6.21. On one hand, the beneficial modifications consist of merging sampling RTT, recomputing RTT updates, and resizing option fields with flag bits. In particular, the TCP socket buffer is required to consider invalid zero timestamps in communication with the RTT sampler upon the ACKed TCP retransmission request, and hence slightly affects its data transfer rate [14]. On the other hand, these modifications increase the number of MMIO requests and reduce the amount of data written into the main memory for each request. The reduced amount of written data lowers the virtual capacity. Therefore, we believe that these modifications of the TCP congestion control significantly improve the virtual capacity in the LK 2.6.22 when Batmem is active.

Figure 2.6 shows the effectiveness of Batmem in reducing UDP packet delays. The UDP packets are transmitted between the guest OS and the host through the virtual NIC. We conduct the

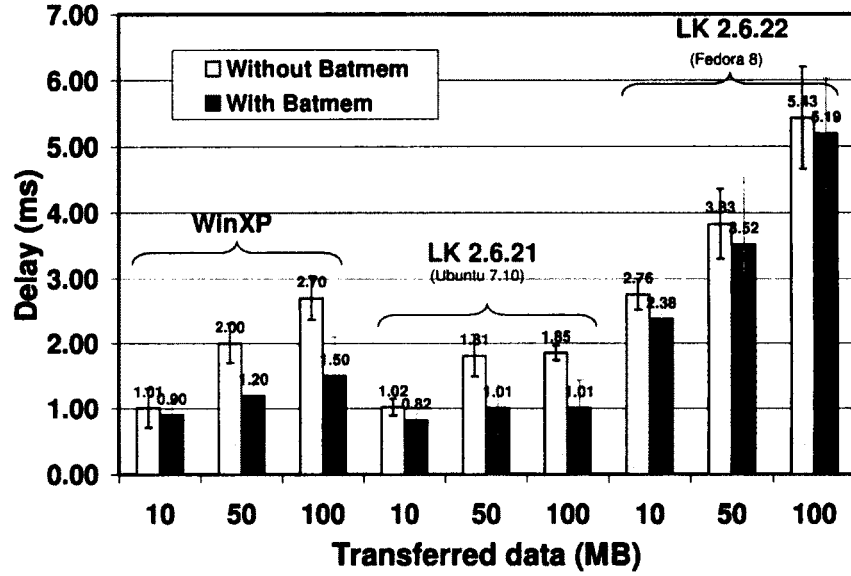


Figure 2.6: UDP packet delay (smaller is better)

	Mem cache ON	Mem cache OFF
WinXP	37.07 ±7.7	33.41 ±3.5
WinXP+Batmem	45.17 ±6.15	44.81 ±8.65

Table 2.1: VGA bandwidth (frames/s, larger is better)

experiments with different amounts of transferred data. Our experimental results demonstrate that Batmem helps Windows reduce the UDP packet delay up to 83%. In Linux systems, we observe that Batmem also reduces the UDP packet delay in the LK 2.6.21 by 45%, but by just 13% in the LK 2.6.22. As expected, in all the systems, Batmem works less effectively with the increase of transferred data because the NIC device driver progressively issues MMIO requests under such an increase. More specifically, the more MMIO requests are issued, the more partitions are re-allocated. Consequently, Batmem induces more overhead to group the partitions.

We use 3DBench [23] to measure the VGA memory bandwidth in Windows, which corresponds

	Write		Read	
	Sequential	Random	Sequential	Random
WinXP	156 \pm 5.2	126 \pm 0.5	122 \pm 3.2	170 \pm 0.6
WinXP+Batmem	480 \pm 6.1	351 \pm 0.4	391 \pm 4.6	242 \pm 0.7

Table 2.2: UHCI bandwidth under Windows (KB/s, NTFS, larger is better)

	Sequential per character		Random Seeks	Sequential create		Random create	
	Output	Input		Create	Read	Create	Read
LK2.6.21	3155 \pm 4.3	2007 \pm 2.8	132.5 \pm 1.1	232 \pm 2.4	477 \pm 6.3	348 \pm 1.0	475 \pm 2.0
LK2.6.21+Batmem	3560 \pm 5.6	2309 \pm 3.1	150.9 \pm 2.0	252 \pm 2.0	508 \pm 7.2	374 \pm 7.2	505 \pm 4.1
LK2.6.22	4010 \pm 4.1	3832 \pm 7.2	177.1 \pm 6.5	438 \pm 1.4	1055 \pm 5.4	717 \pm 7.2	1041 \pm 6.6
LK2.6.22+Batmem	9309 \pm 4.5	7924 \pm 6.5	469.2 \pm 4.2	885 \pm 2.0	2143 \pm 9.2	1453 \pm 7	2141 \pm 5.1

Table 2.3: UHCI bandwidth under Linux (KB/s, file size=128MB, chunk size=4KB, ext3, larger is better)

to the screen-based behavior. The benchmark intensively executes 3D routines that require aggressive I/O data exchanges on the VGA card. These data exchanges depend on three major factors, including processor speed, VGA bus size, and memory cache. Since the processor speed and the VGA bus size cannot be changed, to observe the variations of the VGA bandwidth, we conduct experiments in two cases, with and without memory cache. As shown in Table 2.1, for 60-100 seconds, Batmem helps the Windows system increase the actual VGA bandwidth, represented by FPS, in both cases by 20-33%.

We use the SiSoftware [22] in Windows and Bonnie++ [6] in Linux to measure the UHCI I/O performance, which corresponds to the file system based behavior. To avoid a sensitivity of the file system workload that may affect the overall performance of I/O, we consistently maintain a

data file for such measurements. Table 2.2 shows the improvement of read/write in Windows when Batmem is active. In particular, Batmem increases the I/O speed up to 220% in the sequential mode and 170% in the random mode. In Linux, to measure the operations of read, seek, and delete, we create a 128 MB file, clear the cache, and assign a 4 KB chunk for each operation. For writes, we create an empty file and keep writing 4 KB data chunks to the file until the file size reaches 128 MB. As shown in Table 2.3, Batmem takes advantage of the asynchronous write in the Ext3 file system when the number of MMIO sessions is increased, and thus increases the amount of exchanged data for a period of time. Moreover, Batmem increases the I/O speeds from 7% to 15% in the LK 2.6.21 and from 95% to 164% in the LK 2.6.22, respectively. The accelerations for the LK 2.6.22 are significant when Batmem is active. This is because patches are applied on the LK 2.6.22 to optimize inode read/write functions of the Ext3 file system, thereby increasing the speed of I/O requests. In fact, the improvement of the I/O request speed enhances the host/virtual I/O context switch. Consequently, Batmem can accelerate the MMIO writing and increase the UHCI I/O bandwidth.

We also conduct experiments to evaluate memory saving on the system. Our results show that the compression component can save up to 5% of memory. Since the saving is not significant, we plan to employ more effective compression algorithms for greater improvement in the future.

2.4.2 Low-end Systems

2.4.2.1 Module examinations

We examine our implemented modules on VMBR, including Batmem, malicious services, and the VMBR installation procedure, in terms of their size and complexity features. These modules must limit their sizes and complexities to hide themselves on systems. Since our VMBR is built on KVM, whose original size is given, our focus is on these new modules.

First, the Batmem module includes (1) vector structures, which form the dynamic circular buffer, (2) shared libraries, which consist of memory interactions with devices, and (3) a compression buffer, which supports memory compression. Even without applying source code optimization methods, as shown in Table 2.4, we observe that the module size of Batmem remains almost the same after the compilation (12 KB of source code and 13 KB of binary code). The slight difference between the two numbers is due to the use of the KVM shared memory library.

Second, of the malicious services, the keylogger implementation is more complex than the data transmission service. As a Linux kernel module, the keylogger uses low level kernel I/O functions to lock, read, and write the keyboard data. For data transmission, the inside-the-guest module benefits from high level functions to maintain its communication, while the out-of-the-box module uses primitive kernel read/write functions. Therefore, the binary size of the data transmission module is significantly expanded compared to the keylogger module. This comparison clearly shows the advantage of using low level library functions for malicious module implementations.

Third, we consider the VMBR installation as a procedure, instead of a part of the malicious module. This procedure functions as a script, which includes essential initializations on KVM and

	Batmem	Malicious Services		Installation procedure
		Keylogger	Data transmission	
Code	12,038	7,415	3,624	2,617
Module	13,332	9,194	8,936	-

Table 2.4: VMBR module size (Bytes)

devices, to instantly invoke KVM when the host is started. More specifically, since we only insert and activate this procedure script at the end of the boot sequence, the fundamental structure of the host boot sequence is not changed. In some cases, users may recognize a variation of the guest OS screen resolution because the emulated VGA is not automatically detected. However, this gap can be resolved if attackers retrieve accurate hardware VGA device information to properly configure the guest OS resolution.

2.4.2.2 User Level Experimentation

We run the selected user-level applications on guest OSes under two different conditions: with and without malicious services. The performance metric we used is application response time. As one of the most popular Internet applications, web browsers are sensitive to response time. Our selected applications include Internet Explorer in Windows and Firefox in Linux. We differentiate the response times in two cases, with and without Batmem, on compromised systems running malicious services. Note that we do not change the configuration of the web browsers during the experiments, and all web browser caches are cleared before each test to avoid possible side effects.

In our experiments with malicious services, malware is executed either separately as a single

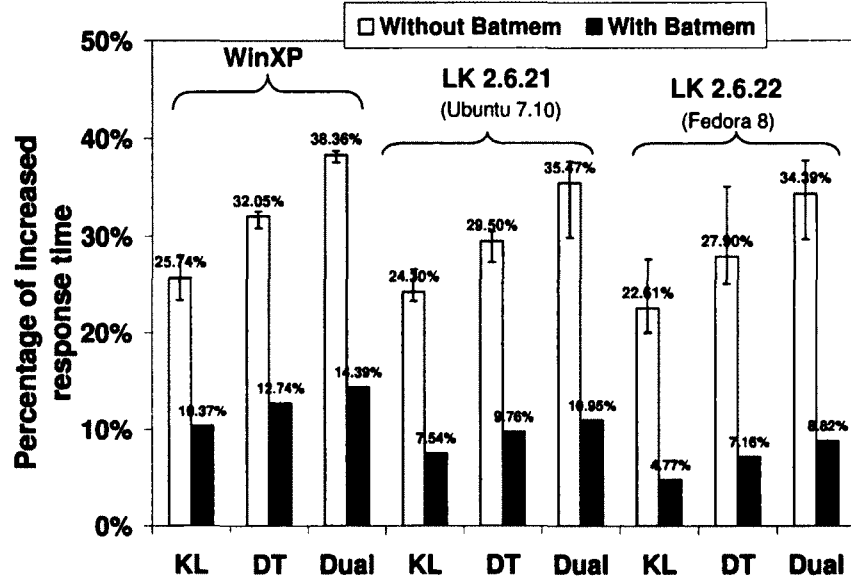


Figure 2.7: Web browser response times with malicious services (smaller is better)

service or simultaneously as a dual one (**Dual**). For the keylogger (**KL**), we use AutoHotkey [3] and Autokey [4] to generate keystroke patterns. For data transmission (**DT**), a connection is automatically established to exchange files between two malicious components. We conduct these experiments in two scenarios, without and with Batmem. Each experimental result is compared with the response time of a vanilla system (i.e., the base value \mathcal{T}). The lowest \mathcal{T} is 18.44 seconds in LK 2.6.21 and the largest \mathcal{T} is 23.62 seconds in Windows.

- **Without Batmem:** Through web browsers, we access a local website, download, and store a given data file into a USB drive. As shown in Figure 2.7, running malicious services significantly increases the user-perceived response times. For example, with a dual service, compared to the corresponding \mathcal{T} , the response time is increased by 38.47% in Windows and 35.76% in Linux.
- **With Batmem:** We repeat the previous tests. As expected, Batmem effectively reduces

VMBR overhead in both Windows and Linux. Thus, the user-perceived response times of the web browsers are greatly decreased, which is evidently shown in Figure 2.7. For Windows systems, the reduction is around 60%, while for Linux systems, the reduction is up to 80%.

Overall, our results clearly demonstrate the capability of Batmem in concealing VMBR's activities from user awareness. The overhead reduction by Batmem in Windows is not as much as that in Linux systems. We believe that this is due mainly to non-optimization of device context switches and I/O system calls in Windows systems.

2.5 Discussion

The dynamic circular buffer and memory compression techniques of Batmem can be applied to other hypervisors because they do not depend on a particular hypervisor architecture. Batmem only attempts to improve the speed of MMIO write by monitoring I/O functions on selected devices. More specifically, while the context switch and reserved memory areas are two primary components of the hypervisor, Batmem only optimizes their memory I/O exchanges and does not modify their fundamental operations. Therefore, the operations of these original components are not affected by Batmem.

For memory compression, the actual benefit is determined by a tradeoff between its overhead and compression ratio. While the chosen compression ratio threshold of 5% is not reasonably high, we believe that it is appropriate because the total system overhead is only increased by 1-1.5%. As expected, the compression behavior highly depends on the chosen algorithm. Although the applied

RLE is less effective than WKdm and/or Lempel-Ziv in terms of compression ratio [87], we also believe that the prototype of Batmem shows the potential of using such a simple technique to reduce memory redundancy in a virtual support system.

2.6 Summary

We have presented the design and implementation of Batmem, a technique that significantly reduces the overhead of the conventional memory exchange mechanism MMIO. To demonstrate its feasibility, we build Batmem in KVM and conduct experimentation in both high-end and low-end systems. For the high-end systems, we evaluate the performance improvement of virtual devices. For the low-end systems, Batmem functions as a VMBR component. Our experimental results on Windows and Linux show significant performance improvements with the use of Batmem in device-level benchmarks and user-level applications.

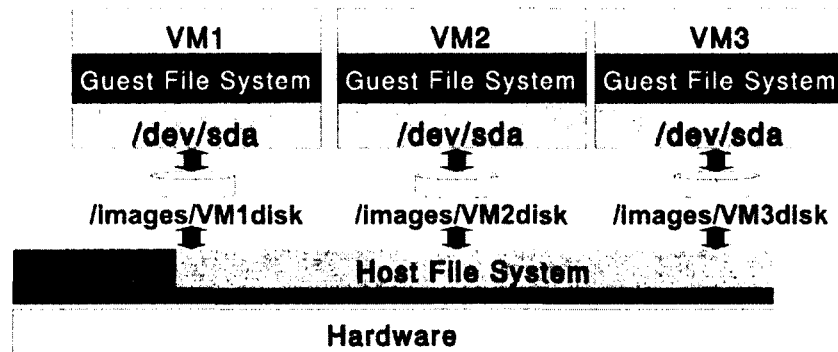


Figure 3.1: Scenario of nesting of file systems.

Chapter 3

Performance Implications of Nested File Systems

3.1 Introduction

Virtualization has significantly improved hardware utilization, thus, allowing IT services providers to offer a wide range of application, platform and infrastructure solutions through low-cost, commoditized hardware (e.g., Cloud [1, 10, 30]). However, virtualization is a double-edged sword. Along with many benefits it brings, virtualized systems are also more complex, and thus, more

difficult to understand, measure, and manage. This is often caused by layers of abstraction that virtualization introduces. One particular type of abstraction, which we use often in our virtualized environment but have not yet fully understood, is the nesting of file systems in the guest and host systems.

In a typical virtualized environment, a host maps regular files as virtual block devices to VMs. Completely unaware of this, a VM would format the block device with a file system that it thinks is the most suitable for its particular workload. Now, we have two file systems – a host file system and a guest file system – both of which are completely unaware of the existence of the other layer. Figure 3.1 illustrates such a scenario. The fact that there is one file system below another complicates an already delicate situation, where file systems make certain assumptions, based on which, optimizations are made. When some of these assumptions are no longer true, these optimizations will no longer improve performance, and sometimes, will even hurt performance. For example, in the guest file system, optimizations such as placing frequently used files on outer disk cylinders for higher I/O throughput (e.g., NTFS), de-fragmenting files (e.g., QCoW [20]), and ensuring meta-data and data locality, can cause some unexpected effects when the real block allocation and placement decisions are done at a lower level (i.e., in the host).

An alternative to using files as virtual block devices is to give VMs direct access to physical disks or logical volumes. However, there are several benefits in mapping virtual block devices as files in host systems. First, using files allows storage space overcommit when they are thinly provisioned. Second, snapshotting a VM image using copy-on-write (e.g., using QCoW) is simpler at the file level than at the block level. Third, managing and maintaining VM images and snapshots as files is also

easier and more intuitive as we can leverage many existing file-based storage management tools. Moreover, the use of nested virtualization [17, 39], where VMs can act as hypervisors to create their own VMs, has recently been demonstrated to be practical in multiple types of hypervisors. As this technique encourages more layers of file systems stacking on top of one another, it would be even more important to better understand the interactions across layers and their performance implications.

In most cases, a file system is chosen over other file systems primarily based on the expected workload. However, we believe, in a virtualized environment, the guest file system should be chosen based on not only the workload but also the underlying host file system. To validate this, we conduct an extensive set of experiments using various combinations of guest and host file systems including Ext2, Ext3, Ext4, ReiserFS, XFS, and JFS. It is well understood that file systems have different performance characteristics under different workloads. Therefore, instead of comparing different file systems, we compare the same guest file system among different host file systems, and vice versa. From our experiments, we observe significant I/O performance differences. An improper combination of guest and host file systems can be disastrous to performance, but with an appropriate combination, the overhead can be negligible.

The main contributions of this work are summarized as follows.

- A quantitative study of the interactions between guest and host file systems. We demonstrate that the virtualization abstraction at the file system level can be more detrimental to the I/O performance than it is generally believed.
- A detailed block-level analysis of different combinations of guest/host file systems. We un-

cover the reasons behind I/O performance variations in different file system combinations and suggest various tuning techniques to enable more efficient interactions between guest and host file systems to achieve better I/O performance.

From our experiments, we have made the following interesting observations: (1) for write-dominated workloads, journaling in the host file system could cause significant performance degradations, (2) for read-dominated workloads, nested file systems could even improve performance, and (3) nested file systems are not suitable for workloads that are sensitive to I/O latency. We believe that more work is needed to study performance implications of file systems in virtualized environments. Our work takes a first step in this direction, and we hope that these findings can help file system designers to build more adaptive file systems for virtualized environments.

3.2 Background

Virtualizing I/O, especially storage, has been proven to be much more difficult than virtualizing CPU and memory. Achieving bare-metal performance from virtualized storage devices has been the goal of many past works. One approach is to use para-virtualized I/O device drivers [76], in which, a guest OS is aware of running inside of a virtualized environment, and thus, uses a special device driver that explicitly cooperates with the hypervisor to improve I/O performance. Examples include KVM's VirtIO driver [76], Xen's para-virtualized driver [33], and VMware's guest tools [27]. Additionally, Jujuri *et al.* [57] proposed to move the para-virtualization interface up the stack to the file system level.

The use of para-virtualized I/O device drivers is almost a de-facto standard to achieve any rea-

sonable I/O performance, however, Yassour *et al.* [88] explored an alternative solution that gives guest direct access to physical devices to achieve near-native hardware performance. In this chapter, we instead focus on the scenario where virtual disks are mapped to files rather than physical disks or volumes. As we will show, when configured correctly, the additional layers of abstraction introduce only limited overhead. On the other hand, having these abstractions can greatly ease the management of VM images.

Similar to nesting of file systems, I/O schedulers are also often used in a nested fashion, which can result in suboptimal I/O scheduling decisions. Boutcher and Chandra [41] explored different combinations of I/O schedulers in guest and host systems. They demonstrated that the worst case combination provides only 40% throughput of the best case. In our experiments, we use the best combination of I/O schedulers found in their paper but try different file system combinations, with the focus on performance variations caused only by file system artifacts. Whereas, for performance purposes, there is no benefit to performing additional I/O scheduling in the host, it has a significant impact on inter-application I/O isolation and fairness as shown in [58]. Many other works [43, 47, 68, 78] have also studied the impact of nested I/O schedulers on performance, fairness, and isolation, and these are orthogonal to our work in the file system space.

When a virtual disk is mapped to an image file, the data layout of the image file can significantly affect its performance. QCOW2 [20], VirtualBox VDI [24], and VMware VMDK [28] are some popular image formats. However, as Tang [84] pointed out, these formats unnecessarily mix the function of storage space allocation with the function of tracking dirty blocks. Tang presented an FVD image format to address this issue and demonstrated significant performance improvements

for certain workloads. Various techniques [40, 55, 83] to dynamically change the data layout of image files, depending on the usage patterns, have also been proposed. Suzuki *et al.* [83] demonstrated that by co-locating data blocked used at boot time, a virtual machine can boot much faster. Bhadkamkar *et al.* [40] and Huang *et al.* [55] exploited data replication techniques to decrease the distance between temporally related data blocks to improve I/O performance. Sivathanu *et al.* [81] studied the performance effect of the image file placed at different locations of a disk.

I/O performance in storage virtualization can be impacted by many factors, such as device driver, I/O scheduler, and image format. To the best of our knowledge, this is the first work that studies the impact of the choice of file systems in guest and host systems in a virtualization environment.

3.3 Macro-benchmark Results

To better understand the performance implications caused by guest / host file system interactions, we take a systematic approach in our experimental evaluation. First, we exercise macro-benchmarks to understand the potential performance impact of nested file systems on realistic workloads, from which, we were able to observe significant performance impact. In Section 3.4, we use micro-benchmarks coupled with low-level I/O tracing mechanisms to investigate the underlying cause.

3.3.1 Experimental Setup

As there is no single “most common” or “best” file system to use in the hypervisor or guest VMs, we conduct our experiments using all possible combinations of popular file systems on Linux (i.e., Ext2, Ext3, Ext4, ReiserFS, XFS, and JFS) in both the hypervisor and guest VMs, as shown in

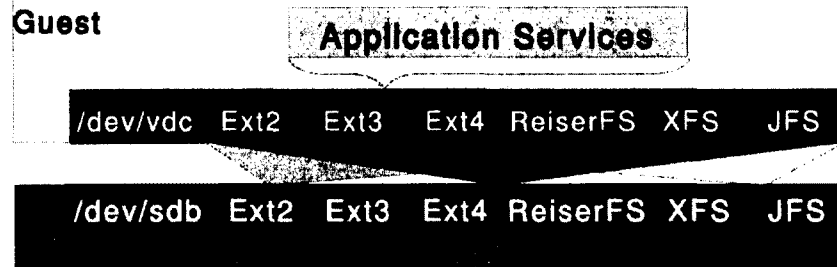


Figure 3.2: Setup for macro-level experimentation

	Hardware	Software
Host	Pentium D 3.4GHz, 2GB RAM	Ubuntu 10.04 (2.6.32-33)
	80GB WD 7200 RPM SATA (<i>sda</i>)	qemu-kvm 0.12.3
	1TB WD 7200 RPM SATA (<i>sdb</i>)	libvirt 0.9.0
Guest	Qemu 0.9, 512MB RAM	Ubuntu 10.04 (2.6.32-33)

Table 3.1: Testbed Setup

Figure 3.2. A single x86 64-bit machine is used to run KVM [61] at the hypervisor level, and QEMU [38] is used to run guest VMs¹. To reflect typical enterprise setting, each guest VM is allocated a single dedicated processor core. More hardware and software configuration settings are listed in Table 3.1.

The entire host OS is installed on a single disk (*sda*) while another single disk (*sdb*) is used for experiments. We create multiple equal-sized partitions from *sdb*, each corresponding to a different host file system. Each partition is then formatted using the default parameters of the host file system's `mkfs*` command and is mounted using the default parameters of `mount`. In the newly created

¹Similar performance variations are observed in the experiments with other hypervisors including Xen and VMWare, which are shown in 3.6.

Host file system				Guest file system		
Devices	#Blocks x10 ⁶	Speed(MB/s)	Type	Device	#Blocks x10 ⁶	Type
sdb2	60.00	127.64	Ext2	vdc2	9.27	Ext2
sdb3	60.00	127.71	Ext3	vdc3	9.26	Ext3
sdb4	60.00	126.16	Ext4	vdc4	9.27	Ext4
sdb5	60.00	125.86	ReiserFS	vdc5	9.28	ReiserFS
sdb6	60.00	123.47	XFS	vdc6	9.27	XFS
sdb7	60.00	122.23	JFS	vdc7	9.08	JFS
sdb8	60.00	121.35	Block Device			

Table 3.2: Physical and logical disk partitions

host file system, we create a flat file and expose this flat file as the logical block device to the guest VM, which in turn, further partitions the block device, having each corresponding to a different guest file system. By default, virtio [76] is used as the block device driver for the guest VM and we consider write-through as a caching mode for all backend storages. The end result is the guest VM having access to all combinations of guest and host file systems. Table 3.2 shows an example of our setup: a file created on `/dev/sdb3`, which is formatted as Ext3, is exposed as a logical block device `vdc` to the guest VM, which further partitions `vdc` into `vdc2`, `vdc3`, `vdc4`, etc. for different guest file systems. Note that all disk partitions of the hypervisor (`sdb*`) and the guest (`vdc*`) are properly aligned using `fdisk` to avoid most of the block layer interference caused by misalignment problems.

In addition to the six host file systems, we also create a raw disk partition that is directly exposed to the guest VM and is labeled as *Block Device (BD)* in Table 3.2. This allows a guest file system to sit directly on top of a physical disk partition without the extra host file system layer. This special case is used as our baseline to demonstrate how large (or how small) of an overhead the host file system layer induces. However, there are some side effects to this particular setup, and namely, the file systems being created on outer disk cylinders will have higher I/O throughput than those created on inner cylinders. Fortunately, as each disk partition created at the hypervisor level is 60GB, only a portion of the entire disk is utilized and thus limits this effect. Table 3.2 also shows the results of running `hdparm` on each disk partition. The largest throughput difference between any two partitions is only about 5%, which is fairly negligible.

The choice of I/O scheduler at host and guest levels can significantly impact performance [41, 56, 78, 79]. As file system is the primary focus of this work, we used CFQ scheduler in the host and Deadline scheduler in the guest as these schedulers were shown to be the top performers in their respective domains by Boutcher and Chandra [41].

3.3.2 Benchmarks

We use Filebench [8] to generate macro-benchmarks of different I/O transaction characteristics controlled by predefined parameters, such as the number of files to be used, average file size, and I/O buffer size. Since Filebench supports a synchronization between threads to simulate concurrent and sequential I/Os, we use this tool to create four server workloads: a file server, a web server, a mail server, and a database server. The specific parameters of each workload are listed in Table 3.3, showing that the experimental working set size is configured to be much larger than the size of the

Services	# Files	# Threads	File size	I/O size
File server	50,000	50	128KB	16KB-1MB
Web server	50,000	100	16KB	512KB
Mail server	50,000	16	8-16KB	16KB
DB server	8	200	1GB	2KB

Table 3.3: Parameters for Filebench workloads

page cache in the VM. The detailed description of these workloads is as follows.

- **File server:** Emulates a NFS file service. File operations are a mixture of `create`, `delete`, `append`, `read`, `write`, and `attribute` on files of various sizes.
- **Web server:** Emulates a web service. File operations are dominated by reads: `open`, `read`, and `close`. Writing to the web log file is emulated by having one `append` operation per `open`.
- **Mail server:** Emulates an e-mail service. File operations are within a single directory consisting of I/O sequences such as `open/read/close`, `open/append/close`, and `delete`.
- **Database server:** Emulates the I/O characteristic of Oracle 9i. File operations are mostly `read` and `write` on small files. To simulate database logging, a stream of synchronous `writes` is used.

3.3.3 Macro-benchmark Results

Our main objective is to understand how much of a performance impact nested file systems have on different types of workloads, and whether or not the impact can be lessened or avoided. As

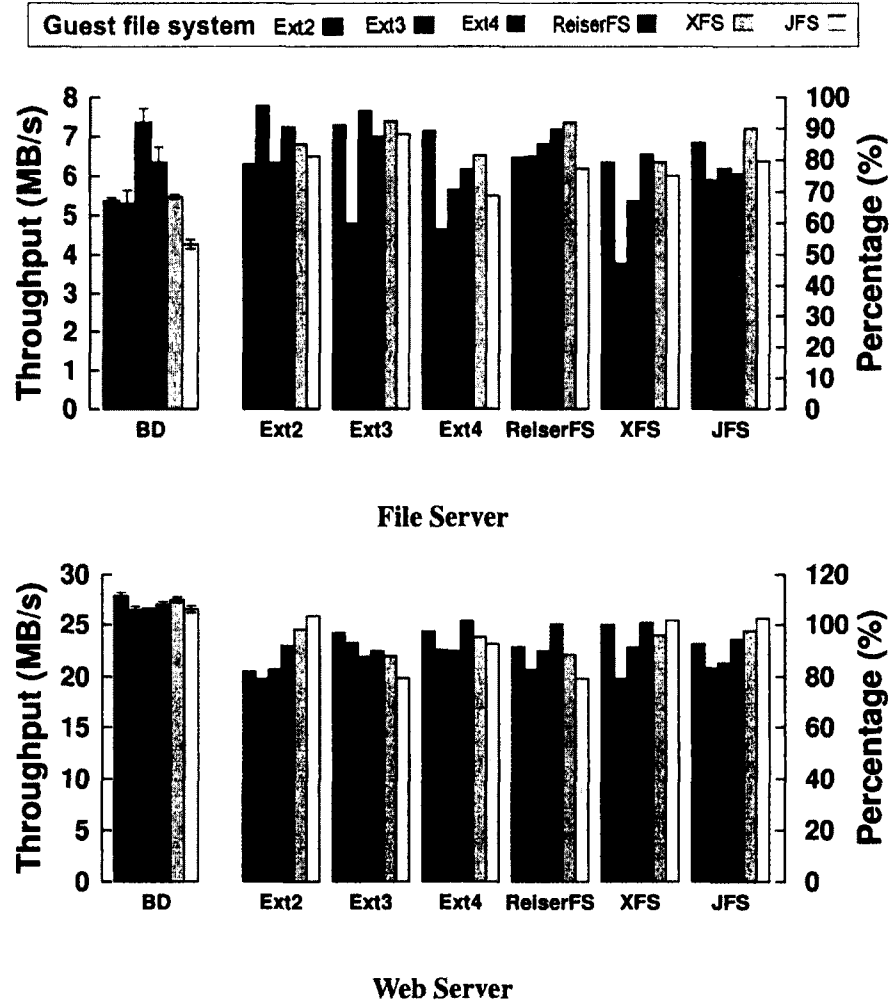


Figure 3.3: I/O throughput for Filebench workloads (**higher is better**)

mentioned before, we use all combinations of six popular file systems in both the hypervisor and guest VMs. For comparison purpose, we also include one additional combination, in which the hypervisor exposes a physical partition to guest VMs as a virtual block device. This results in 42 (6×7) different combinations of storage / file system configurations.

The performance results are shown in Figures 3.3, 3.4, 3.7, and 3.8, in terms of I/O throughput and I/O latency, respectively. Each sub-figure consists of a left and a right side. The left side

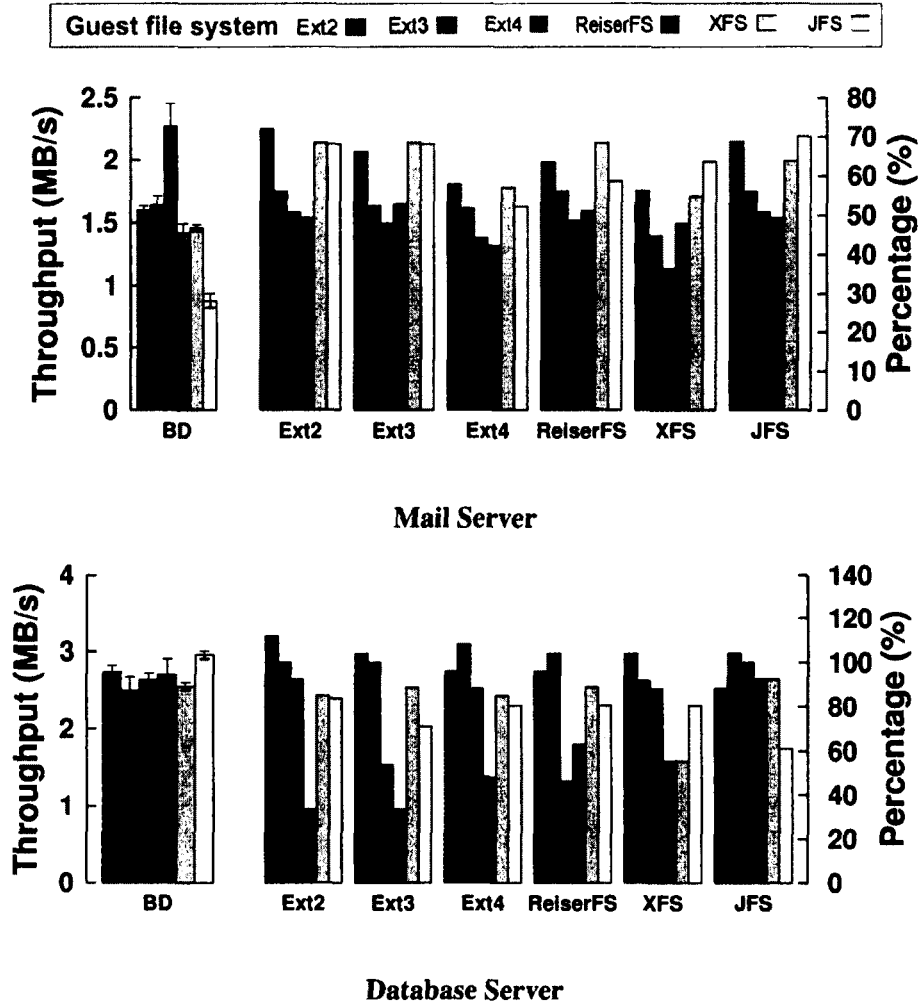


Figure 3.4: I/O throughput for Filebench workloads (higher is better)

shows the performance results when the guest file systems are provisioned directly on top of raw disk partitions in the hypervisor. These are expressed in absolute numbers (i.e., MB per second for throughput or millisecond for latency) and are used as our baseline. The right side shows the relative performance (to the baseline numbers) of the guest file systems when they are provisioned as files in the host file system. In these figures, each column group represents a different storage option in the hypervisor, and each column within the group represents a different storage option in the guest

	File server		Web server		Mail server		Database	
Guest File Systems	Worst	Best	Worst	Best	Worst	Best	Worst	Best
Ext2	79	91	82	100	56	72	84	106
Ext3	47	97	79	93	45	56	81	104
Ext4	67	96	83	91	36	51	41	104
ReiserFS	76	91	90	101	42	53	33	94
XFS	79	93	88	98	55	68	69	102
JFS	69	88	79	103	52	70	64	88

Table 3.4: Best and worst case I/O throughput (relative to baseline) of each guest file system across different host file systems (%).

VM.

3.3.3.1 Throughput

The baseline numbers (leftmost column group) show the intrinsic characteristics of various file systems under different types of workloads. These characteristics indicate that some file systems are more efficient on large files than small files, while some file systems are more efficient at reading than writing. As an example, when ReiserFS runs on top of BD, its throughput under the web server workload (27.2 MB/s) is much higher than that under the mail server workload (1.4MB/s). These properties of file systems are well understood, and how one would choose which file system to use is a straight-forward function of the expected I/O workload. However, in a virtualized environment where nested file systems are often used, the decision becomes more difficult. Based on the experimental results, we make the following observations:

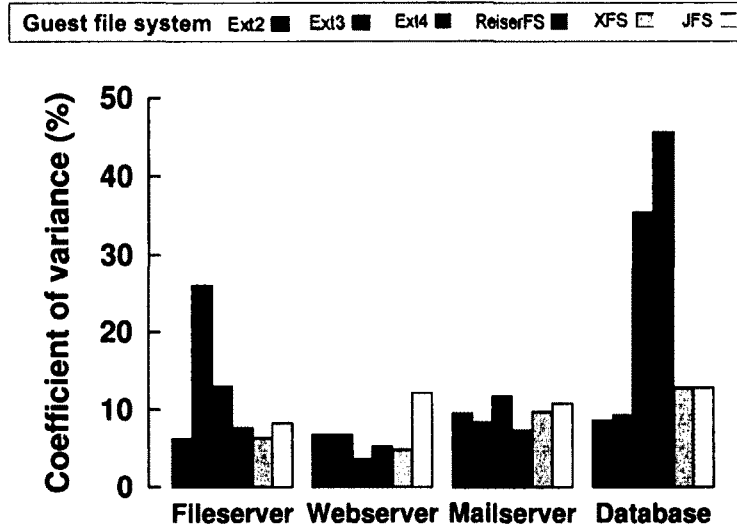


Figure 3.5: Coefficient of variance of guest file systems' throughput under Filebench workloads across different host file systems.

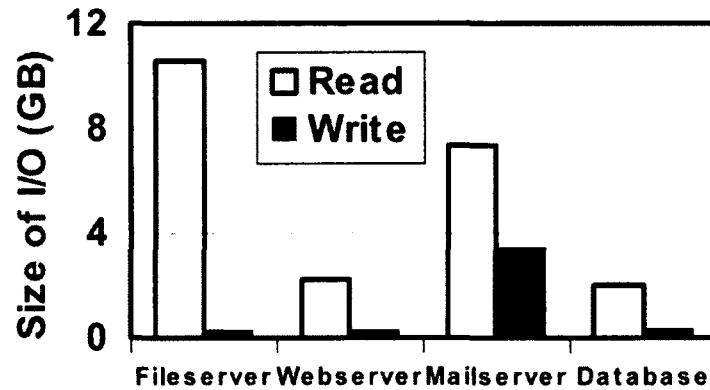


Figure 3.6: Total I/O transaction size of Filebench workloads

- A guest file system's performance varies significantly under different host file systems.

Figure 3.4 shows an example of the database workload. When ReiserFS runs on top of Ext2, its throughput is reduced by 67% compared to its baseline number. However, when it runs on top of JFS, its I/O performance is not impacted at all. We use coefficient of variance to quantify how differently a guest file system's performance is affected by different host file systems, which is shown in Figure 3.5. For each workload, a variance number is calculated based

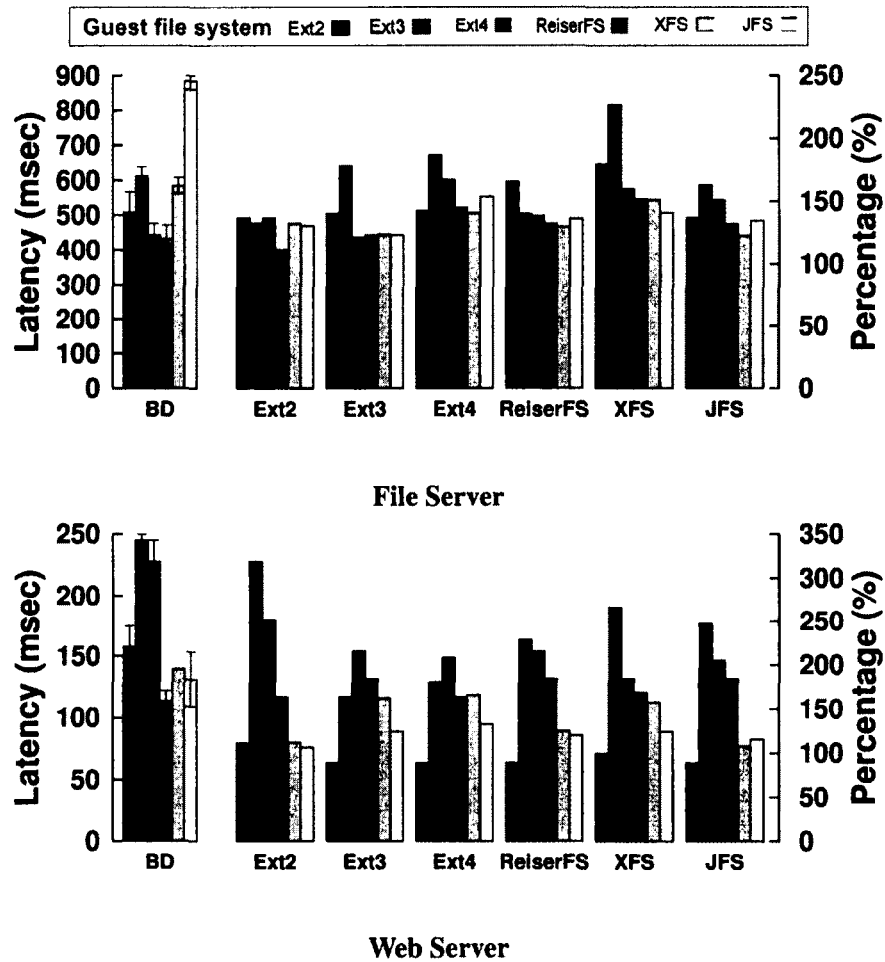


Figure 3.7: I/O latency of guest file systems under different workloads (lower is better)

on relative performance values of a guest file system when it runs on top of different host file systems. Our results show that the throughput of ReiserFS experiences a large variation (45%) under the database workload, while that of Ext4 varies insignificantly (4%) under the web server workload. The large variance numbers indicate that having the right guest/host file system combination is critical to performance, and having a wrong combination can result in serious performance degradation. For instance, under the database workload, ReiserFS/Ext2 is a right combination, but ReiserFS/JFS is a wrong combination.

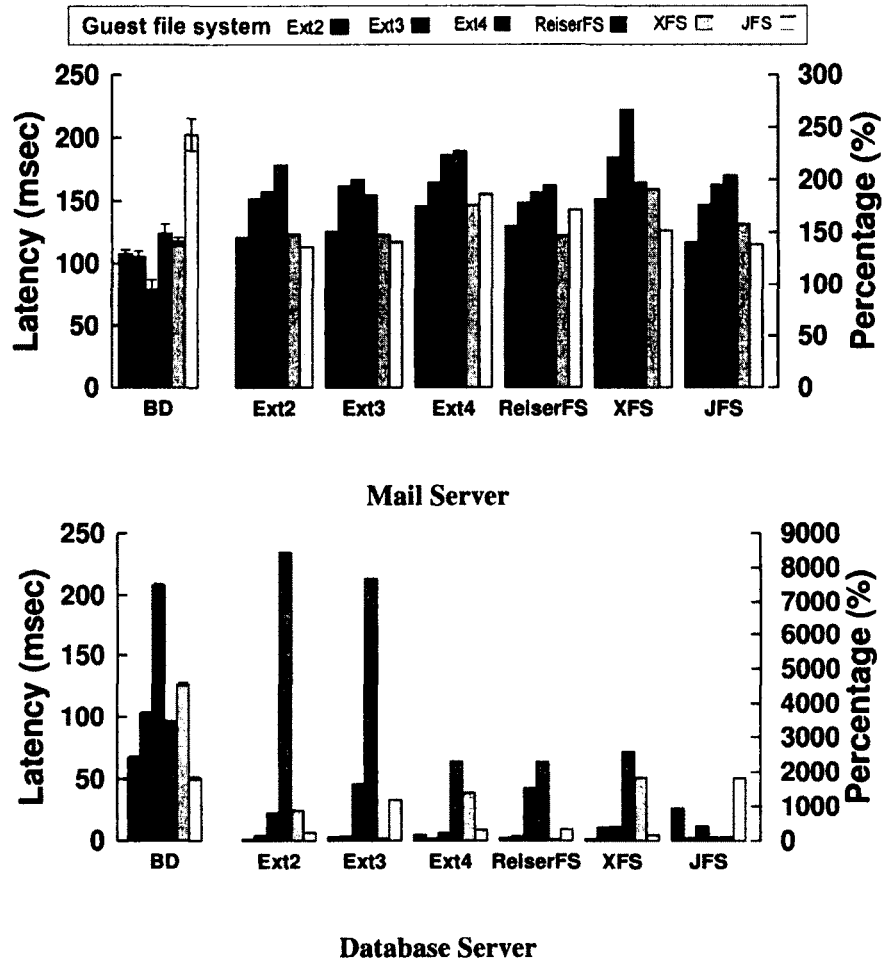


Figure 3.8: I/O latency of guest file systems under different workloads (lower is better)

- A host file system impacts different guest file systems' performance differently. Similar to the previous observation, a host file system can have a different impact on different guest file systems' performance. Figure 3.3 shows an example of the file server workload. When Ext2 runs on top of Ext3, its throughput is slightly degraded by about 10%. However, when Ext3 runs on top of Ext3, the throughput is reduced by 40%. Based on results of coefficient of variance of guest file systems' throughputs shown in Figure 3.5, we observe that this bi-directional dependency between guest and host file systems again stresses the importance of

	File server		Web server		Mail server		Database	
Guest File Systems	Best	Worst	Best	Worst	Best	Worst	Best	Worst
Ext2	137	180	89	111	141	182	150	162
Ext3	133	227	164	319	176	222	133	197
Ext4	121	167	185	251	188	267	149	241
ReiserFS	111	152	164	185	185	228	137	263
XFS	123	151	108	167	147	191	126	153
JFS	123	154	107	133	136	186	198	232

Table 3.5: Best and worst case I/O latency (relative to baseline) of each guest file system across different host file systems (%).

choosing the right guest/host file system combination.

- **A right guest file system/host file system combination can produce minimal performance degradation.** Also based on results shown in Figure 3.5, one can also observe how badly performance can be impacted when a wrong combination of guest/host file system is chosen. However, it is possible to find a guest file system whose performance loss is the lowest. For example, the results of the mail server workload show that once Ext2 runs on top of Ext2, its throughput degradation is the lowest (by 46%).
- **The performance of nested file systems is affected much more by write than read operations.** As one can see in Figure 3.4, *all* the combinations of nested file systems perform poorly for the mail server workload, unlike the other three workloads. We study the detailed disk traces from these workloads by examining request queuing time, request merging, re-

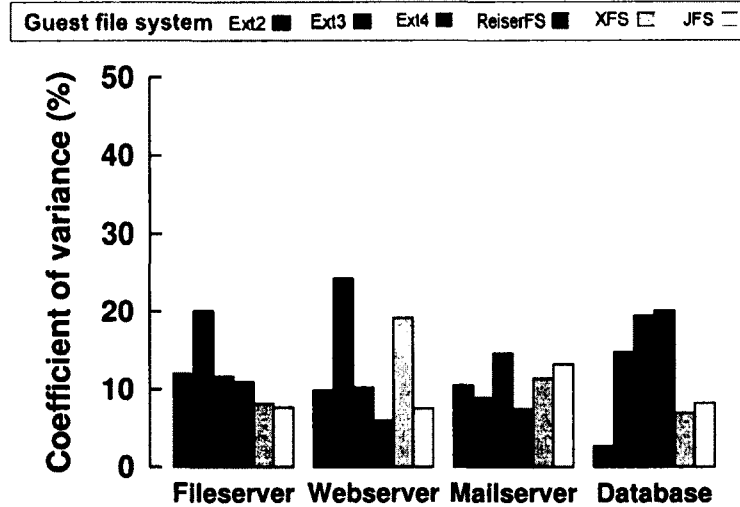


Figure 3.9: Coefficient of variance of guest file system's latency across different host file systems.

quest size, etc., and find that the mail server workload is only significantly different from the others in having a much higher proportion of writes than reads, as shown in Figure 3.6. We will use micro-benchmarks in Section 3.4 to describe the reasons behind this behavior.

3.3.3.2 Latency

The latency results are illustrated in Figures 3.7 and 3.8. Similar to I/O throughput, latency is also deteriorated when guest file systems are provisioned on top of host file systems rather than raw partitions. Whereas the impact to throughput can be minimized (for some workloads) by choosing the right combinations of guest/host file system, latency is much more sensitive to nesting of file systems. In comparison to the baseline, the latency of each guest file system varies in a certain range when it runs on top of different host file systems. Even for the lowest cases, latency is increased by 5-15% across the board (e.g., Ext2 guest file system under the web server workload). Coefficient of variance for latency, as shown in Figure 3.9, is similar to that of throughput shown in Figure 3.5.

However, for latency sensitive workloads, like the database workload, such a significant increase in I/O response time could be unacceptable.

3.4 Micro-benchmarks Results

We first study nested file systems using a micro-level benchmark *FIO* [9]. Based on the experimental results, we further conduct an analysis at the block layer on the guest VM and the hypervisor, respectively, using an I/O tracing mechanism [5].

3.4.1 Benchmark

We use *FIO* as a micro-level benchmark to examine disk I/O workloads. As a highly configurable benchmark, *FIO* defines a test case based on different I/O transaction characteristics, such as total I/O size, block size, number of I/O parallelism, and I/O mode. Here our focus is on the performance variation of primitive I/O operations, such as *read* and *write*. With the combination of these I/O operations and two I/O patterns, *random* and *sequential*, we design four test cases: random read, random write, sequential read, and sequential write. The specific I/O characteristics of these test cases are listed in Table 3.6.

3.4.2 Experimental Results

On the same testbed, the experiments are conducted with many small files, which create a 5GB of total data footprint for each workload. Figures 3.10 and 3.11 show the performance in both sequential and random I/Os. Based on the experimental results, we make two observations:

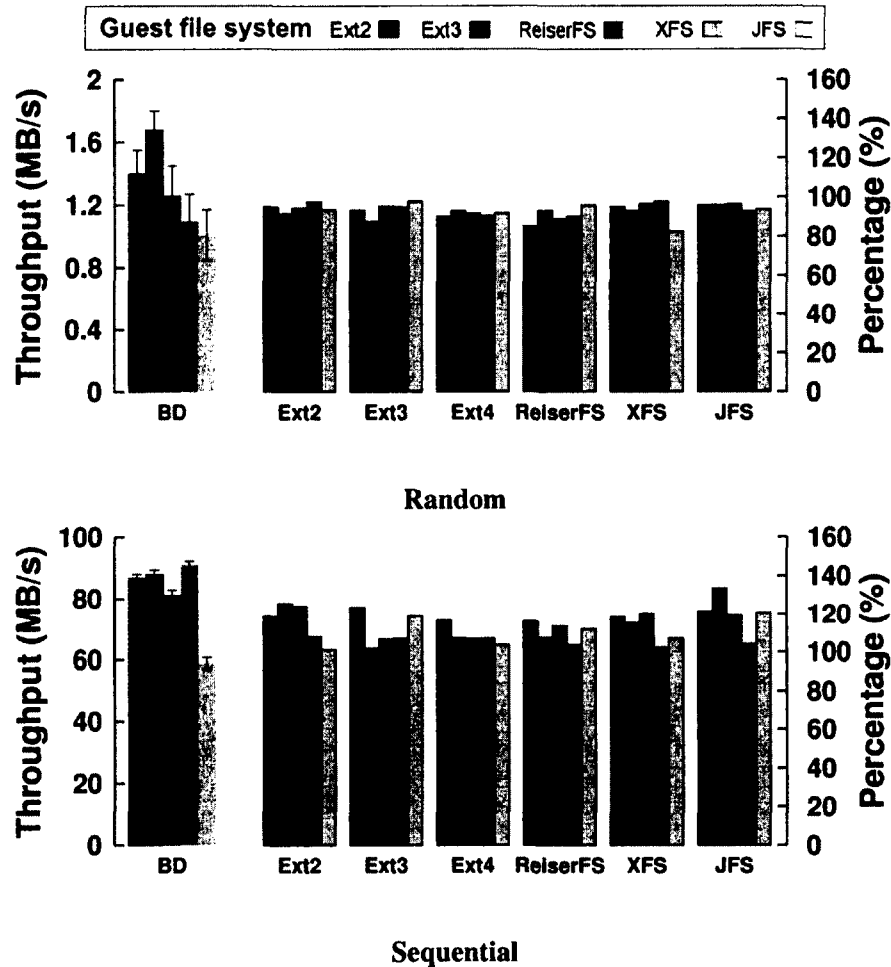


Figure 3.10: I/O throughput of guest file systems in reading files

- The performance of those workloads that are dominated by read operations is largely unaffected by nested file systems. The performance impact is weakly dependent on guest/host file systems. More interestingly, for sequential reads, in a few scenarios, a nested file system can even improve I/O performance (e.g., by 34% for Ext3/JFS).
- The performance of those workloads that are dominated by write operations is heavily affected by nested file systems. The performance impact varies in both random and sequential writes, with higher variations in sequential writes. In particular, a host file system like

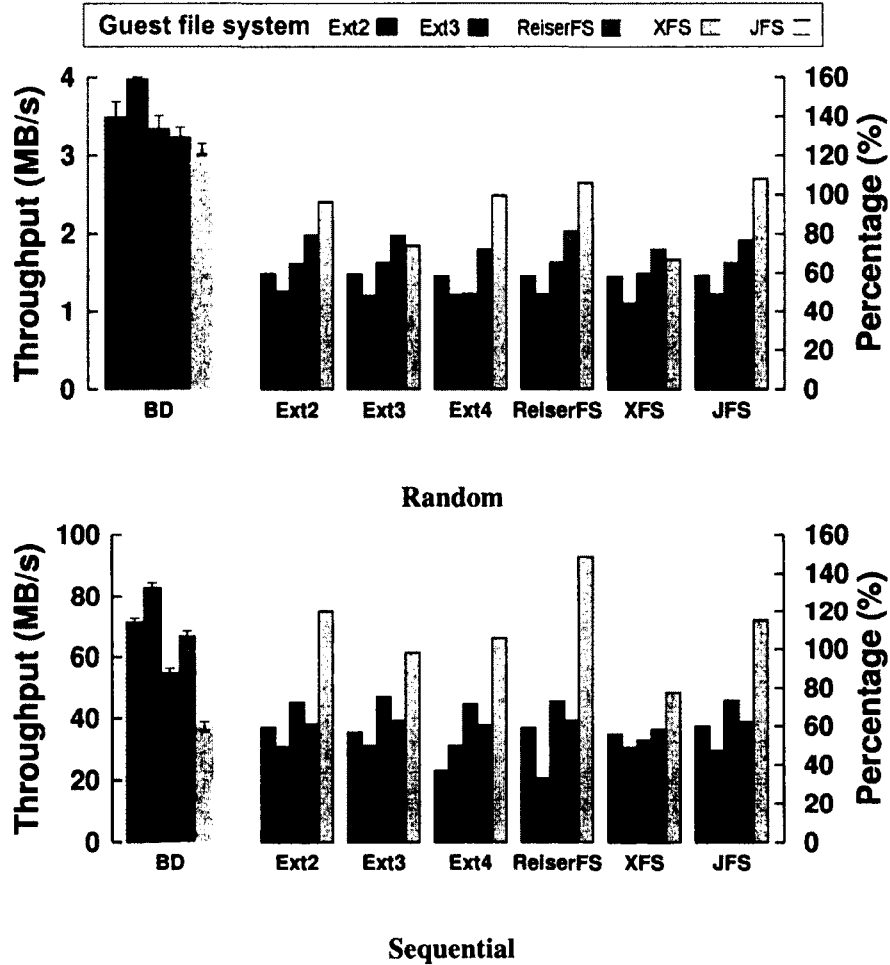


Figure 3.11: I/O throughput of guest file systems in writing files

XFS can degrade the performance by 40% for both random and sequential writes. As a result, it is important to understand the root cause of this performance impact, especially on the sequential write dominated workload.

To interpret these observations, our analysis will focus on sequential workloads and the performance implication across certain guest/host file system combinations. For this set of experiments with micro-benchmark, due to space constraints, we only concentrate on deciphering the I/O behavior of these representative file system combinations. Although only a few combinations are

Description	Parameters
Total I/O size	5 GB
I/O parallelism	255
Block size	8 KB
I/O pattern	Random/Sequential
I/O mode	Native asynchronous I/O

Table 3.6: FIO benchmark parameters

considered, principles used here are applicable to other combinations as well.

For sequential read workloads, we attempt to uncover the reasons behind the significant performance improvement on the *right* guest/host file system combinations. We select the combinations of **Ext3/JFS** and **Ext3/BD** for analysis. For sequential write workloads, we try to understand the root cause of the significant performance variations in the scenarios of (1) different guest file systems running on the same host file system and (2) the same guest file system operating on different host file systems. We analyze three guest file system/host file system combinations: **Ext3/ReiserFS**, **JFS/ReiserFS**, and **JFS/XFS**. Here **Ext3/ReiserFS** and **JFS/ReiserFS** are used to examine how different guest file systems can affect performance differently on the same host file system, while **JFS/ReiserFS** and **JFS/XFS** are used to examine how different host file systems can affect performance differently on the same guest file system.

3.4.3 I/O Analysis

To understand the underlying cause of the performance impact due to nesting of file systems, we use blktrace to record I/O activities at both the guest and hypervisor levels. The resulting trace files are stored on another device, thus increasing only 3-4% CPU utilization. Therefore, the interference with our benchmarks from such an I/O recoding is negligible. Blktrace keeps detailed account of each I/O request from start to finish as it goes through various I/O states (e.g., put the request onto an I/O queue, merge with an existing request, and wait on the I/O queue). The I/O states that are of interest to us in this study are described as follows.

- Q: a new I/O request is *queued* by an application.
- I: the I/O request is *inserted* into an I/O scheduler queue.
- D: the I/O request is being served by the *device*.
- C: the I/O request has *completed* by the device.

Blktrace records the timestamp when an I/O request enters a new state, so it is trivial to calculate the amount of time the request spends in each state (i.e., Q2I, I2D, and D2C). Here Q2I is the time it takes to insert/merge a request onto a request queue. I2D is the time it takes to idle on the request queue waiting for merging opportunities. D2C is the time it takes for the device to serve the request. The sum of Q2I, I2D, and D2C is the total processing time of an I/O request, which we denote as Q2C.

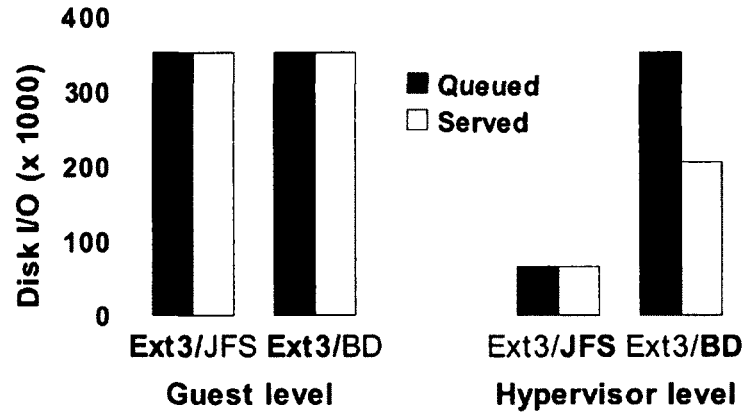


Figure 3.12: Disk I/Os under sequential read workload

3.4.3.1 Sequential Read Workload

As mentioned in the experimental setup, the logical block device of the guest VM can be represented as either a flat file or a physical raw disk partition at the hypervisor level. However, the different representation of the guest VM's block device directly affects the number of I/O requests served at the hypervisor level. For the selected combinations of **Ext3/JFS** and **Ext3/BD**, as Figure 3.12 shows, the number of I/O requests served at the hypervisor's block layer is significantly lower than that at the guest's block layer. More specifically, if JFS is used as a host file system, it greatly reduces the number of queued I/O requests sent from the guest level, resulting in much fewer I/O requests served at the hypervisor level than those at the guest level. If a raw disk partition is used instead, although there is no reduction on the number of queued I/O requests, the hypervisor level's block layer also lowers the number of served I/O requests by merging queued I/O requests.

There are two root causes for these I/O behaviors: (1) the file prefetching technique at the hypervisor level, known as *readahead*, and (2) the merging activities at the hypervisor level introduced by the I/O scheduler. The detailed descriptions of these root causes are given below.

First, there are frequent accesses to both files' content and metadata in a sequential read dominated workload. To expedite this process, readahead I/O requests are issued at the kernel level of both the guest and the hypervisor. Basically, readahead I/O requests populate the page cache with data already read from the block device, so that subsequent reads from the accessed files do not block on other I/O requests. As a result, it decreases the number of accesses to the block device. In particular, at the hypervisor level, a host file system issues readahead requests and attempts to minimize the frequent accesses on the flat file by caching the subsequently accessed contents and metadata in the physical memory. Therefore, the I/Os served at the hypervisor level are much fewer than those at the guest level.

However, when accessing a raw disk partition, there is no readahead. Thus, for sequential workloads, a host file system outperforms a raw disk partition due to more effective caching. This discrepancy of data caching at the hypervisor level is clearly shown in Figure 3.13.

Second, to optimize I/O requests being served on the block device, the hypervisor's block layer attempts to reduce the number of accesses into the block device by sorting and merging queued I/O requests. However, when many I/O requests are sorted and merged, they need to stay longer in the queue than normal. For JFS (host file system), as shown in Figure 3.12, due to the effective caching, much fewer I/O requests are sent to the disk, and thus much fewer sorting/merging activities occur at the I/O queue. However, when a raw partition is used, much more I/O requests need to be sorted/merged. The sorting/merging activities cause a higher idle time (I2D) for I/O requests being served on the block device than those on the JFS (host file system). This behavior is depicted in Figure 3.14 (hypervisor level).

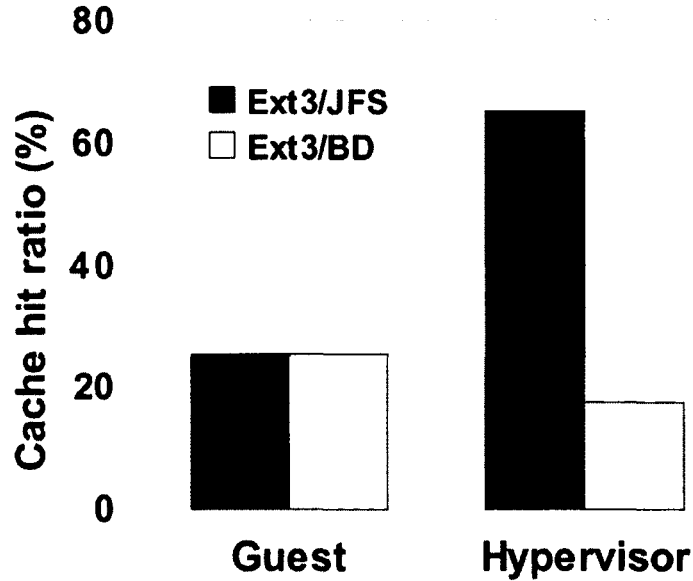


Figure 3.13: Cache hit ratio under sequential read workload.

Remark: When a flat file is used as a guest VM’s logical block device, sequential read dominated workloads can take advantage of the readahead at the hypervisor, achieving effective data caching. In contrast, when a disk partition is used, there is no readahead and data caching. Therefore, for all file systems, to gain high I/O performance, we recommend cloud administrators to select a flat file over raw partitions for services dominated by sequential reads.

3.4.3.2 Sequential Write Workload

Our investigation uncovers the root causes of the nested file systems’ performance dependency under a sequential write workload in two cases: (A) two file system combinations hold the same host file system, and (B) two combinations hold the same guest file system. The analysis detailed below focuses on two principal factors: sensitivity of an I/O scheduler and effectiveness of block allocation mechanisms.

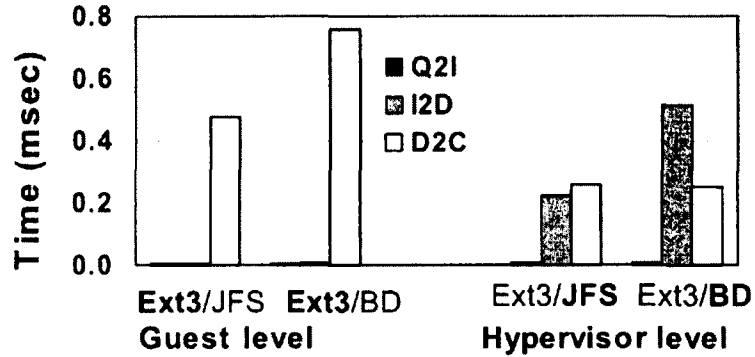


Figure 3.14: I/O times under sequential read workload.

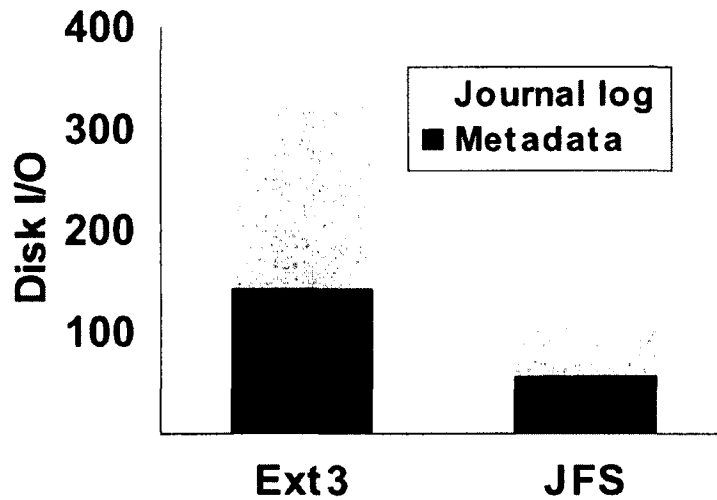


Figure 3.15: Extra I/O for journal log and metadata updates under sequential write workload.

A. Different guests (Ext3, JFS) on the same host (ReiserFS): As shown in Figure 3.11, we can see that the performance of sequential I/Os of Ext3/ReiserFS is much worse than that of Ext3/BD, while the I/O performance of JFS/ReiserFS is much better than JFS/BD. At the guest level, we analyze the performance dependency of Ext3 and JFS based on the comparison of their I/O characteristics. The details of this comparison are shown in Figure 3.16.

Figure 3.16 (A) shows that most I/Os issued from Ext3 and sent to the block layer are well merged at the guest level's I/O scheduler. The effective merging of I/Os significantly reduces the

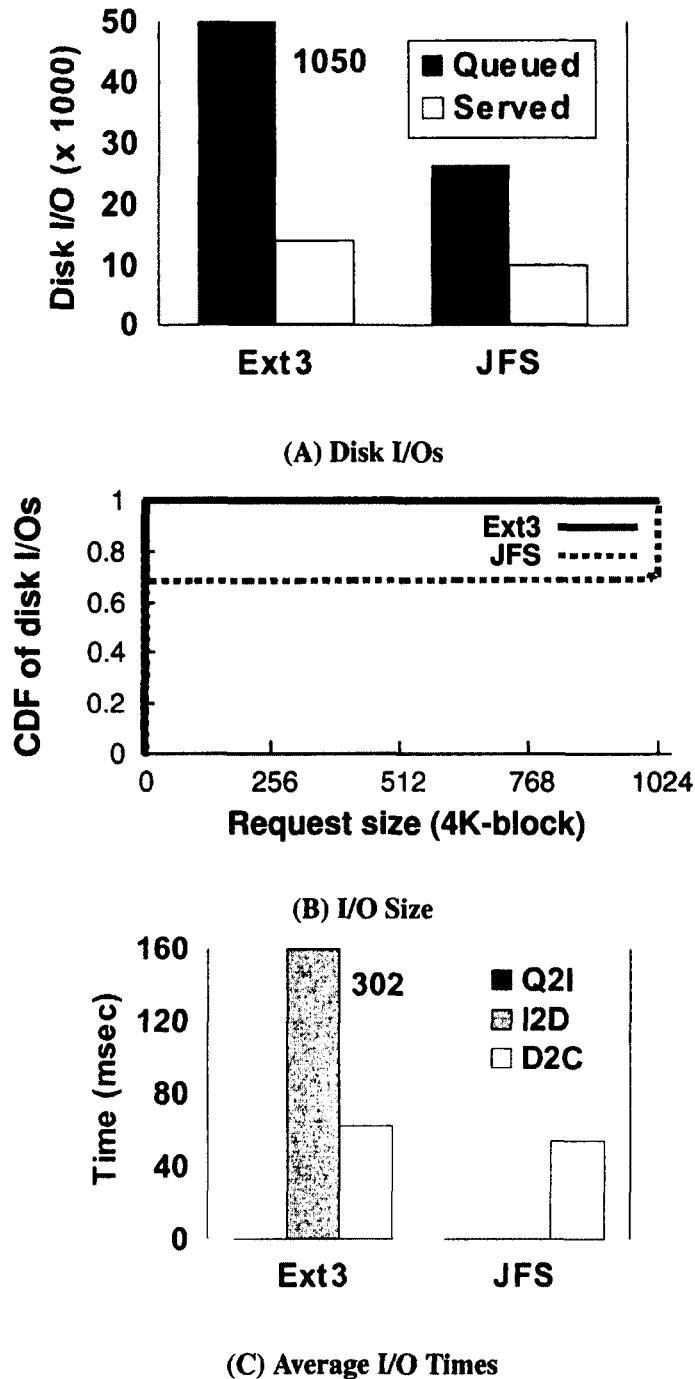


Figure 3.16: I/O characteristics at guest level of JFS/ReiserFS

number of I/Os to be served on Ext3 (guest). Meanwhile, Figure 3.16 (B) shows that 99% I/Os of Ext3 are in small size (8K) and those of JFS is 68%. Apparently, merging multiple small size I/Os

incurs additional overhead. This is because the small requests have to be waited longer in the queue in order to be merged, thus, increasing their idle times. This behavior is illustrated in Figure 3.16 (C).

To understand the root cause of merging happened on Ext3 and JFS (guest), we perform a deep analysis by monitoring every issued I/O activities at the guest level. What we found is that the block allocation mechanism causes this performance variation. To minimize disk seeks, Ext3 issues I/Os to allocate blocks of data on disk close to each other. The data includes regular data file, its metadata, and journal logs of metadata. This allocation scheme makes most I/Os be *back merged*. A back merge behavior denotes that a new request sequentially falls behind an exiting request on an order of the start sector, as they are logically adjacent. Note that two I/Os are logically adjacent when the end sector of one I/O is logically located next to the begin sector of the other I/O. As we can see, clustering adjacent I/Os facilitates the data access. However, it requires the issued I/Os to be waited longer in the queue for being processed.

JFS is more efficient than Ext3 in journaling. For regular data file written into disk, both Ext3 and JFS effectively coalesce multiple write operations to reduce the number of I/O committed into disk. However, for metadata and journal logs, instead of independently committing every single concurrent log entry as Ext3, JFS requires multiple concurrent log entries to be coalesced as one commit. For this reason, as shown in Figure 3.15, JFS has less I/Os spent for journaling, resulting in less performance degradation.

Remarks: The efficiency provided by the I/O scheduler's optimization is no longer valid for all nested file systems. Since file systems allocate blocks on disk differently, nested file systems have

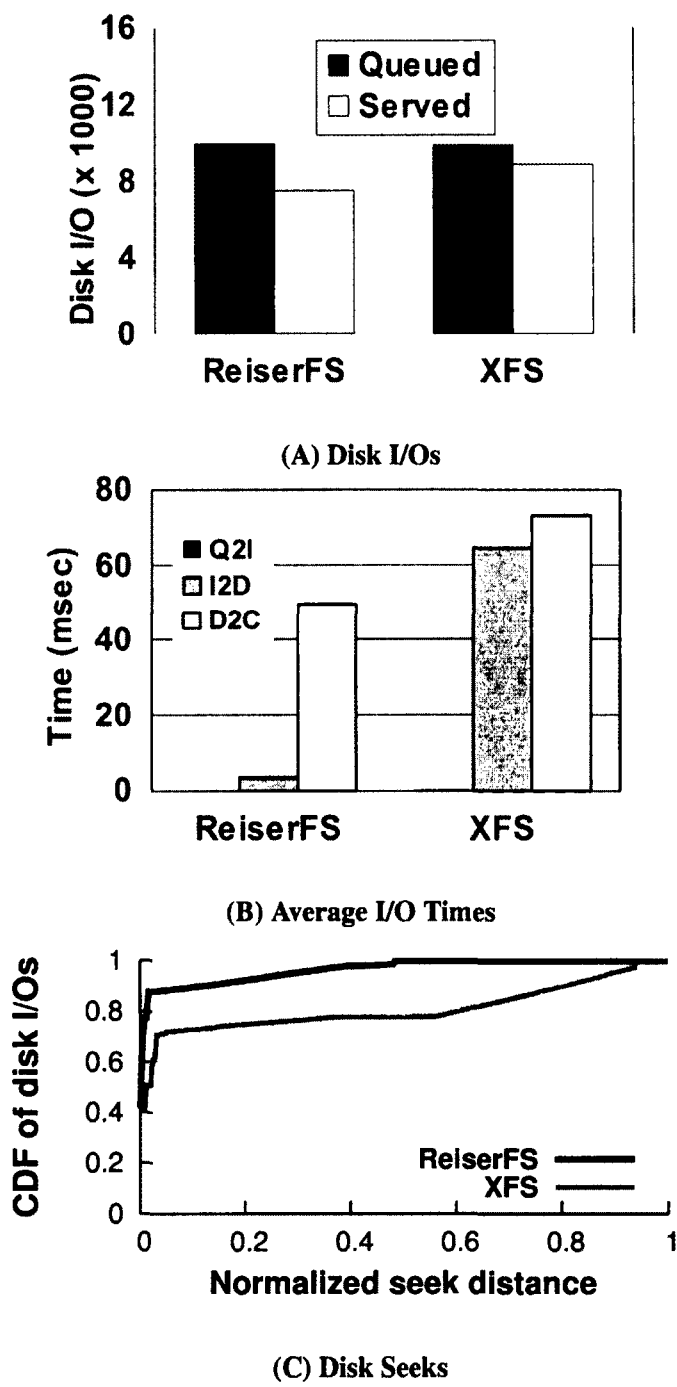


Figure 3.17: I/O characteristics at hypervisor level of ReiserFS/XFS.

different impacts on performance when one particular I/O scheduler is used. Therefore, a nested file system should be chosen based on the effectiveness of underlying I/O scheduler's operations on its

block allocation scheme.

B. Same guest (JFS) on different hosts (ReiserFS, XFS): Based on results of sequential writes shown in Figure 3.11, JFS (guest) performs better on ReiserFS than on XFS. We analyze I/O activities of these host file systems to uncover differences of their block allocation mechanisms. The detailed analysis is given below.

The analysis of I/O activities reveals that the I/O scheduler processes ReiserFS' I/Os similarly to those of XFS. As shown in Figure 3.17 (A), the number of host file systems' I/Os to be queued and served are fairly similar in ReiserFS and XFS. However, Figure 3.17 (B) denotes that XFS' I/Os are executed slower than those of ReiserFS. A further analysis is needed to explain this behavior. In general, file systems allocate blocks on disk differently, thus, resulting in a different execution time for I/Os. For this reason, we perform an analysis on the disk seeks. Based on the results shown in Figure 3.17 (C), we find that long distance disk seeks on XFS cause high overhead and reduce its I/O performance. Note that in Figure 3.17 (C), the x-axis is represented as a normalized seek distance and 1 denotes the longest seek distance of the disk head, from one end to the other end of the partition.

With respect to the case of one host file system allocates disk blocks more effectively than another under the same workload, we analyze the mechanisms to allocate disk blocks of ReiserFS and XFS and find that XFS induces an overhead because of a multiple journal logging. The detailed explanations are as follows:

A multiple logging mechanism of metadata also incurs an overhead on XFS. Basically, XFS is able to record multiple separate changes occurred on the metadata of a single file and store them

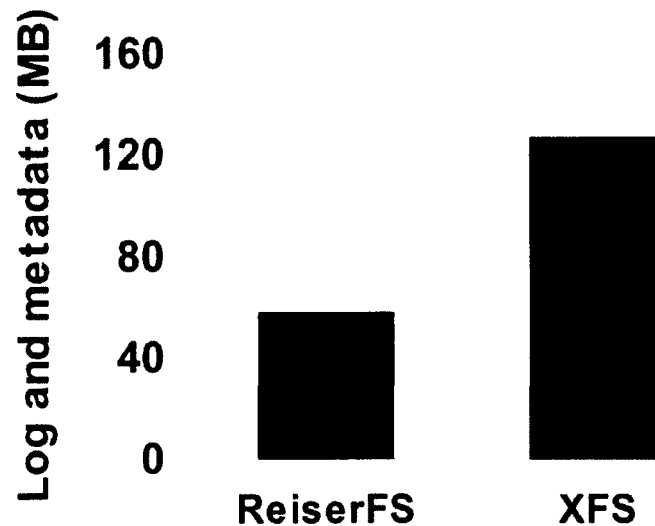


Figure 3.18: Extra data written into disk under the same workload from JFS (guest).

into journal logs. This technique effectively avoids such changes to be flushed into disk before another new change will be logged. However, every change of metadata can be range from 256 Bytes to 2 KB in size, while the default size of the log buffer is only 32 KB. Under an intensive write dominated workload, this small log buffer causes multiple changes of the file metadata to be frequently logged. As shown in Figure 3.18, this repeatedly logging produces extra data written into disk, thus, resulting in a performance loss.

Remarks: (1) An effective block allocation of one particular file system no longer guarantees a high performance when it runs on top of another file system. (2) Under an intensive write dominated workload, an update of journal logs on disk should be carefully considered to avoid performance degradation. Especially for XFS, the majority of its performance loss is attributed to not only a placement of journal logs, but also a technique to handle updates of these logs.

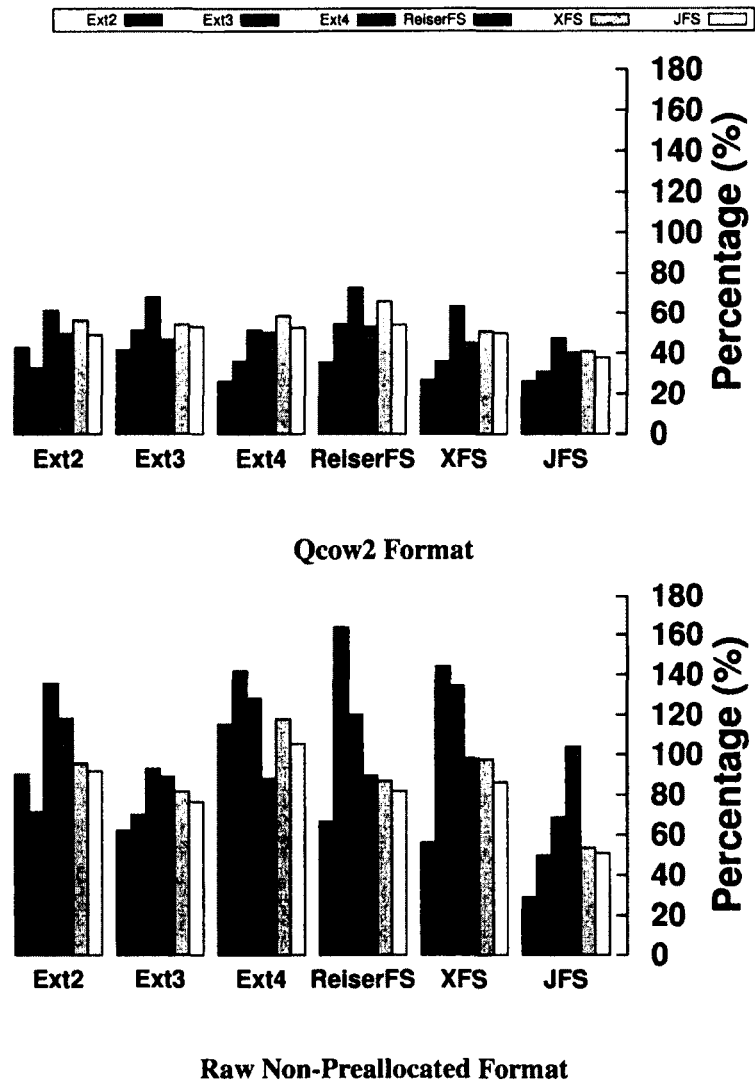


Figure 3.19: Latency of guest file systems under different formats of disk images.

3.4.4 Disk Image Formats

The logical block devices of a guest VM can be represented in other formats under the flat files, such as Qcow2, Raw non-preallocated or raw preallocated. As a native disk image file format, a Qcow2 file grows as needed. The more features provided by the Qcow2 over the Raw include base images, snapshots, compression, and encryption. In contrast, a Raw disk image needs to be allocated in the

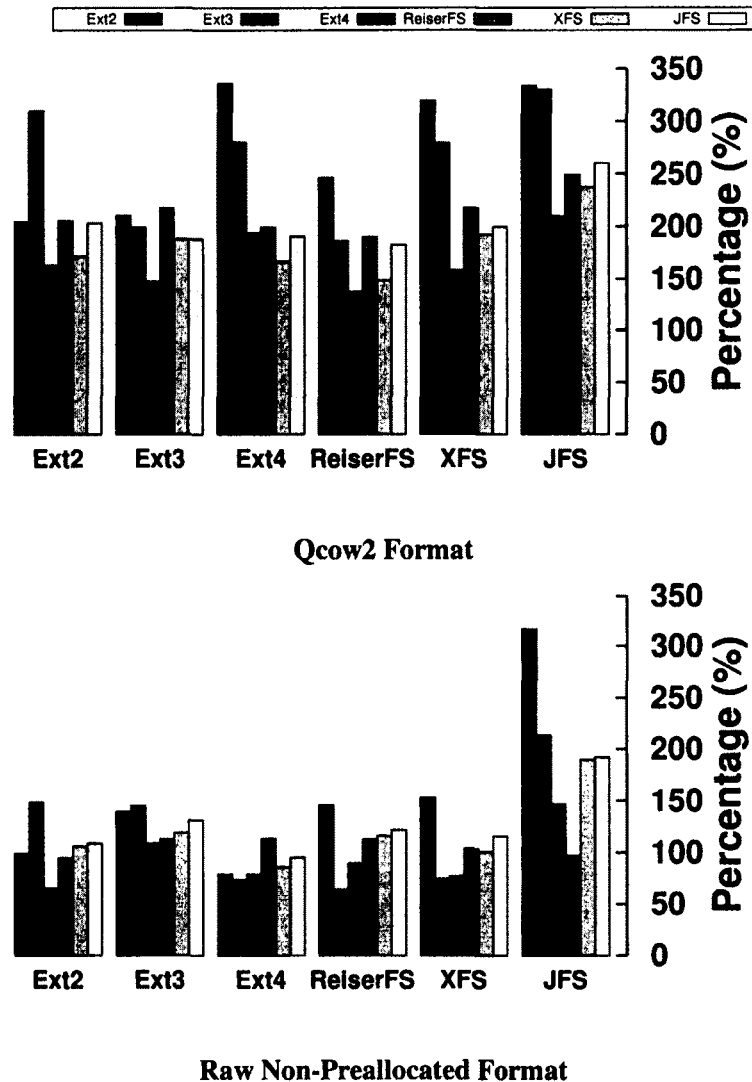


Figure 3.20: Latency of guest file systems under different formats of disk images.

full size beforehand. If a Raw sparse disk image is based on a regular sparse file, a Raw one is based on a non-sparse file whose empty data blocks are filled up by *null*, making its actual size on the disk generally smaller than its logical size. Since this Raw format is an exact bit-for-bit copy of a block device, its structure contains files and folders of stored data, and other components of a block device, such as a boot sector and file allocation tables.

To examine I/O behaviors when different disk image formats are used, we conduct a set of

experiments where the disk image is formatted as Qcow2, Raw sparse, and Raw respectively. Based on FIO benchmark parameters used for previous experiments, we create a workload. This workload simultaneously performs sequential reads/writes and creates a 5GB of data footprint. In total we test 36 different combinations of guest file system/host file system. Based on the experimental results shown in Figure 3.19 and Figure 3.20, we make two important observations:

A. Performance discrepancy between Qcow2 and Raw disk images: With the copy-on-write strategy on a fixed size block device, a Qcow2 disk image only reflects changes made on the underlying disk. Thus, it enables to efficiently maintain a small size of disk image. However, managing a small size of disk image also induces overhead, which produces negative impacts on the I/O performance. Unlike Qcow2, a Raw sparse disk image does not need to minimize its actual size on the block device. This is because blocks marked as null on the file image are simply filled up by data. Thus, it increases the actual size of the disk image on the block device. As a result, using Raw sparse image can achieve higher I/O performance for sequential workloads in nested file systems than using Qcow2.

B. Trade-off between performance and other storage features: Qcow2 disk images offer more features than Raw and Raw sparse disk images. However, based on the experimental results with different guest/host file system combinations, administrators of cloud storage systems should make a trade-off between performance and management when choosing disk format. For example, in a virtualization system that requires high I/O performance, the administrator should select Raw sparse rather than Qcow2 to format disk images. However, if the system requires a guarantee on reliability

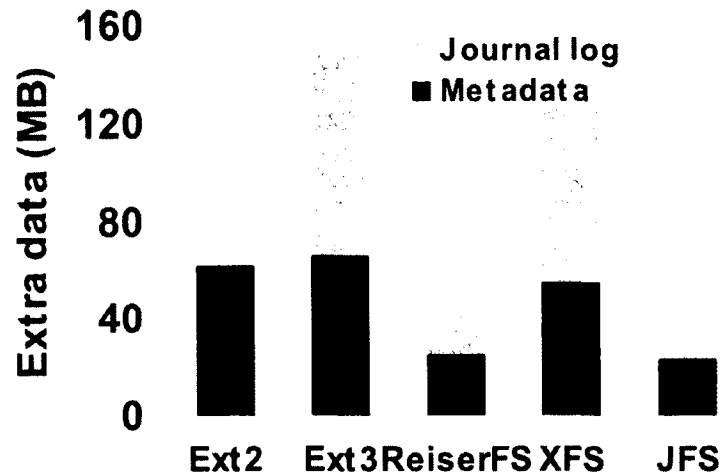


Figure 3.21: (hypervisor level) Extra data written into disk under a write-dominated workload from guest VM.

or security, instead of performance, for its services, the disk image should be formatted as Qcow2.

3.5 Discussion

Despite various practical benefits in using nested file systems in a virtualized environment, our experiments have shown the associated performance overhead to be significant if not configured properly. Here we offer five advice on choosing the *right* guest/host file system configurations to minimize performance degradation, or in some cases, even improve performance.

Advice 1 For workloads that are read-dominated (both sequential and random), using nested file systems has minimal impact on I/O throughput, independent of guest and host file systems. For workloads that have a significant amount of sequential reads, nested file systems can even improve throughput due to the readahead mechanism at the host level.

Advice 2 On the other hand, for workloads that are write-dominated, one should avoid using nested file systems in general due to (1) one more layer to pass through and (2) additional metadata update operations. If one must use nested file systems, journaled file systems in the host should be avoided. Journaling of both metadata and data can cause significant performance degradation, and therefore, is not practical to use for most workloads, and if only metadata is journaled, a crash can corrupt a VM image file easily, thus, giving no benefit to metadata-only journaling mode in the host. As shown in Figure 3.21, the additional metadata writes to the journal log can result in significantly more I/O traffic. Performance is even more impacted if the location of the log is placed far away from either the metadata or the data locations.

Advice 3 For workloads that are sensitive to I/O latency, one should also avoid using nested file systems. As shown in Figures 3.7 and 3.8, even in the best case scenarios, nested file systems could increase I/O latency by 10-30% due to having an additional layer of file system to traverse and one more I/O queue to wait for.

Advice 4 In a nested file system, data and metadata placement decisions are made twice, first in the guest file system and then in the host file system. Guest file system uses various temporal and spatial heuristics to place related metadata and data blocks close to each other. However, when these placement decisions reach the host file system, it can no longer differentiate between data and metadata and treats everything as data. As a result, the secondary data placement decisions made by a host file system are both unnecessary and less efficient than those made by a guest file system. Ideally, the host file system should simply act as a pass-through layer such as VirtFS [57].

Advice 5 In our experiments, we used the default set of formatting and mounting parameters in all the file systems. However, just like in a non-virtualized environment, these parameters can be tuned to improve performance. There are more benefits in tuning the host file system's parameters than guest's as it is ultimately the layer that communicates with the storage device.

One should tune its parameters in such a way that the host file system most resembles a “dumb” disk. For example, when a disk is instructed to read a small disk block, it will actually read the entire track or cylinder and keep them in its internal cache to minimize mechanical movement for future I/O requests. A host file system can emulate this behavior by using larger block sizes.

Metadata operations at host file system is another source of overhead. When a VM image file is accessed or modified, its metadata often has to be modified, thus, causing additional I/O load. Parameters such as *noatime* and *nodiratime* can be used to avoid updating the last access time without losing any useful information. However, when the image file is modified, there is no option to avoid updating the metadata. As the image file will stay constant in size and ownership, the only field in the metadata that needs to be updated is the last modified time, which for an image file is just pure overhead. Perhaps this can be implemented as a file system mount option. Note that journaling, as mentioned previously, in the metadata-only mode has very little usage in the host level.

Lastly, using more advanced file system features to configure block groups and B+ trees to perform intelligent data allocation and balancing tasks will most likely be counter-productive. This is because these features will cause guest file system's view of disk layout to deviate further from the reality.

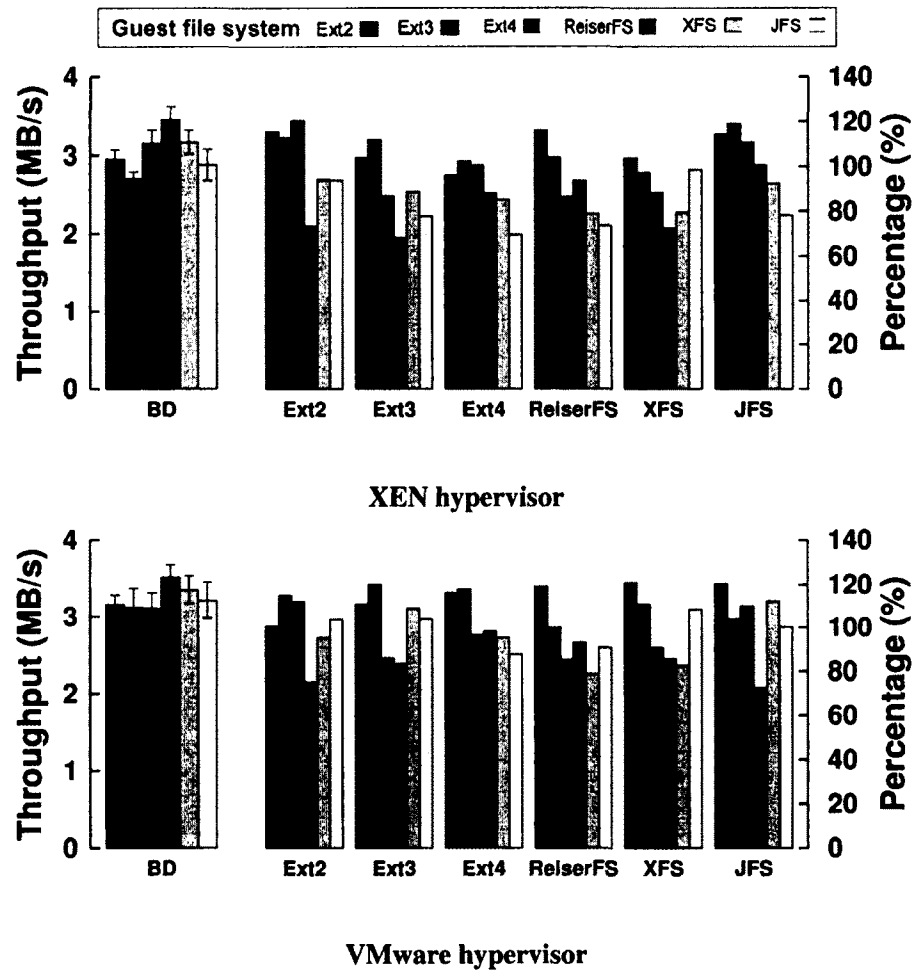


Figure 3.22: Other hypervisors show variation of relative I/O throughput of guest file systems under database workload (higher is better)

3.6 Summary

Our main objective is to better understand performance implications when file systems are nested in a virtualized environment. The major finding is that the choice of nested file systems on both hypervisor and guest levels has a significant performance impact on I/O performance. Traditionally, a guest file system is chosen based on the anticipated workload, regardless of the host file system. By examining a large set of different combinations of host and guest file systems under various

workloads, we have demonstrated the significant dependency of the two layers on performance, and hence, system administrators must be careful in choosing *both* file systems in order to reap the greatest benefit from virtualization. In particular, if workloads are sensitive to I/O latency, nested file systems should be avoided or host file systems should simply perform as a pass-through layer in certain cases.

We also have conducted experiments with the database workload to verify if the I/O performance of nested file systems is hypervisor-dependent. The chosen hypervisors are architecturally akin to KVM, such as VMware Player 3.1.4 with guest tools [27], and Xen 4.0 with Xen para-virtualized device drivers [32]. Figure 3.22 shows that the I/O performance variations of guest file systems on Xen and VMware are fairly similar to those on KVM.

Chapter 4

Shadow Patching: Minimizing Maintenance Window

4.1 Introduction

Cloud service providers allow customers to instantiate new virtual machines and manage their life-cycle on-demand to best suit business needs and budget constraints. It is always a challenge to strike the right balance between ensuring VMs in a secure and compliant state and needing to schedule downtimes to make such changes to running systems [44]. Depending on the nature of the change and the complexity of the affected applications, downtime (maintenance window) can extend from several hours to days. During maintenance windows, services and applications are first shutdown, and then changes are applied, for which, one or more system reboots might be required. Once changes are committed, various tests are performed to verify the system is still in a working state, i.e., functional, performance, scalability, etc., which can also be time consuming. During this time window, the services provided by the affected systems are often completely disrupted (unless high-availability mode is enabled, which might or might not be always possible depending on the type

of the service and the additional cost to enable high-availability.) To minimize such disruptions, some may choose to continue to operate on outdated software [73], resulting in high security risks. Ideally, this downtime should be minimized if not eliminated.

To eliminate change windows completely, various online patching techniques have been proposed [35, 64, 71]. However, these techniques are applicable to only specific OSes and applications. For example, Ksplice [35] allows Linux kernel patches to be applied to a running kernel without rebooting by changing in-memory data structures on the fly. Conceptually, the same approach can be applied to other operating systems, but unless their respective vendor or a third party is making a similar tool available, this will remain as a Linux-only tool. For a large enterprise or a Cloud provider where many different OSes are supported, this is not a general approach one can use across all the different systems.

We present a software patching framework, called Shadow Patching. Shadow Patching uses parallel virtual machine instances similar to Devirtualizable Virtual Machines (DVM) [64]. Unlike DVM which uses a highly customized VMM based on HP Alphaserver DS20, Shadow Patching will work with any commodity VMMs, e.g., KVM, Vmware, Xen, etc., and does not require any changes to VMM or guest VMs. The parallel VM instance is created for the duration when patches are applied, and is deprovisioned when finished, thus, not consuming resources during steady-state operations. The parallel VM instance is an exact replica of the VM to be patched in terms of OS, middleware, applications, and all the configurations. Thus, when a patch is applied, it will occur the same behavior on the parallel instance, or known as a cloned VM, as it would on the original VM. When a patch is applied to the parallel VM, we monitor all file system operations that change con-

tents on persistent storage. Changes to persistent storage can happen due to patching itself, system reboots, configuration file changes, and any other changes due to post-patching testing. These steps are done in the parallel VM while the original VM continues to run, and all file system changes in both VMs will be captured during this time. Once these steps are done, both VMs will need to be quiesced and shut down so that the deltas can be compared and merged. One would imagine there should be no conflicts between the two deltas as the delta from the parallel VM is a result of applying the patch and the delta from the original VM is a result of the running workload making changes to the application data, thus, merging of the deltas should be straightforward. However, in reality, there are many conflicts. We provide a default set of rules that would allow deltas to be merged correctly in common cases. Essentially, Shadow Patching removes conventional patch management operations from the critical patch (i.e., during maintenance window). This allows unexpected problems to be resolved outside of the maintenance window and transforms a patching operation to a set of simple file merging operations.

Based on the similar technique used for VM cloning, Shadow Patching also enables administrators to leverage storage utilities to clone virtual disk images. Since a cloned disk image is hypervisor independent, a cloned VM can be launched on top of a different physical system, thus, resulting in no performance impact on the system that hosts the original VM. The main contributions of this paper are summarized as follows:

- We develop a simple but effective software patching framework. We demonstrate that our proposed mechanism always takes less time than traditional methods for software patching.
- In case of failure patches, our framework significantly lowers the maintenance window of

actual enterprise systems, since we can resolve the patching issues on the cloned system.

4.2 Related Work

The Shadow Patching framework is based on various approaches, which are related to online and offline system software patching techniques. Online patching techniques try to eliminate the system and/or services down time while guaranteeing system stability. However, it is challenging to build a generic online patching framework for both kernel and application level updates or it may require some significant modifications to system infrastructure in order to employ such techniques [52].

Arnold et al. propose a tool called Ksplice [35], which focuses on live Linux kernel update. Ksplices runs on the object code layer, and it transforms patches into hot updates and do not require system to be rebooted. However, Ksplice is only suitable for Linux kernel update, in which data structures do not change frequently. When it comes to applications, it is common that data structures are changed much more frequently, and in these cases, Ksplice requires software developers to write some new code to cope with the patches. In contrast, Shadow Patching is able to handle both application and kernel updates, and we do not need to write any additional code for each patch.

Another important form of online patching is dynamic software updating, which is the extreme form of online patching. In short, it does not require the program in question to be stopped and then restarted. This actually eliminates the downtime, however, existing techniques are designed for some specific programming languages, such as the applications written in C or C++ [34, 53, 65], or the operating systems written in C or C++ [37, 42, 66], or systems written in Java [82]. Subramanian et al. [82] propose JVOLVE, and their experiments demonstrate that JVOLVE is able to update

Jetty webservers, JavaEmailServer, and CrossFTP server. However, the commonality is, all these software are written in Java. So, while these dynamic software updating techniques are efficient in dealing with some specific programming languages, they cannot handle any other languages. In addition, these techniques are either able to handle applications updating, or able to cope with operating systems updating, but not both. By contrast, our framework does not depend on any specific programming languages, and it can handle both application and operating system updates.

There are also several other online software patching solutions, which count on the approach of migration. This migration approach provides an ability to preserve a running state of a system or a service, thus flexibly transferring them between old and new versions. As an example, parallel instance is a common technique that facilitates this migration. Besides the original instance, another instance is created to host migrated running services, so that software patches are able to be deployed on the original one. Basically, an instance can be a physical system [31], a VM [64, 71, 48], or a process [62, 48]. A limitation of these approaches is that when a patch changes the underlying data structure and/or interfaces, the changes will prevent the migration operations to properly function. Our approach uses mature virtualization techniques, e.g., VM snapshot and clone, and is hypervisor-agnostic, and thus can be adopted more widely. Additionally, the troublesome migration operation in previous work is transformed in our work to a simple offline data merging task, again, enhancing its practicality. This is done at the expense of requiring an explicit downtime. However, as most patches would require the affected software to be refreshed for the change to take effect, service interruption time is unavoidable in most cases anyway.

On the other hand, a patching system needs to be restarted [29] or stayed at dormant states [90] if

its software is offline patched. Even though offline patching techniques facilitate the software maintenance, these techniques impact on the software maintenance window, thus, reducing the availability of system and hosted services. Our approach, even though does not eliminate the maintenance window, will minimize its duration by hiding the time spent to apply patches, resolve any problems, perform regression tests so it is not visible from the user's perspective.

With respect to the offline software patching, different solutions are proposed. VMWare's vSphere Update Manager inserts patches into the software update process [29]. Microsoft's VMST wakes a dormant VM up, and then applies software patches on this system [15]. Nuwa requires rewriting installation scripts to apply software patches on a mounted disk image [90].

The solutions proposed in [29] and [15] do not guarantee the stability of a patched system. By stability, we mean that any conflicts caused by newly patched software can only be triggered after the system is rebooted. The Shadow Patching framework instead applies software patches on a cloned VM rather than on an original system, thus reducing impacts on the original system.

Nuwa of Zhou et. al [90] leverages Mirage [72], a storage mechanism for cloud environments, to apply software patches on dormant virtualized systems. The overlap between Nuwa and Shadow Patching is the requirement to analyze a virtual machine at the file system level to properly apply patches. Besides this similarity, the two frameworks are different in terms of goals and techniques being used.

First, we focus on keeping the maintenance window short for online VMs, whereas Nuwa focuses on keeping offline VMs up-to-date. Second, instead of using a cloned VM, Nuwa uses `chroot` to patch software on a mounted disk image and rewrites update scripts to resolve conflicts

and dependencies incurred by software patching. Last but not least, Nuwa cannot apply all software patches. This technique conceals a limitation, in which unsuccessful software update could happen. This is because some software patches can only be properly applied under a running system. Using disk cloning, Shadow Patching is able to avoid this limitation. In addition, the disk cloning technique eases maintenance jobs, such as rewriting system installation scripts, analyzing memory changes caused by update commands, and resolving conflicts caused by a replacement of software and dependencies. Finally, Shadow Patching allows the cloned VM to be restarted several rounds, instead of waiting for a next reboot to verify if patches are successfully applied. Thus, the system's stability and availability are guaranteed.

4.3 Technical Background

The focus of this research is to improve the effectiveness of software patching in commodity enterprise systems, especially in virtualized cloud environments. To better understand software patching in such environments, we first present the technical background and related challenges. Then, we briefly describe our chosen approach.

4.3.1 Software patching

Current software patching is centered on the replacement of software components at the level of file systems. As an advantage, this method allows old software components to be replaced with new ones. However, this method does not guarantee the stability of the recently patched system. The reason is that the majority of software patches are system dependent. Due to differences across

systems, software conflicts could occur, leading to system crashes. It has been proved that ideal software packages must be well tested before they are published. However, nearly 70% of software patches are buggy in their first released, regardless of the available techniques used to either self-fix or determine software faults [44, 73, 80]. For this reason, an enhanced software patching mechanism must balance its simplicity and effectiveness, while addressing the problems of buggy patches.

4.3.2 Virtual disk image and VM cloning

It has been well known that VM disk images facilitate the management and maintenance of VMs in cloud environments [72]. The use of such thinly provisioned files allows storage space overcommit or flexible snapshotting. The VM images based on the copy-on-write strategy not only optimize the storage space, but also simplify the snapshotting at the file level, making it easier than that at the block level. The particular disk image formats (e.g. QCoW or vmdk) support different types of snapshots, allowing deltas to be stored internally in the VM image or externally as separated files.

Cloning a VM is an effective method to deploy multiple VMs in cloud environments. By cloning, system settings of the VM, which include configured virtual devices, installed software, and other VM contents, are copied. Intuitively, when a VM is cloned, the resulting cloned VM is independent of the original VM. The changes made to the cloned VM are not reflected on the original VM, and vice versa. The cloned VM can either be a fresh boot, a replica of a template VM [1, 29], or a VM of the fork primitive (a parent VM copy itself) [63]. Cloning a running VM is more challenging than cloning a dormant VM, due to various system changes occurred, which must be handled, including unflushed I/Os, dirty memory, or CPU states. To manage all these changes, people have

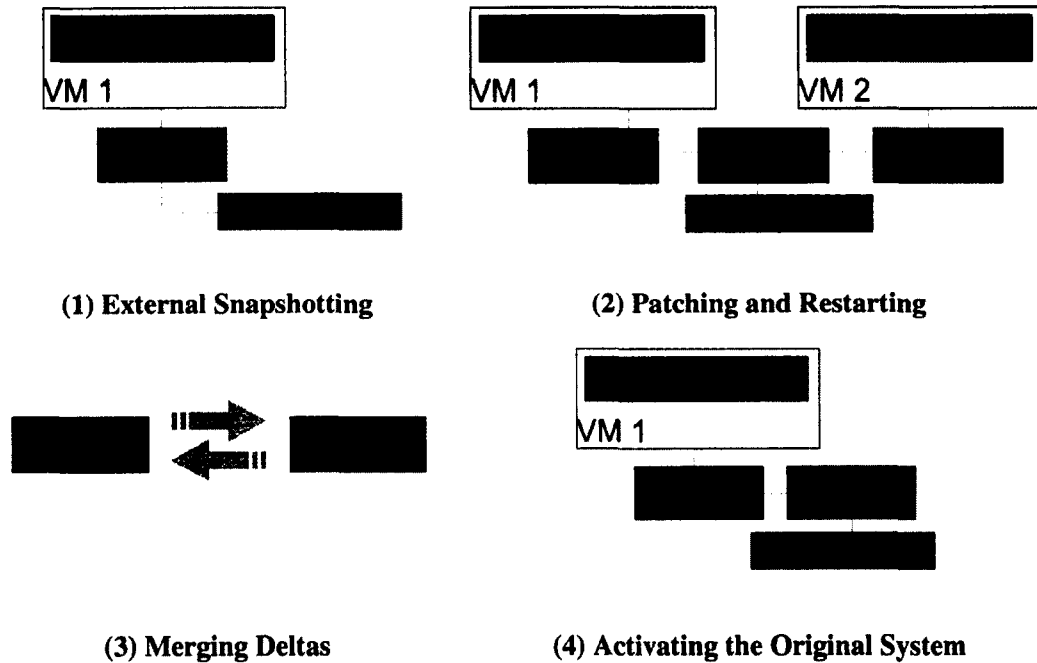


Figure 4.1: Scenario of Shadow Patching session

introduced sophisticated techniques that induce high overhead, thus, resulting in a negative impact on the original system's performance, as well as hosted services' down time.

4.3.3 Disk cloning approach

Our disk cloning approach takes advantage of cloning VM and leverages the feature of disk image snapshotting for software patching. As a straightforward approach, we concentrate on cloning disk images by using snapshotting technique, instead of using sophisticated techniques to clone an entire virtualized system. Compared to snapshotting of an entire system, snapshotting of a disk image only requires to handle ongoing I/Os, thus significantly reducing incurred overhead. In contrast, snapshotting of an entire system requires to manage I/Os and states of running system and applications, leading to much higher overhead.

In the snapshotting of disk image, as an available feature of CoW disk image formats, an external snapshot allows deltas to be immediately written into the newly created disk image, and the original disk image becomes a read-only base. Therefore, to clone a disk image of a running VM while guaranteeing a consistency between the original and cloned disks, the read-only base is cloned as soon as an external snapshot of the original disk image is taken. By consistency, we mean that data content and data structure at the file system level and block level of two disk images are identical. Once a clone of disk image is completed, a simulated VM is able to operate.

4.4 Shadow Patching Framework

4.4.1 Patching scenario

The Shadow Patching framework must be simple and effective, as well as generic. By generic, we mean that Shadow Patching should be able to conduct software patching on any virtualized system. Figure 4.1 illustrates a session of Shadow Patching software patching. Instead of scheduling this maintenance at a system's off-peak time, Shadow Patching allows the software maintenance of a virtualized system to perform on demand. The details of the session is described as follows:

- **External snapshotting.** As a virtualized enterprise server, VM1 includes a running OS that hosts multiple enterprise application services. By default, VM1 is based on a freshly installed OS, which is stored at a preserved read-only base image. When VM1 is running, its changes are maintained and stored in *Deltas-0*. *Deltas-0* functions as a disk layer created on top of the base image. As soon as software of VM1 need to update, an external snapshot of *Deltas-*

0 is taken, thus creating *Deltas-1* on top of Deltas 0. Therefore, Deltas-0 becomes another read-only base, while the changes of VM1 are written into Deltas-1.

- **Patching and restarting.** To clone VM1, *Deltas-2* is created on top of Deltas-0. A cloned VM1, called VM2, is created based on Deltas-2. Since VM1 and VM2 share the same base image Deltas-0, the changes of VM2 are independently stored on Deltas-2 without producing any impact on Deltas-1. This independent storage allows VM2 to be properly activated. Once VM2 is activated, it is able to install software upgrades to the OS, apply software patches, set up a new version of the application, or reconfigure other software components. After software patches are applied and configuration settings are adjusted, VM2 can be restarted and then we can conduct the thorough tests on the patched software and new settings.
- **Merging deltas.** At the appointed time, as soon as both VMs are dormant, the changes of Deltas-1 and Deltas-2 are merged. Merging deltas denotes that the software components generated on Deltas-2, due to the software patching on VM2, are copied back to Deltas-1 of VM1. It is known that when software is patched on VM2, VM1 is still running. Therefore, the conflicts induced by a modification on the same components at both systems could occur. To resolve such conflicts, we consider a hybrid method, which is detailed in Section 4.4.2.2.
- **Activating the original system.** Once VM1 starts, OS and applications are running with their new patches and configuration settings. For VM2, because it is no longer used, Deltas-2 can be discarded.

4.4.2 Software component replacement

Replacing software components or merging deltas, between Deltas-1 and Deltas-2, must guarantee consistency and effectiveness. By consistency, we mean that the changes at the level of file systems of Deltas-1 and Deltas-2 are well managed. By effectiveness, we mean that the induced overhead must be minimized. Here we present how deltas are managed in Shadow Patching.

4.4.2.1 Monitoring deltas

In terms of storage, deltas can be managed at two levels: disk block and file system. At the disk block level, the snapshots of a disk image handle the deltas as modified data blocks. These modified data blocks are exposed as the disk blocks of the VM's block device. Intuitively, by monitoring the changes taken place on every disk block, we are able to monitor deltas in a fine-grained manner. However, this fine-grained monitoring incurs high overhead. At the file system level, deltas include the modifications of data and metadata. In particular, I/O operations on files and directories, such as create, delete, move, or modify, and open, close, or access, cause data changes and metadata changes, respectively. Deltas can be determined by simply scanning the entire file system. However, a file system could consist of thousands of files and this scanning may be impractical and induce a significant overhead.

To ease the monitoring of deltas in Shadow Patching, *inotify* is used at the file system level [11] of VM1 and VM2. This technique is chosen because of its efficiency and accuracy. As a Linux kernel subsystem, *inotify* provides an interface between user and kernel levels to instantly capture exact changes occurred on the system's device node. Thus, this interface can precisely indicate the

modification of data and metadata of any files or directories. Next, we specify how inotify is used in Shadow Patching to monitor deltas.

First, inotify only focuses on the changes of data, instead of both data and metadata. To manage the changes of all files and directories of the file system, inotify recursively monitors I/O events of the system's root directory. Only four types of events are monitored, including `IN_MODIFY`, `IN_MOVE_FROM/TO`, `IN_DELETE`, and `IN_CREATE`. Once an event is triggered, inotify captures this timestamped event. The information of this event, including file or directory name, is stored in an individual delta file.

Files and directories are frequently modified due to system I/Os. To minimize overhead induced by monitoring files and directories, Shadow Patching's inotify maintains a list of non-scanning items. Specifically, those items include deltas and special system files, such as character/block devices, pipes, and sockets. Basically, the list is initiated based on the categorized system directories of the Linux system. When inotify monitors deltas, if any files or directories, which fall in the category of special system files, are detected, their names will be appended into the list. For example, the exclusive list consists of `/dev` directory because this directory includes all device files. However, since executable binary files are stored in `/bin`, the directory of `/bin` is not included in the list. Due to the limited number of files and directories stored in the exclusive list, in comparison with the brute force scanning of the entire system, the exclusive list helps to lower about 3% induced overhead.

Second, inotify must monitor every I/O event occurred on VM2. However, on VM1, inotify only needs to monitor I/O events after the snapshot Deltas-1 is taken, thus minimizing the size of

Rules	On Deltas-1	On Deltas-2	Time conditions	Decisions
1	Yes	No	$SS \leq T_1$	Delete on Deltas-1
2	No	Yes	$T_2 \leq SS$	Copy from Deltas-2 to Deltas-1
3	Yes	Yes	$T_1 \leq SS$ and $SS \leq T_2$	Copy from Deltas-2 to Deltas-1
4	Yes	Yes	$SS \leq T_1$ and $SS \leq T_2$	Hybrid copy from Deltas-2 to Deltas-1

Table 4.1: Time base comparison between files/directories. SS : Snapshot time, T_1 : Last modification time of file on Deltas-1, T_2 : Last modification time of file on Deltas-2

deltas. Note that the captured file system events are time stamped, to guarantee the consistency of captured I/O events, the system timers of VM1, VM2, and the host must be synchronized.

4.4.2.2 Merging deltas

The underlying technique of merging deltas between Deltas-1 and Deltas-2 is the proper replacement of files and directories between the file systems of VM1 and VM2. Based on the delta files and exclusive list provided by inotify, Shadow Patching can determine the modified files and directories. In particular, the focus of this merging is to decide whether or not the files or directories in Deltas-1 should be kept, deleted, or replaced. The rule is based on the modification time of the file or directory and the snapshot time of VM1. Here, the modification time of a file or directory can be retrieved from its inode. Based on these estimated times, Table 4.1 lists the rules made on those files or directories. The details of these rules are described as follows:

- **Rule 1:** A file on VM1 has not been accessed or modified since the snapshot time. If the cloned version of this file on VM2 is deleted by a software update, the file must be deleted.
- **Rule 2:** A file on VM2 is freshly created by a software update. It must be copied back to VM1.
- **Rule 3:** A file on VM1 has not been accessed or modified since the snapshot time. However, the cloned version of this file on VM2 is modified because of a software update. Thus, the file must be replaced by its newer version, which is copied back from VM2.
- **Rule 4:** A file on VM1 and its cloned version on VM2 are modified after the snapshot time. These two files must be kept on VM1 after the merging by performing a *hybrid* copy. Basically, a hybrid copy consists of three steps: (1) Renaming those two files based on their inode information, so that their names are different. (2) Copying a newly renamed file from VM2 to VM1. And (3) creating a symbolic link on VM1 based on the original name of the file. To guarantee that the freshly copied file will be used once VM1 starts, the symbolic link is linked to the newly copied file rather than its original version.

4.4.3 Prototype of Shadow Patching

A working prototype of Shadow Patching is built on Linux systems, which supports software maintenance of Linux distributions.

Shadow Patching requires disk images that host VMs to be formatted as QCoW/QCoW2, instead of raw. As a copy-on-write data structure, a QCoW disk can be externally snapshotted without impacting on I/O performance of a running VM. This feature cannot be achieved on a raw disk. Note

that a QCoW disk image induces a larger overhead than a raw one. This is because a QCoW disk must allocate new clusters of data blocks once the disk needs to grow [20]. For a small disk, converting back and forth between QCoW and raw formats when systems are dormant may avoid such an overhead. However, for a large disk (e.g., hundreds of GBs), the conversion will significantly increase the system downtime.

To merge deltas, QCoW disk images are exposed as mount points at the file system level of the host machine. Different storage utilities can be used to leverage this mounting feature, such as `kvm/qemu-nbd` [61] for QCoW/QCoW2, `Vmtoolsd` [26] for `vmdk`, and `losetup` [21] for raw formats. Shadow Patching benefits from `kvm/qemu-nbd` that includes two components: client and server. As a kernel module, the client handles requests passed through the device node. These requests are forwarded to the server that stays at the user level. Then, the server processes the requests in order to access the data resided in QCoW disks.

While our prototype of Shadow Patching works to maintain software of Linux systems, the principles and considerations are applicable to other systems, such as Microsoft Windows. A feasible extension is discussed in Section 4.6.

4.5 Experimentation

4.5.1 Experimental setup

To evaluate Shadow Patching, we use 2 metrics: correctness and the size of the maintenance window required. To verify correctness, we run application-specific benchmarks after patch deployment to check if the patched software has the right version and its functions and performance are as expected.

	Hardware	Software
Host	Pentium D 3.4GHz, 1TB SATA, 2GB RAM	Ubuntu 11.10, kernel 3.0.0-12, libvirt 0.9
Guest	Qemu 0.14.1, 1GB RAM	Ubuntu (10.04, 11.04)

Table 4.2: Testbed setup

Additionally, as patching in Shadow Patching is transformed to file compare, replace, merge, and delete operations, we scan file systems to verify all files and directories associated with a patch are correctly placed. Maintenance window is another key metric. We compare Shadow Patching with traditional patch management method for both success and failure scenarios.

The software and hardware configurations of our test machine are shown in Table 4.2. All experiments are performed within the virtual machines on the same hypervisor. In the next section, we first compare Shadow Patching with traditional method for applying individual software patches to contrast the two methods in the common path where patches are successfully applied. In Section 4.5.3, we apply various service packs containing hundreds of individual patches and compare the two methods when failures occur. For all experiments, we maintain all software patches in a local repository so as to avoid possible variations in results due to network fluctuations.

4.5.2 When Patch Succeeds

In traditional software patching practice, a maintenance window is scheduled for making changes to running systems, e.g., patching. The action of applying a patch (usually would succeed) takes only a few minutes. However, running a regression test and/or resolving any unexpected problems would take much longer amount of time. Thus, maintenance windows are usually scheduled to range from

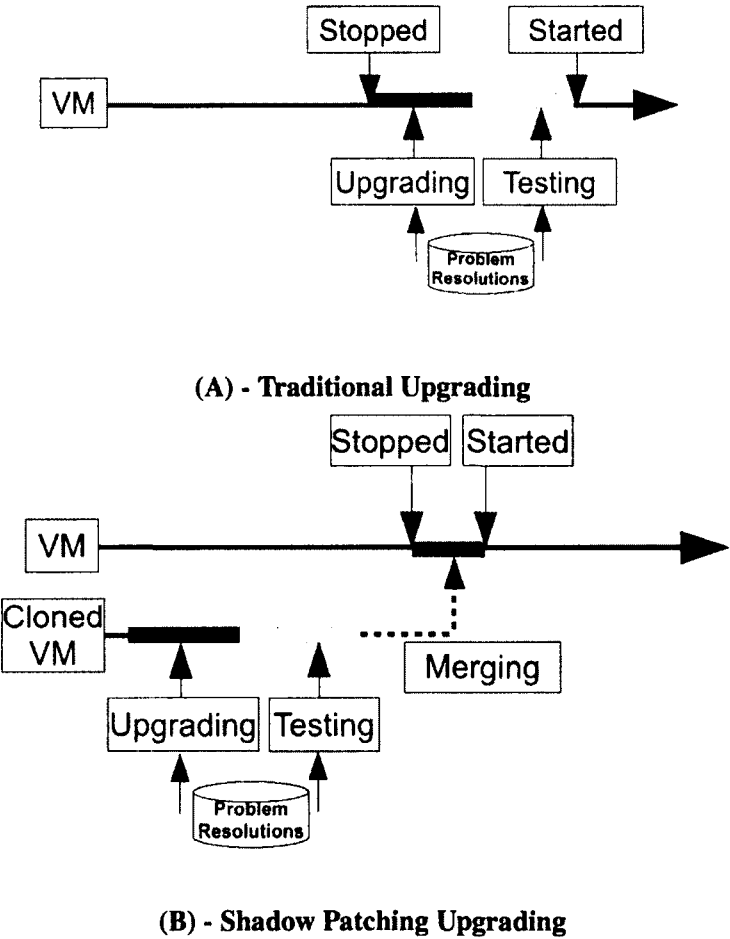


Figure 4.2: Upgrading process of application service

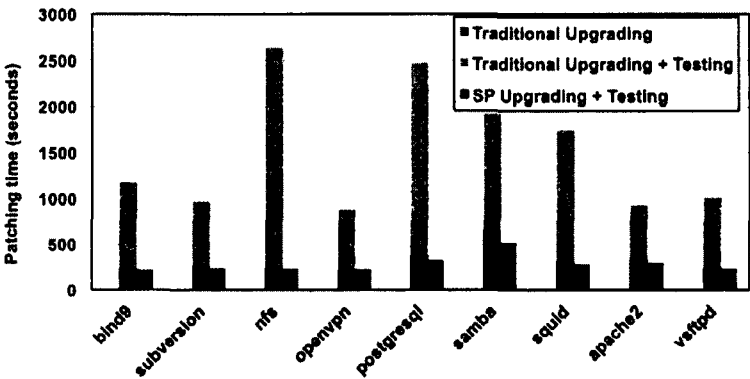


Figure 4.3: Individual upgrading application services (lower is better)

Services	Old Packages			New Packages			Overhead (%)	Benchmarks
	Versions	Sizes	# Files	Versions	Sizes	# Files		
Bind9	9.4.2	744	36	9.7.0	1.024	51	1.23	DnsPerf
SubVersion	1.4.6	3.400	28	1.6.6	4.204	35	1.30	Collabnet
NFS	1.1.2	504	35	1.2.2	640	43	1.01	IOzone
OpenVPN	2.1	1.060	86	2.1.3	1.208	93	1.34	NetPerf
PostgreSQL	8.3.16	13.884	95	8.4.9	14.804	92	1.40	PGbench
Samba	3.0.28	9.216	43	3.4.7	16.676	55	1.67	Dbench/Netbench
Squid	1.9	1.584	33	2.7	1.892	36	0.95	Web Polygraph
Apache2	2.2.8	4.356	492	2.2.14	8.864	564	1.86	Apache Bench
VsFTPD	2.0.6	396	41	2.2.3	460	44	1.23	Dkftpbench

Table 4.3: Upgraded application services and utilized benchmarks (Size in KB).

hours to days depending on the complexity of the patch. To allow sufficient amount of time to perform problem diagnosis and resolution, service providers are usually conservative in scheduling the maintenance windows. However, even if the entire window is not used, it would be difficult for users to salvage any of the remaining time to reduce services downtime as the patch completion time within the window is non-deterministic. This traditional process is illustrated in Figure 4.2(A), and Shadow Patching's is shown in Figure 4.2(B).

In our experiments, we use Ubuntu's `dpkg` to perform upgrades, or patching. Nine applications, listed in Table 4.3, are selected to be patched. As mentioned previously, one method to ensure correctness post-patch is by running application-specific regression tests, for which, we ran those commonly used benchmarks, which are also shown in the table. Besides a detailed specification of each application software, the column *Overhead* denotes results of overhead induced by inotify. Since inotify works as a part of the Linux's virtual file system, it only induces 1-2% overhead on system I/Os, which is fairly negligible.

First, we compare user perceived services downtime, which is shown in Figure 4.3. In traditional approach, the time it takes to apply the patch and perform regression test will all be visible to users. However, in the case of Shadow Patching, patching and testing occur in a separate cloned VM. This is completely hidden from users and can be done before maintenance window even starts. The downtime is only visible when we compare and merge disk deltas of the two VMs. In Figure 4.3, for each application, the left column shows the user perceived downtime when traditional approach is used, and the right column shows when Shadow Patching is used.

Second, we quantify the I/O activities caused by merging deltas and show the results in Table 4.4. The columns in the table are grouped based on the type of activities. Since rules 2 and 3 consist of regular copies, the results of these activities are combined as one column. Based on these results, we make the following observations:

- **Shadow Patching significantly shortens services downtime.** Because the tests of upgraded services are conducted on the cloned VM, the functional testing time, or *Shadow Patching testing*, does not impact on the patching time. As an example, a thorough test of a patched

Services	Rule 1	Rule 2 & 3	Rule 4
Bind9	2	48	0
SubVersion	4	29	2
NFS	2	35	3
OpenVPN	2	89	0
PostgreSQL	5	82	4
Samba	2	40	8
Squid	3	34	0
Apache2	5	552	6
VsFTPD	5	41	0

Table 4.4: Rule-based activities in merging deltas of application services: (1) deletes, (2 & 3) copies, and (4) hybrid copies.

NFS server using IOzone can take up to 40 minutes. Running this test on the cloned VM would take the same time and provide the same results but without being visible to users.

- **Shadow Patching lowers overhead incurred by the software component replacement.**

Comparing two versions of an application, if changes are minor, most files and directory structures will be similar, if not almost identical. If changes are more extensive, the similarities are insignificant. Traditionally, patching an application involves three steps: (1) removing current application's files and directories, (2) extracting the new version of the application into a temporary location, and (3) copying the extracted files and directories into the right place.

For a package whose changes are minor, this technique incurred unnecessary I/Os on files and

directories, which are identical between two versions. However, Shadow Patching avoids this redundancy by comparing inode information of files and directories between two versions before each delta merge, thus, resulting in fewer I/O operations. As an example, the results show that Shadow Patching helps Postgresql and Samba minimize their upgrading time.

- **Shadow Patching achieves less variations in services downtime than traditional approach across different software upgrades.** In traditional approach, we observe large time variations in services downtime for different applications. This is largely due to the testing needed for these applications can differ significantly.
- **Shadow Patching does not impact on the number of merging activities.** Application services include sets of files, which can be unchanged or significantly modified from their previous versions. Shadow Patching utilizes this observation to minimize the number of copies. In addition, although merging activities include deletes, copies, and hybrid copies, we can see that the majority of the activities are regular copies. The number of copies occurred on each package depends on differences between software versions, rather than its size or the number of files.

4.5.3 When Patch Fails

We further compare Shadow Patching with traditional approach when one or more patches fail. A service pack is a bundle of many patches to upgrade the current system version to the next stable version. Patches are applied in a certain order to satisfy software dependencies, and if any one fails, it is simply skipped (as well as any dependent ones). However, the failed patches will eventually

Ubuntu Versions	# Succeed Packages	Merging Activities			# Failed Packages	Fixing effort (hours)
		Rule 1	Rule 2 & 3	Rule 4		
10.04 → 10.10	339	1.460	43.079	943	8	20.0
10.10 → 11.04	346	1.597	47.406	1.148	5	12.5
11.04 → 11.10	432	1.474	62.351	1.237	0	0

Table 4.5: Upgrading different versions of Linux Ubuntu

need to be resolved within the change window.

A patch can fail for many reasons, such as insufficient hardware, driver problems, incompatible setup process, inconsistent system configuration, wrong architecture edition, data loss, permission/access problems, or software bugs. To resolve a failed upgrade, the following steps are usually taken: (1) reporting a problem, (2) looking for solutions from different databases, while waiting for the problem being solved, and (3) applying solutions to fix the failed upgrade. If a failure is caused by software bugs, bug-fixing is a non-trivial task. Generally, the time to fix a bug can be up to 200 days, although this number depends on the nature of the bug [59]. Recent studies of system configurations to upgrade software indicate that on average the time to fix one particular issue is no more than 5 hours [16, 86]. This average time is also known as a *fixing effort* to denote an effort in person-hours to resolve an issue. In general, fixing issues existed in separate software packages can be accomplished in parallel, but requires more labor. Otherwise, the issues must be fixed sequentially.

Table 4.5 shows the results of upgrading Linux Ubuntu systems, including successful and failed upgrades. To estimate the fixing efforts for failed upgrades, two scenarios are considered: (1) all

failed upgrades must be fixed in order, and (2) at least two failed upgrades can be resolved at one time. In the worst case scenario, resolving 8 failed upgrades takes $8(\text{failures}) \times 5(\text{hours}) = 40(\text{hours})$. However, $40(\text{hours})/2(\text{failures}) = 20(\text{hours})$ would be a fairly estimated fixing effort for the other scenarios. Note that for comparison purpose, Ubuntu systems are upgraded under both traditional and Shadow Patching methods. Shadow Patching does not modify the patching process, and thus producing similar results to the traditional method.

Conducting a thorough test on an upgraded system is a complex and time consuming task. This is because upgrading a Linux system requires multiple replacements of software components, including executable binaries, shared libraries, configuration settings, databases, etc. To verify the accuracy and stability of upgraded software, various regression tests should be conducted. However, it is non-trivial to fully understand and prepare thorough tests for all upgraded software, and it is also out of scope of this work. Due to this complexity, we focus on upgrading time, rather than testing time. Basically, the upgrading time includes the time to replace software components and the time to reboot the system. For Shadow Patching, the upgrading time consists of both the time to reboot the system and the time to merge deltas. Based on the comparison of results between traditional and Shadow Patching upgrades, which are shown in Figure 4.4, we make following observations:

- **Shadow Patching helps a system administrator avoid failed upgrades.** Since many reasons can cause failed upgrades, if the maintenance window is short (e.g., a few hours) such failed upgrades may not be resolved. Thus, the system will not be fully upgraded. The experimental results shown in Table 4.5 clearly illustrate this incident. Increasing the maintenance window gives more time to resolve the problem, however, it also greatly increases the services

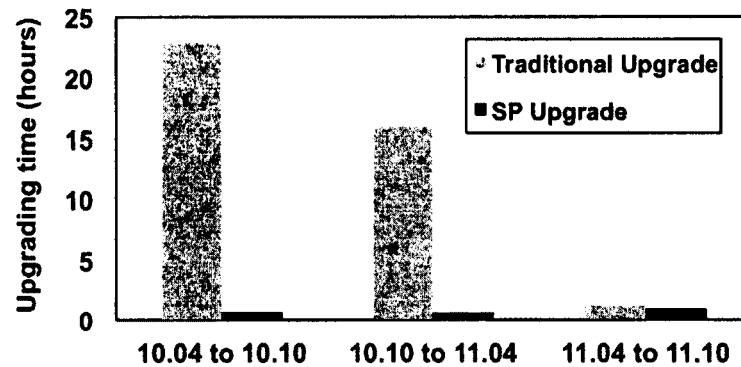


Figure 4.4: Upgrading different Linux Ubuntu server distributions (lower is better)

downtime. Shadow Patching is able to address this problem since resolving failed upgrades is performed in the cloned system.

- **Shadow Patching shows a low variation of upgrading time between different versions of a Linux system.** Traditional software upgrades do not guarantee that a Linux system can be successfully upgraded after a specific amount of time. This is because an actual time spent for an upgrade can be varied from one to several hours. More specifically, the upgrading time induced by Shadow Patching depends on not only system's configuration, but also the number and type of software packages. Our experimentation to upgrade different versions of a Linux system on the same test-bed shows a low variation of the upgrading time. This is due to the similarity between different versions of the service pack.

The choice of Linux distributions can be varied in enterprise environments. Systems can be built based on different Linux distributions, such as Ubuntu, openSUSE, or Fedora. To verify whether or not Shadow Patching can successfully upgrade various Linux distributions, we conduct a set of experiments to upgrade an openSUSE system, from version 11.3 to version 12.1. We observe that a newly upgraded openSUSE 12.1 is able to perform properly.

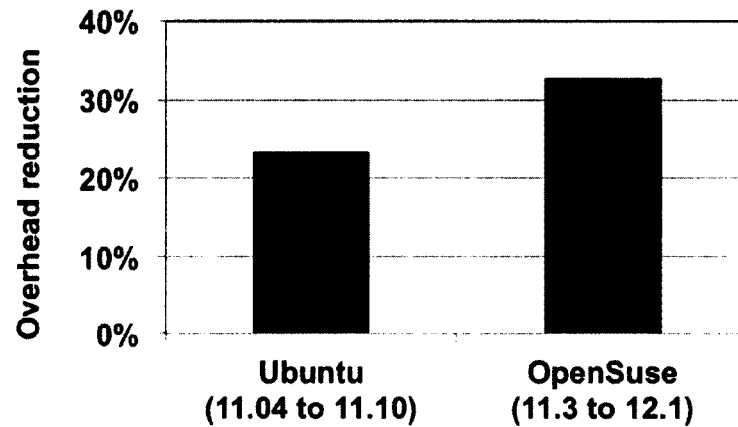


Figure 4.5: Overhead Reduction by Shadow Patching between different Linux distributions (higher is better)

Different Linux systems may benefit differently from Shadow Patching in reducing the service downtime. By default, the service downtime depends on the booting time of the system and the upgrading time of software patches. Here the upgrading time includes the time to remove old software, extract the new ones, and merge deltas. For an upgrade using service packs that contain hundreds of individual patches, the majority of the service downtime is attributed to the upgrading time. It is true that service packs are different across Linux distributions in terms of size and packed software patches. Therefore, the overhead induced by upgrading these service packs are also different between Linux distributions. Figure 4.5 shows how much overhead, which is induced by upgrading service packs, can be reduced when Shadow Patching is chosen over the traditional upgrades. As we can see, the upgrade of OpenSUSE from 11.3 to 12.1 benefits more by using Shadow Patching than the upgrade of Ubuntu from 11.04 to 11.10, in terms of the upgrading time reduction.

4.6 Discussion

In this section, we first discuss pros and cons of Shadow Patching, and then we present different approaches to further enhance the framework.

4.6.1 Pros and Cons

Shadow Patching framework does not require any changes to hypervisors, guest VMs, or software patches to perform a system upgrade. Typically, software packages and patches are complex to understand. Thus, to perform a system upgrade, traditional upgrades usually require either software engineering or system skills from administrators. Shadow Patching simplifies this requirement, so that administrators can perform any software upgrades without having a full understanding of either software packages or software patches. The software engineering skill is only required to resolve failed upgrades.

Most techniques employed in traditional software upgrades try to achieve a zero down time by inserting updated software components into running application services. However, such techniques can only successfully upgrade particular applications, thus, not becoming generic solutions. Although Shadow Patching focuses on software upgrades in virtualization environments, the framework is generic because it does not depend on one particular application or system. At the host level, Shadow Patching can leverage different hypervisors, such as KVM, Vmware, or Xen, to maintain a clone VM without modifying core features of hypervisors. At the guest level, Shadow Patching does not modify the kernel, but benefits from kernel subroutines to facilitate delta monitoring.

Shadow Patching also exposes some restrictions, such as disruption of running application ser-

vices and extra consumed system resources. The two major limitations of Shadow Patching are detailed as follows.

Merging deltas of Shadow Patching can only be performed when VMs are dormant. Thus, Shadow Patching cannot provide a live system upgrade. Specifically, a live system upgrade must achieve a zero downtime or only a brief delay of running services. However, Giuffrida *et. al* reveal that an ideal solution to achieve zero downtime in upgrading systems is not possible [52]. Although the recent work of Ksplice demonstrates a zero downtime in upgrading Linux kernel [35], this technique cannot be widely employed to live upgrade of any Linux systems or application software. However, a short planned downtime (from 5 to 20 minutes) caused by Shadow Patching for a system maintenance is fairly insignificant, compared to an actual maintenance window of enterprise systems (from 5 to 12 hours).

Shadow Patching consumes extra system resources to run a cloned VM. The cloned VM consumes CPU cycles, network traffic, and disk I/Os to fully obtain and apply software patches. Due to the consumption of extra system resources, using Shadow Patching to simultaneously upgrade multiple virtualized systems may impact on the hypervisor's I/O performance. System administrators should avoid this circumstance by scheduling a maintenance window of VMs in suitable times. In other words, since deltas are preserved within copy-on-write disk images, merging those deltas should be only performed when a maintenance time of the original system is appointed.

4.6.2 Shadow Patching Enhancements

There are alternative approaches to further enhance Shadow Patching, which are briefly described as follows.

4.6.2.1 Selective monitoring

Selective monitoring is a technique to minimize the overhead induced by monitoring deltas. In general, deltas are determined based on footprints of inotify's events, known as inotify traces. Since all I/O events are recorded, it is possible that the traces may include I/O events occurred on unmodified files and directories. Apparently, monitoring these I/O events is unnecessary. To effectively monitor deltas, the selective monitoring classifies the directories at the root level to avoid a recursive scan of the entire root directory. Basically, these directories are classified based on their purposes. For instance, `/bin`, `/boot`, and `/dev` directories include command binaries, boot loaders, and devices, respectively. Thus, it is possible to assume that once the system runs, those directories are not changed while the most modifications of the file systems are limited to other directories, such as `home`, `tmp`, `var`, or etc. Using inotify to focus on monitoring deltas of these modified directories does not impact on the results of deltas, but reduces the overhead of merging activities.

4.6.2.2 Flexible merging

A flexible merging tries to reduce the overhead of merging deltas by analyzing deltas and data files before each merge. Basically, to merge deltas, one particular file is copied between VMs regardless its size. However, if the size of the deltas is small (few Bytes or KBs) compared to that of the file (hundreds GBs), a regular copy induces much redundant I/Os. This is because the majority of I/Os are used to copy similar portions of the file from the cloned VM to the original VM. If the type of the file is determined, such as regular data, character/block device, domain socket, named pipe, or symbolic link, the flexible merging is able to avoid redundant I/Os by specifically copying dissimilar

portions, instead of the entire content of the file. To determine the dissimilar portions of the files, we need to thoroughly analyze deltas at two different levels: file system and disk block.

- **File systems:** The dissimilar portions are determined by thoroughly scan contents of two versions of the file. A merging process properly overwrites dissimilar portions of the file, from the cloned VM to its version on the original system.
- **Disk block:** The dissimilar portions of two files can also be directly obtained as data blocks from the layer of disk blocks, rather than indirectly through the layer of file systems. A merging process is performed based on a copy of such data blocks, which are associated with dissimilar portions of files.

4.6.2.3 Patching other systems

Shadow Patching's disk images must be copy-on-write and guest VM's file systems must be structured. These requirements enable us to perform an external snapshot of virtualized storage and to expose the file systems of the disk images. As we can see, the requirements are at the level of storage, rather than other particular system states, such as virtualized CPU, harddrive interfaces, or allocated memory pages. Thus, this allows Shadow Patching to be able to patch non-Linux systems without major changes of the framework. For example, administrators can use Shadow Patching to upgrade Microsoft Windows Server 2008 systems running on NTFS or DFS file systems. Since NTFS and DFS are structured file systems, they can be exposed under Shadow Patching for a side-by-side comparison of files and directories, thus, facilitating merging deltas between VMs.

Note that deltas are monitored differently under different file systems. Linux file systems use

inode, while Windows file systems (NTFS or DFS) rely on `fileID`. Because of the different data structure, a modification of `inotify` is required in order to properly monitor and merge deltas on the Windows systems.

4.7 Summary

In this chapter, we propose Shadow Patching framework to reduce the maintenance window associated with deploying software patches. Software patching, testing, and troubleshooting are all done in a cloned VM so that these tasks will have no impact on the original VM. File system changes in the cloned VM are recorded and are subsequently merged with the original VM. The only down time perceived by the original VM is when it is taken offline to perform this merge operation, which is much faster and reliable than what is done in the traditional method. By hiding post-patch regression test and troubleshooting steps, maintenance window can be significantly shortened.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

In this dissertation, we aim to tackle problems of understanding abstractions brought by virtualization. The focus of our work lies in different abstraction aspects, such as an intensive memory multiplexing, called Batmem, and a nesting of file systems. We later propose a framework, called Shadow Patching, to effectively minimize the maintenance window time for software upgrading. The summarized discussion of contributions of these work are as follows:

- **Batmem:** We believe that this work will help researchers to better understand the critical issues of memory sharing and VMBR in both high-end and low-end virtual support systems. We hope that our work will also motivate system designers to carefully evaluate security gaps at the real/virtual boundary in designing devices for virtual environments and to pay more attention to the threats posed by the adaptive behaviors of VMBR.
- **Nested File System:** The intricate interactions between host and guest file systems represent an exciting and challenging optimization space for improving I/O performance in virtualized

environments. Our preliminary investigation on nested file systems will help researchers to better understand critical performance issues in this area, and shed light on finding more efficient methods in utilizing virtual storage. We hope that our work will motivate system designers to more carefully analyze the performance gap at the real and virtual boundaries.

- **Shadow Patching:** Indeed, compared with an actual time of the system upgrade, a given maintenance window is usually much larger. This is attributed to unpredictable software bugs, which mainly cause upgrade failures, traditional software patching frameworks have no choice, but increase the maintenance windows. Through extensive experiments, we have demonstrated that SP is able to not only avoid failed upgrades, but also significantly minimize the maintenance windows. We believe that our framework will help system administrators in enterprise environments to optimize the software maintenance process. We also expect that our work will motivate software developers and system administrators to carefully monitor deltas at different levels, such as file systems and disk blocks, to shorten the software upgrading time.

5.2 Future Work

5.2.1 Memory Optimization in Virtualization

In high-end systems, Batmem is embedded into the hypervisor without violating the security design of the hypervisor. System administrators can protect Batmem from other malicious accesses inside VMs by placing Batmem as a read-only component within a protected memory area of the host.

Such a technique follows the similar approach of memory shadowing proposed by Riley *et al.* [74].

As a result, Batmem is protected in high-end systems.

For low-end systems, we discuss the challenges to protect a low-end system from an installation of VMBR, as well as to detect its presence, in the rest of this section.

5.2.1.1 Preventing VMBR Installation

To protect the host boot sequence from malicious modifications of VMBR, we can employ software or hardware solutions. Software solutions secure BIOS or boot processes by using encryption or out-of-the-box verification. Attackers need to retrieve the BIOS information to properly configure virtual devices when the host is started. Encryption methods prevent this retrieval by encrypting/decrypting the BIOS information upon its exchange among legitimate system components. Out-of-the-box methods use the checkpoint verification technique, which compares system snapshots between suspicious and legitimate boot sequences to discover the malicious modifications. In general, hardware solutions can be built on a tainting technique that monitors exchanged data among legitimate system components. Those suspicious uses of tainted data will be considered as illegitimate. However, for a low-end system, both software and hardware solutions are difficult to apply because they either need to reboot the system for the snapshot comparison or degrade the system performance by their aggressive verifications of primitive data.

5.2.1.2 Detecting Batmem

Since Batmem is operated as an embedded component within VMBR, detecting its presence is challenging. However, as we mentioned in Section 2.3, to easily locate the buffer of each MMIO

session, Batmem maps the offset of the dynamic circular buffer to the first page of the main memory. This design may motivate defenders to scan and compare the content of the device memory and the first page of main memory to determine grouped regions, and hence, detect the presence of Batmem. Unfortunately, aggressively checking memory partitions is very expensive, leading to significant performance degradation.

As an alternative, we can check the local time source to detect VMBR [50]. However, this method is not very robust since attackers can evade detection by using other similar approaches as Batmem to cloak their malicious activities on virtual components. In contrast, Garfinkel *et al.* [51] show a possibility of detecting VMBR without timing based techniques. Nevertheless, they target highly resource constrained VMBRs [60], and the flexible and small ones like Bluepill [77] are not considered.

We can also exploit a vulnerability of KVM by checking shutdown conditions of the VCPU triple faults at the user level [12]. The effectiveness of this technique highly depends on possibilities to conceal such shutdown conditions of attackers.

5.2.2 Virtualization Storage

Besides file systems, a nesting of other I/O interfaces or storage components in a virtualization environment may also impact on the I/O performance of running applications, especially those I/O components that direct I/Os among applications, back-end drivers, block devices, and shared storage systems. Therefore, it is important to fully understand their performance implications under different nesting scenarios, which are discussed as follows.

5.2.2.1 I/O Interfaces

Commodity systems attempt to guarantee not only a high I/O performance on state-of-the-art devices, but also a high compatibility between the systems and conventional devices. In a regular computer system, an I/O interface, which is represented by one particular device, handles I/Os passed through I/O buses. In a virtualized environment, due to the two-level of guest and host, the abstraction of I/O buses is created: the buses at the guest level are used to connect virtual block devices with back-end drivers, and the buses at the host level are used to connect the back-end drivers with mass storage devices. Guest/host I/Os can be exchanged under different interfaces, resulting in a variation of the system performance. Intuitively, an improper combination of the guest and host I/O interfaces can be disastrous to system performance. With an appropriate combination, the overhead induced can be negligible. Because of this, we consider a future study of nested I/O interfaces. More specifically, our focus will be on the performance implication of virtualized systems under different I/O interfaces employed at guest and host levels, such as IDE, SATA, PATA, USB, SCSI, SAS, FICON, Fibre Channel, InfiniBand, or Thunderbolt.

5.2.2.2 Storage Sharing

Commodity storage systems enable the virtualized enterprise features to scale out data stored on the disk. In general, such features create a dynamic pool of shared storage resource available to guest VMs. With the help from the dynamic pool, the performance and availability of I/Os across the physical storage are balanced and guaranteed. However, due to various I/O behaviors when different shared storages are employed, it is important to take advantage of the dynamic pool for one

particular system. Intuitively, the best performance of I/Os depends on a nesting of virtual storage. This nested virtual storage is specified by not only the types of workload and storage at the guest level, but also the physical storage at the host level. Therefore, our future work will focus on the issues of nesting storage, in terms of (1) types of storage (host-based or network-based), (2) methods to optimize a storage utilization (thin or fat provisioning), and (3) mechanisms to differentiate data and metadata.

Host-based Storage vs. Storage Area Network: Host-based storage for virtualization is implemented through a logical volume management. As an advantage, this management is able to minimize the complexity in controlling physical storage by providing a flexible logical view of storage systems. However, virtualized systems must be manually configured and separately managed.

A storage area network (SAN) provides block-level operations on interconnected storage devices, rather than stacking another file abstraction on the physical storage. As a significant advantage, a SAN based environment is completely transparent to virtualized systems. Thus, it is able to accommodate heterogeneous virtualized environments and minimize administrative tasks. However, to effectively leverage these block-level operations, another level of file system should be built on top of this SAN. In addition, a potential bottleneck induced by interconnected storage devices can impose a performance penalty on the entire virtualized environment.

In general, host-based or SAN-based large scale virtualized environments are created by a stacking of various levels of storage systems. This stacking includes multiple levels of physical or logical storage resources, whose I/O behaviors across the levels of storage are transparent. Thus, understanding I/O behaviors across those levels is necessary. As a future research direction, we plan to

conduct a thorough study on the multiple levels of storage systems to fully understand the performance implication of virtualized environments.

Storage Provisioning: A conventional storage provisioning technique, called fat (or thick) provisioning, enables an allocated storage space beyond current needs. This technique does not require to accurately predicate the actual capacity of the storage during I/Os, which may impact on I/O performance of storage systems.

Thin provisioning is a technique to optimize the available space that is utilized for storage systems, especially SANs. The major advantage of this technique is that it allows the system administrators to allocate storage space "just in time" and use only as little as they need. Thus, thin provisioning significantly saves available disk space.

To achieve the best performance of I/Os, it is important to choose a *right* provisioning technique for a virtualized system. For example, while the thin provisioning can save the disk space, it may also cause an unstable performance. For this reason, a deep study of storage provisioning techniques used for virtualized systems and their impact on the system's I/O performance are needed. As the problem will become even more challenging if the provisioning techniques could be deployed at different levels of storage, such as the physical level or the virtual level, we plan to explore this problem in the future.

Data and Metadata Differentiation: In a virtualized environment, it is difficult to differentiate guest VM's I/Os and determine if they are used for either data or metadata. Recently, few solutions have been considered to resolve this issue, such as (1) using device agents for VMs [29] to charac-

terize I/Os, (2) storing files as objects along with user-defined metadata [7], and (3) relying on an individual metadata manager as a service node for distributed file systems.

The proposed solutions may differentiate data from metadata. However, due to the changes made at both guest and hypervisor levels, those solutions may impact the scalability of a virtualized system, as well as the resiliency of data exchanged between two levels. Thus, they do not guarantee the best I/O performance if virtualized systems are hosted on different storage systems. By considering the effectiveness of different solutions in handling metadata for I/Os, we plan to further study the I/O performance of virtualized systems and provide a guideline for improving system I/O performance.

5.2.3 Shadow Patching Enhancements

5.2.3.1 Selective monitoring

A system downtime depends on a time to patch software. In fact, as software packages are different in terms of size and directory structure, the time to patch one particular software can be varied across different software packages, application versions, or upgrading procedures. As presented in experimental results, patching different application services results in different sizes of deltas to be merged. In general, deltas are determined based on footprints of inotify's events, known as inotify traces. Since all I/O events are recorded, it is possible that these traces may include I/O events occurred on unmodified files and directories. Apparently, monitoring these I/O events is unnecessary and may incur overhead.

To effectively monitor deltas, the selective monitoring classifies directories at root level to avoid

a recursive scan of the entire root directory. Basically, these directories are classified based on their purposes. For instance, `/bin`, `/boot`, and `/dev` directories include command binaries, boot loaders, and devices, respectively. Thus, it is possible to assume that once the system runs, these directories should not be changed while the majority of modifications is occurred on other directories, such as `home`, `tmp`, `var`, or etc. Using `inotify` to monitor deltas of these directories, this method does not impact on the results of deltas, but reduces overhead for merging activities.

5.2.3.2 Flexible merging

This method is based on an idea that an overhead induced by merging deltas is able to reduced if the file type and file contents can be determined in order to analyze. In general, based on deltas, merging activities will copy files from the cloned VM to to original system. However, if the size of delta is small (few Bytes or KBs) compared to the original file size (hundreds GBs), a regular copy significantly induces redundant I/Os on the storage and lowers the performance of merging activities. This is because the majority of these I/Os are used to copy a similar portion of the file from the cloned VM to another place. If a file type can be determined, such as regular data, character/block device, domain socket, named pipe, or symbolic link, a flexible merging is able to avoid these redundant I/Os by specifically copying portions instead of the entire file. To determine portions of the file to be copied, we need to thoroughly analyze deltas at two levels: file system or disk block.

- **File system:** Dissimilar portions are determined by thoroughly scan contents of two versions of the file. Merging activities properly overwrites dissimilar portions of the file, from the cloned VM to its version on the original system.

- **Disk block:** Dissimilar portions of two files can also be directly obtained as data block at the layer of disk block, rather than indirectly through the layer of file system. A merging process is performed based on a copy of disk blocks which are associated with dissimilar portions of files.

It is noted that for the best performance in merging deltas, a buffer used for copying dissimilar portions of a file should be considered. Also, a size and a structure of this buffer should be defined properly based on the data type to represent dissimilar portions of the file.

5.2.3.3 Patching other systems

Shadow Patching requires virtual disk images to be formatted as multiple-layer disks and file systems used at the guest level must be structured. These requirements provides an ability to perform an external snapshot and to mount disk images as individual storage at the file system level of the host. Since these requirements focus on the disk level, rather than other particular system settings, such as states of virtualized CPU, states of harddrive interface, or allocated memory, it allows to perform Shadow Patching to patch other systems than Linux without major changes required of the framework. As an example, Shadow Patching can be used to patch a VM that runs Microsoft Windows Server 2008 on NTFS or DFS file systems. Since NTFS and DFS are structured file systems, they can be exposed under Shadow Patching for a side-by-side comparison of files and directories between original and cloned VMs.

Note that deltas are handled differently under different file systems. Linux file systems use `inode`, while Windows file systems (NTFS or DFS) rely on `fileID`. Because of this different data

structure, a modification of `inotify` is required in order to properly monitor and merge deltas on such Windows systems.

Bibliography

- [1] Amazon Elastic Compute Cloud - EC2. [http://http://aws.amazon.com/ec2](http://aws.amazon.com/ec2).
- [2] Anubis: Analyzing Unknown Binaries. <http://anubis.iseclab.org> [Accessed: May 2012].
- [3] Autohotkey: Program with hotkeys and autotext. www.autohotkey.com.
- [4] Autokey: Text replacement tool for Linux. <http://autokey.sourceforge.net>, [December 2009].
- [5] blktrace - generate traces of the I/O traffic on block devices. [git://git.kernel.org/pub/scm/linux/kernel/git/axboe/blktrace.git](http://git.kernel.org/pub/scm/linux/kernel/git/axboe/blktrace.git) [Accessed: May 2012].
- [6] Bonnie++: File system benchmarks. www.coker.com.au/bonnie++.
- [7] DataDirect Networks - Web Object Scaler. http://www.ddn.com/pdfs/WOS_2_0_Whitepaper.pdf [Accessed: May 2012].
- [8] Filebench. www.solarisinternals.com/wiki/index.php/FileBench [Accessed: May 2012].
- [9] FIO - Flexible I/O Tester. <http://freshmeat.net/projects/fio> [Accessed: May 2012].
- [10] IBM Cloud Computing. <http://www.ibm.com/ibm/cloud/> [Accessed: May 2012].
- [11] Inotify - Monitoring File System Events. <http://www.kernel.org/doc/man-pages/online/pages/man7/inotify.7.html> [Accessed: May 2012].
- [12] Kernel TRAP - KVM: detect if VCPU triple faults. <http://kerneltrap.org/mailarchive/git-commits-head/2008/4/27/1622284>.
- [13] Lguest: The simple x86 hypervisor. <http://lguest.ozlabs.org>.
- [14] Linux kernel 2.6.22 modification. www.kernel.org/pub/linux/kernel/v2.6/ChangeLog-2.6.22.
- [15] Microsoft Virtual Machine Servicing Tool 3.0. <http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=23300> [Accessed: May 2012].

- [16] Microsoft Windows Server 2003 vs. Red Hat Enterprise Linux AS 3.0: IT Professionals Running a Production Environment. www.veritest.com [Accessed: May 2012].
- [17] Nested SVM virtualization for KVM. <http://avikivity.blogspot.com/2008/09/nested-svm-virtualization-for-kvm.html> [Accessed: May 2012].
- [18] Nlanr/dast : Iperf - the tcp/udp bandwidth measurement tool. <http://dast.nlanr.net/Projects/Iperf>.
- [19] Norman SandBox. http://www.norman.com/security_center/security_tools [Accessed: May 2012].
- [20] The QCOW2 Image Format. <http://people.gnome.org/~markmc/qcow-image-format.html> [Accessed: May 2012].
- [21] Set up and control loop devices. <http://linux.die.net/man/8/losetup> [Accessed: May 2012].
- [22] Sisoftware Sandra - Windows system analyser. www.sisoftware.co.uk/.
- [23] Superscape 3d vga benchmark. www.bookcase.com/library/software/msdos.util.screen.vga.html.
- [24] VirtualBox VDI. <http://forums.virtualbox.org/viewtopic.php?t=8046> [Accessed: May 2012].
- [25] Vlogger at the hacker's choice. www.thc.org.
- [26] Vmware diskmount utility. www.vmware.com/pdf/VMwareDiskMount.pdf [Accessed: May 2012].
- [27] VMware Tools for Linux Guests. http://www.vmware.com/support/ws5/doc/ws_newguest_tools_linux.html [Accessed: May 2012].
- [28] VMWare Virtual Disk Format 1.1. <http://www.vmware.com/technical-resources/interfaces/vmdk.html> [Accessed: May 2012].
- [29] VMware vSphere Update Manager. http://www.vmware.com/support/pubs/vum_pubs.html [Accessed: May 2012].
- [30] Window Azure - Microsoft's Cloud Services Platform. <http://www.microsoft.com/windowsazure/> [Accessed: May 2012].
- [31] Windows 2000 clustering: Performing a rolling upgrade. <http://technet.microsoft.com/en-us/library/bb742504.aspx> [Accessed: May 2012].
- [32] Xen Hypervisor Source. http://xen.org/products/xen_archives.html [Accessed: May 2012].
- [33] Xen Source -Progressive paravirtualization. http://xen.org/files/summit_3/xen-pv-drivers.pdf [Accessed: May 2012].

- [34] GAUTAM ALTEKAR, ILYA BAGRAK, PAUL BURSTEIN, AND ANDREW SCHULTZ. OPUS: online patches and updates for security. In *Proceedings of the 14th Conference on USENIX Security Symposium*, Berkeley, CA, USA, July 2005.
- [35] JEFF ARNOLD AND M. FRANS KAASHOEK. Ksplice: automatic rebootless kernel updates. In *Proceedings of the 4th ACM European Conference on Computer Systems*, Nuremberg, Germany, March 2009.
- [36] PAUL BARHAM, BORIS DRAGOVIC, KEIR FRASER, STEVEN HAND, TIM HARRIS, ALEX HO, ROLF NEUGEBAUER, IAN PRATT, AND ANDREW WARFIELD. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, New York, NY, USA, October 2003.
- [37] ANDREW BAUMANN, GERNOT HEISER, JONATHAN APPAVOO, DILMA DA SILVA, ORRAN KRIEGER, ROBERT W. WISNIEWSKI, AND JEREMY KERR. Providing dynamic update in an operating system. In *Proceedings of the USENIX Annual Technical Conference*, Anaheim, CA, USA, April 2005.
- [38] FABRICE BELLARD. QEMU, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference*, Anaheim, CA, USA, April 2005.
- [39] MULI BEN-YEHUDA, MICHAEL D. DAY, ZVI DUBITZKY, MICHAEL FACTOR, NADAV HAR'EL, ABEL GORDON, ANTHONY LIGUORI, ORIT WASSERMAN, AND BEN-AMI YASSOUR. The Turtles Project: Design and Implementation of Nested Virtualization. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, Vancouver, Canada, October 2010.
- [40] MEDHA BHADKAMKAR, JORGE GUERRA, LUIS USECHE, SAM BURNETT, JASON LIP-TAK, RAJU RANGASWAMI, AND VAGELIS HRISTIDIS. BORG: Block-reORGanization for Self-optimizing Storage Systems. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies*, San Francisco, CA, USA, February 2009.
- [41] DAVID BOUTCHER AND ABHISHEK CHANDRA. Does virtualization make disk scheduling passé? In *Proceedings of the Workshop on Hot Topics in Storage and File Systems*, Big Sky, MT, USA, October 2009.
- [42] HAIBO CHEN, RONG CHEN, FENGZHE ZHANG, BINYU ZANG, AND PEN-CHUNG YEW. Live updating operating systems using virtualization. In *Proceedings of the 2nd International Conference on Virtual Execution Environments*, Ottawa, Canada, June 2006.
- [43] LUDMILA CHERKASOVA, DIWAKER GUPTA, AND AMIN VAHDAT. When virtual is harder than real: Resource allocation challenges in virtual machine based IT environments. In *HP Laboratories - HPL-2007-25*, Palo Alto, CA, USA, February 2007.
- [44] CRISPIN COWAN, HEATHER HINTON, CALTON PU, AND JONATHAN WALPOLE. The Cracker Patch Choice: An Analysis of Post Hoc Security Techniques. In *Proceedings of the 23rd National Information Systems Security Conference*, Baltimore, MD, USA, October 2000.

- [45] FRANCIS M. DAVID, ELLICK M. CHAN, JEFFREY C. CARLYLE, AND ROY H. CAMPBELL. Cloaker: Hardware Supported Rootkit Concealment. In *Proceedings of the 29th IEEE Symposium on Security and Privacy*, Oakland, CA, USA, May 2008.
- [46] ARTEM DINABURG, PAUL ROYAL, MONIRUL SHARIF, AND WENKE LEE. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, Alexandria, Virginia, USA, 2008.
- [47] KENNETH J. DUDA AND DAVID R. CHERITON. Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In *Proceedings of the seventeenth ACM Symposium on Operating Systems Principles*, Charleston, SC, USA, October 1999.
- [48] TUDOR DUMITRAȘ AND PRIYA NARASIMHAN. Why do upgrades fail and what can we do about it?: toward dependable, online upgrades in enterprise system. In *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, Urbana, IL, USA, November 2009.
- [49] YASUHIRO ENDO, ZHENG WANG, J. BRADLEY CHEN, AND MARGO SELTZER. Using latency to evaluate interactive system performance. In *Proceedings of the 2nd Symposium on OS Design and Implementation*, Seattle, WA, USA, October 1996.
- [50] PETER FERRIE. Attacks on virtual machine emulators, December 2006. www.symantec.com/avcenter/reference/Virtual_Machine_Threats.pdf.
- [51] TAL GARFINKEL, KEITH ADAMS, ANDREW WARFIELD, AND JASON FRANKLIN. Compatibility is Not Transparency: VMM Detection Myths and Realities. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems*, San Diego, CA, USA, May 2007.
- [52] CRISTIANO GIUFFRIDA AND ANDREW S. TANENBAUM. A Taxonomy of Live Updates. In *Proceedings of the 16th Annual Conference of the Advanced School for Computing and Imaging*, Veldhoven, The Netherlands, November 2010.
- [53] GISLI HJALMTYSSON AND ROBERT GRAY. Dynamic C++ Classes: A lightweight mechanism to update code in a running program. In *Proceedings of the USENIX Annual Technical Conference*, New Orleans, LA, USA, June 1998.
- [54] TOM HO. Architecture for emulating an ethernet network interface card. United States Patent 7023878, April 2006.
- [55] HAI HUANG, WANDA HUNG, AND KANG G. SHIN. FS2: dynamic data replication in free disk space for improving disk performance and energy consumption. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, Brighton, United Kingdom, October 2005.
- [56] KHOA HUYNH AND STEFAN HAJNOCZI. KVM/QEMU Storage Stack Performance Discussion. In *Proposals of Linux Plumbers Conference*, Cambridge, MA, USA, November 2010.

- [57] VENKATESWARARAO JUJURI, ERIC VAN HENBERGEN, AND ANTHONY LIGUORI. VirtFS - A virtualization aware File System pass-through. In *Proceedings of the Ottawa Linux Symposium*, Ottawa, Canada, July 2010.
- [58] MUKIL KESAVAN, ADA GAVRILOVSKA, AND KARSTEN SCHWAN. On Disk I/O Scheduling in Virtual Machines. In *Proceedings of the 2nd USENIX Workshop on I/O Virtualization*, Pittsburgh, PA, USA, March 2010.
- [59] SUNGHUN KIM AND E. JAMES WHITEHEAD, JR. How long did it take to fix bugs? In *Proceedings of the 3rd ACM International Workshop on Mining Software Repositories*, Shanghai, China, May 2006.
- [60] SAMUEL T. KING, PETER M. CHEN, YI-MIN WANG, CHAD VERBOWSKI, HELEN J. WANG, AND JACOB R. LORCH. SubVirt: Implementing malware with virtual machines. In *Proceedings of the 27th IEEE Symposium on Security and Privacy*, Oakland, CA, USA, May 2006.
- [61] AVI KIVITY, YANIV KAMAY, DOR LAOR, URI LUBLIN, AND ANTHONY LIGUORI. kvm: the Linux Virtual Machine Monitor. In *Proceedings of the Linux Symposium*, Ottawa, Canada, July 2007.
- [62] OREN LAADAN AND JASON NIEH. Operating system virtualization: practice and experience. In *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*, Haifa, Israel, May 2010.
- [63] HORACIO ANDRÉS LAGAR-CAVILLA, JOSEPH ANDREW WHITNEY, ADIN MATTHEW SCANNELL, PHILIP PATCHIN, STEPHEN M. RUMBLE, EYAL DE LARA, MICHAEL BRUDNO, AND MAHADEV SATYANARAYANAN. SnowFlock: rapid virtual machine cloning for cloud computing. In *Proceedings of the 4th ACM European Conference on Computer Systems*, Nuremberg, Germany, March 2009.
- [64] DAVID E. LOWELL, YASUSHI SAITO, AND EILEEN J. SAMBERG. Devirtualizable virtual machines enabling general, single-node, online maintenance. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, USA, October 2004.
- [65] KRISTIS MAKRIS AND RIDA A. BAZZI. Immediate multi-threaded dynamic software updates using stack reconstruction. In *Proceedings of the USENIX Annual Technical Conference*, San Diego, CA, USA, June 2009.
- [66] KRISTIS MAKRIS AND KYUNG DONG RYU. Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels. In *Proceedings of the 2nd ACM SIGOPS/European Conference on Computer Systems*, Lisbon, Portugal, March 2007.
- [67] MICHAEL R. MARTY AND MARK D. HILL. Virtual hierarchies to support server consolidation. June 2007.
- [68] DIEGO ONGARO, ALAN L. COX, AND SCOTT RIXNER. Scheduling I/O in virtual machine monitors. In *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, Seattle, WA, USA, March 2008.

- [69] PRADEEP PADALA, XIAOYUN ZHU, ZHIKUI WANG, SHARAD SINGHAL, AND KANG G. SHIN. Performance Evaluation of Virtualization Technologies for Server Consolidation. Technical report, HP Labs Report HPL-2007-59, Palo Alto, CA, USA, April 2007.
- [70] JONATHAN W. PALMER. Web Site Usability, Design, and Performance Metrics. *Information Systems Research*, 13(2), June 2002.
- [71] SHAYA POTTER AND JASON NIEH. Reducing downtime due to system maintenance and upgrades. In *Proceedings of the 19th Conference on Large Installation System Administration Conference*, San Diego, CA, USA, December 2005.
- [72] DARRELL REIMER, ARUN THOMAS, GLENN AMMONS, TODD MUMMERT, BOWEN ALPERN, AND VASANTH BALA. Opening black boxes: using semantic information to combat virtual machine image sprawl. In *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, Seattle, WA, USA, March 2008.
- [73] ERIC RESCORLA. Security holes... who cares? In *Proceedings of the 12th Conference on USENIX Security Symposium*, Washington, DC, USA, August 2003.
- [74] RYAN RILEY, XUXIAN JIANG, AND DONGYAN XU. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection*, Cambridge, MA, USA, September 2008.
- [75] JOHN SCOTT ROBIN AND CYNTHIA E. IRVINE. Analysis of the Intel Pentium's ability to support a secure virtual machine monitor. In *Proceedings of the 9th Conference on USENIX Security Symposium*, Denver, CO, USA, August 2000.
- [76] RUSTY RUSSELL. virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems Review*, 42(5), July 2008.
- [77] JOANNA RUTKOWSKA. Introducing Blue Pill, June 2006. <http://theinvisiblethings.blogspot.com/2006/06/introducing-blue-pill.html>.
- [78] SEETHARAMI R. SEELAM AND PATRICIA J. TELLER. Virtual I/O scheduler: a scheduler of schedulers for performance virtualization. In *Proceedings of the 3rd International ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, San Diego, CA, USA, June 2007.
- [79] PRASHANT J. SHENOY AND HARRICK M. VIN. Cello: A Disk Scheduling Framework for Next Generation Operating Systems. In *Proceedings of ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Seattle, WA, USA, June 1997.
- [80] STELIOS SIDIROGLOU, SOTIRIS IOANNIDIS, AND ANGELOS D. KEROMYTIS. Band-aid patching. In *Proceedings of the 3rd USENIX Workshop on Hot Topics in System Dependability*, Edinburgh, UK, June 2007.

- [81] SANKARAN SIVATHANU, LING LIU, MEI YIDUO, AND XING PU. Storage Management in Virtualized Cloud Environment. In *Proceedings of the 3rd IEEE International Conference on Cloud Computing*, Miami, FL, USA, July 2010.
- [82] SURIYA SUBRAMANIAN, MICHAEL HICKS, AND KATHRYN S. MCKINLEY. Dynamic software updates: a VM-centric approach. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Dublin, Ireland, June 2009.
- [83] KUNIYASU SUZAKI, TOSHIKI YAGI, KENGO IJIMA, NGUYEN ANH QUYNH, AND YOSHITO WATANABE. Effect of readahead and file system block reallocation for lbcas. In *Proceedings of the Linux Symposium*, Ottawa, Canada, July 2009.
- [84] CHUNQIANG TANG. FVD: a high-performance virtual machine image format for cloud. In *Proceedings of the USENIX Conference on USENIX Annual Technical Conference*, Portland, OR, USA, June 2011.
- [85] CARL A. WALDSPURGER. Memory resource management in VMware ESX server. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*, Hollywood, CA, USA, October 2002.
- [86] CATHRIN WEISS, RAHUL PREMRAJ, THOMAS ZIMMERMANN, AND ANDREAS ZELLER. How long will it take to fix this bug? In *Proceedings of the 4th IEEE International Workshop on Mining Software Repositories*, Minneapolis, MN, USA, May 2007.
- [87] PAUL R. WILSON, SCOTT F. KAPLAN, AND YANNIS SMARAGDAKIS. The case for compressed caching in virtual memory systems. In *Proceedings of the USENIX Annual Technical Conference*, Monterey, CA, USA, June 1999.
- [88] BEN-AMI YASSOUR, MULI BEN-YEHUDA, AND ORIT WASSERMAN. On the DMA mapping problem in direct device assignment. In *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*, Haifa, Israel, May 2010.
- [89] HENG YIN, DAWN SONG, MANUEL EGELE, CHRISTOPHER KRUEGEL, AND ENGIN KIRDA. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, Alexandria, VA, USA, October 2007.
- [90] WU ZHOU, PENG NING, XIAOLAN ZHANG, GLENN AMMONS, RUOWEN WANG, AND VASANTH BALA. Always up-to-date: scalable offline patching of VM images in a compute cloud. In *Proceedings of the 26th Annual Computer Security Applications Conference*, Austin, TX, USA, December 2010.

VITA

Duy Le

Duy Le has been at The College of William & Mary, Williamsburg, VA since 2006. He got his M.S degree from The Francophone Institute of Computer Science and B.S degree of Computer Science from Hanoi University of Technology, Vietnam in 2003 and 2001, respectively.

His primary research interests focus on File Systems and Storages, Virtualization, and Cloud Computing. His research interests also include Energy Efficiency and System Security. In details, his research aims to tackle challenges of understanding the I/O behavior brought by virtualization and leveraging the virtualization technology in commodity computing systems. By focusing on both performance and security issues in a virtualization system, findings from his research will motivate system designers to carefully analyze the performance gap at the real and virtual boundaries.