Computer Systems Management with a Para Pass-through Virtual Machine Monitor

Computer Systems Management with a Para Pass-through Virtual Machine Monitor

Yushi OMOTE

(Doctoral Program in Computer Science)

Submitted in Partial Fulfillment of the Requirements

for Doctor of Philosophy in Engineering

at the

University of Tsukuba

March 2015

Abstract

System virtualization has brought many benefits to computer systems management and is widely used in various systems to improve manageability. On the other hand, because of virtualization overhead and limited physical device functionality, virtualization is prohibitive in many cases both in servers and desktops. However, some virtualization benefits, especially rich storage I/O controllability of virtualization is still desirable feature even in such cases. For example, storage I/O redirection by virtualization enables network-booting, caching or version management of guest OSs in the OS-independent manner and enables cost-efficient centralized management of OSs. Storage I/O encryption by virtualization enables end-point protection which significantly enhances security of the entire systems.

However, existing frameworks today have difficulty in simultaneous pursuit of such sufficient OS-transparent I/O controllability on storage and the merits of performance and device functionality. We have designed a new systems management framework to achieve the both by mediating storage I/Os in the para pass-through VMM. The framework runs the guest OS without virtualization, transparently intercepts storage I/Os of the guest OS, and provides storage management functions such as redirection, multiplexing and conversion of storage I/Os.

We have applied our framework to the two systems management scenarios to solve their problems. First, we allow rapid OS provisioning in bare-metal clouds to mitigate their limitations in agility and elasticity. Secondly, we perform background storage encryption to allow both easy deployment and higher security in desktops management. Finally, we evaluate our framework in both cases and demonstrate that they provide flexible storage management functions while preserving performance and physical device functionality.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

System virtualization has brought many benefits to computer systems management today. For example, the ability to execute multiple OSs on a single physical machine enables multi-tenancy and server consolidation for power saving. The state controllability on OSs enables live migration and checkpointing for fault tolerance or load balancing. The I/O controllability appends advanced I/O functions to OSs such as I/O redirection and encryption for flexible management or security enhancement. Because of these beneficial features, virtualization is commonly used in various areas of systems management (e.g. server management in data centers, desktop management in offices and even the management of mobile devices).

However, there are many cases where performance costs and limitations of virtualization are prohibitive. Virtualization adds overhead and jitters to the performance of guest OSs because of heavy device abstraction; Virtual Machine Monitors (VMMs) implement virtual devices that require costly interventions of VMM operations into the device operations of the guest OSs (e.g. CPU-mode transitions, nested memory management and I/O device emulation). Performance degradation and unpredictability caused by the interventions are regarded as prohibitive in many compute-intensive and I/O-intensive applications (e.g. high-

performance clusters, gaming servers, streaming servers and databases [22, 66, 47]). The virtualization overhead is also prohibitive on devices whose performance is limited such as mobile devices.

Moreover, virtualization prevents guest OSs from utilizing some important capabilities of physical devices by abstracting real devices (e.g. GPUs, wire-less network devices and power management capabilities [53]). These limitations in device functionality can be an obstacle to the management of desktops and mobile devices, where various kinds of device capabilities are expected by users. Even for server machines, device functions poorly-supported in virtualized environment are often required (e.g. GPUs [11] and RAID [8]). As a result of these limitations, many users and companies including Google and Facebook do not use virtualization to provide their core services [4, 5]. Providers of Infrastructure-as-a-Service (IaaS) including IBM recently start supplying physical machines instead of virtual machines [13, 6, 12].

Even in such use cases where virtualization is prohibitive and is not accepted, some of the features provided by virtualization are attractive. For most performance-intensive applications, the feature of multi-tenancy is less important because a single OS taking the full advantage of the physical machine is more suitable configuration for maximum performance. On the other hand, rich storage I/O controllability provided by virtualization is still desirable feature. For example, storage I/O redirection by virtualization enables network-booting, caching or version management of guest OSs in the OS-independent manner and enables centralized management of OS images, which greatly saves management cost of software resources. Storage I/O encryption by virtualization enables end-point protection regardless of types and versions of OSs, which significantly enhances security of the entire systems. However, existing frameworks today have difficulty in simultaneous pursuit of the sufficient OS-transparent I/O controllability on storage and the mitigation of the virtualization drawbacks. The goal of our research is to design and implement a new systems management framework that provides OS-transparent rich I/O controllability that

enables flexible storage management: storage I/O redirection, multiplexing, manipulation or transformation, while minimizing the penalty on performance and device functionality.

To support various storage management features, the design of our framework fulfills the following requirements. To allow network-booting, caching or version management of OSs, our framework has the mechanism that allows OS-transparent I/O redirection over network; disk access issued by OSs are transparently intercepted and redirected to the server storage over network. To allow caching or version management of OSs by use of local disks, our framework has the mechanism that allows OS-transparent I/O multiplexing; system administrators can share, modify or control local disks of each machine, while allowing OSs running on each machine to access to the disks. To allow encryption or protection, our framework has the mechanism that allows OS-transparent I/O transformation; content of I/Os is converted or modified by system administrators in the OS-transparent manner.

1.2 Existing frameworks for storage management

Existing frameworks have difficulty in achieving sufficient OS-transparent I/O controllability without the virtualization drawbacks. A major approach to achieve advanced storage I/O functions such as I/O redirection or encryption is to implement special device drivers, which inevitably depends on the types and versions of the guest OSs. This approach therefore compromises OS transparency, which requires OS-dependent setup and configuration for device drivers, raising management complexity.

On the other hand, there are many researches to mitigate the virtualization drawbacks of conventional VMMs. *Pass-through* technique [57] allows guest OSs direct control on physical devices, bypassing virtual devices. This technique improves both performance and device functionality without interventions and abstraction by VMMs. However, the design of conventional VMMs has difficulty in pass-through of some device operations because conventional VMMs are originally designed to abstract all devices. For example, operations for interrupt handling are difficult to pass-through, which still causes virtualization

overhead [42]. The virtualization overhead of memory management cannot be eliminated completely especially for memory-intensive operations [42]. Power management capabilities (e.g. ACPI [53]) are difficult to be available to guest OSs because of the design limitations of conventional VMMs. *Pass-through VMMs* [75] are newly-designed VMMs for security enhancement (e.g. integrity checking). By the fundamental design change from the conventional VMMs, pass-through VMMs completely abandon virtual devices, allowing guest OSs full control on physical devices. Thanks to this design, they can successfully avoid the virtualization drawbacks. However, pass-through VMMs no longer provide controllability because they do not have any mechanisms to mediate I/Os of guest OSs.

1.3 Storage I/O mediation in the Para Pass-through VMM

Our approach is to achieve the rich I/O controllability by enhancing the para pass-through VMM [77]. Using the para pass-through VMM is a promising approach because the para pass-through VMM is the framework that basically allows I/O pass-through without virtualization but has the ability to selectively intercept targeted I/Os. However, the original design of the para pass-through VMM focuses on storage data encryption and thus does not assume rich I/O controllability such as I/O redirection or multiplexing; it works passively in response to each I/O issued by guest OSs and converting the data being transferred.

Implementing virtual devices in the para pass-through VMM is a possible approach to rich I/O controllability. However, this approach again imposes virtualization overhead and limited device functionality. Instead of stepping back to I/O control mechanisms that rely on virtual devices, we have proposed a novel mechanism for storage I/O control, *storage I/O mediators*, that perform polling-based device-interface-level I/O mediation by carefully monitoring, intercepting, manipulating and inserting data I/O requests in a manner conforming to device specifications. To control storage I/Os, this method does not perform abstraction but interpretation for identifying and controlling only data I/Os that are requisite minimal to provide necessary I/O features such as I/O redirection or encryption.

With this mechanism, we demonstrate important management operations of OS images with the para pass-through VMMs. We first demonstrate network booting of OSs, and caching and installing of OS images with high performance and full device functionality of guest OSs. Then, we also demonstrate background encryption of OS images by combining device mediators and the existing encryption mechanism of the para pass-through VMM.

1.4 Application to OS provisioning

1.4.1 Network booting

Network booting is an effective technique for simplifying management of large-scale distributed systems, enabling system administrators to perform centralized management of OS images. For example, they only need to modify a single OS image on a central image server for updating, patching, configuring OSs and applications without modifying each OS image on distributed machines. Because of these benefits, network booting is widely used in many organizations for management of both desktops and servers.

However, existing network booting techniques require OS-specific drivers or kernel modules as well as configuration. This OS dependency raises the complexity of management especially when multiple types or versions of OSs are used. For example, while Linux uses kernel modules for NFS or iSCSI for network booting, Windows needs different special driver products. Mac OS relies on another vendor-specific mechanism for network booting. Moreover, using wrong versions of modules cause problems. System administrators therefore need to carefully setup each type and version of OS, which requires much time and various skills for management.

To avoid these OS-specific tasks, the usage of VMMs is becoming more common today. By redirecting storage access from guest OSs to image servers over network, VMMs easily allow network booting independently of OSs. However, VMMs causes virtualization overhead and limits of device functionality. Virtualization overhead causes insufficient user

experience (e.g. bad performance effects on graphics). Limited device functionality prevents users from utilizing various kinds of useful devices.

We avoid the limitations of performance and device functionality by incorporating the ability of network booting into a para pass-through VMMs. Our VMM basically allows guest OSs direct access to and full control on physical devices. Our VMM intercepts only disk access and redirects it to image servers over network.

The challenge here is that we cannot use virtual devices straightforwardly because they add virtualization overhead and show virtualized interface which can be different from physical one, which potentially limits device functionality of physical disks such as power management. To address this, we perform mediation of disk I/Os issued by the guest OSs to physical disks instead of emulating virtual disks. We monitor and interpret disk I/Os from the guest OS, and redirect them to image servers in a manner conforming to device specifications. We have implemented the mediation mechanism by extending the existing para pass-through VMM, BitVisor [77].

1.4.2 Network deployment

Because network booting heavily relies on the performance of image servers, network booting suffers the limitation of scalability, making servers I/O bottlenecks in very large-scale systems such as data centers for cloud computing. We have therefore improved device mediation so that the para pass-through VMM achieves further controllability on local disks to utilize them as caches or to deploy OS images on them completely. Caching and deploying OSs on local disks enables VMMs to reduce dependency on server performance and achieve greater scalability. This improvement has widened the range of applications of para pass-through VMMs to the infrastructure of cloud computing.

Bare-metal clouds are an emerging form of infrastructure-as-a-service (IaaS) that leases physical machines (*bare-metal instance*) rather than virtual machines. Bare-metal clouds therefore allow resource-intensive applications to have exclusive access to physical hardware,

which cannot be achieved in a traditional IaaS cloud with virtualization. Unfortunately, bare-metal instances require time-consuming or OS-specific tasks for the initial OS deployment. A straightforward approach to deploying an OS is to copy an entire OS image from a server to the local disk prior to starting-up the instances. Thus, the customer must wait for up to tens of minutes (depending on the network bandwidth and image size) for the copy procedure to finish. Rebooting the machine after the copy further increases the total wait time by several minutes. This long deployment time significantly impairs the beneficial features of a traditional IaaS cloud, such as agility and elasticity, and becomes an obstacle in, for instance, temporal testing, quick scale-up on demand, and hour-based pay-as-you-go services.

OS streaming deployment [37] is a promising approach to reduce wait time. This approach first performs a network boot and then copies the OS image to the local disk in the background. This enables quick startup of instances and eventual bare-metal performance after OS deployment is completed. Unfortunately, OS streaming deployment depends heavily on OS functions and configurations, thereby sacrificing another important feature of the IaaS cloud services, i.e., OS transparency. Abandoning OS transparency results in crucial limitations for customers and cloud providers because unskilled customers must test and verify the compatibility of special drivers with OS kernels whenever they update, patch, or customize their OSs.

Our goal here was to achieve streaming deployment in para pass-through VMMs. To this end, we have improved device mediators so that they allow VMMs share the physical disks with guest OSs. While redirecting I/Os to perform network booting, VMMs multiplex I/Os to the device to write the physical disks and install OS images. Compared to only performing network booting, the implementation needs deep interpretations of I/Os to physical devices.

Quick startup of instances improves agility and elasticity significantly, and OS transparency simplifies management tasks for cloud customers. Streaming deployment on phys-

ical hardware allows us to seamless elimination of the VMMs after entire OS images are copied onto the local disk. This allows *pure* bare-metal performance and completely identical to bare-metal execution without any intervening layers underneath OSs. We have further extended the para pass-through VMM, BitVisor [77]. Including redirection and multiplexing mechanism.

1.5 Application to background full-disk encryption

While allowing OSs access to physical disks, device mediators can modify OS images on the same disks. This feature is also applicable to the advanced storage encryption. To prevent data breaches, many companies deploy full disk encryption to their computers. Full disk encryption is a technique that encrypts entire contents of disks. Full disk encryption prevents attackers that physically steal disks from interpreting the contents. Widely-accepted implementation of full disk encryption is OS-based encryption because of its easiness of deployment into existing systems, supporting *background encryption*. Background encryption allows users to continue to use their PCs during encryption of pre-installed data and programs on the physical disks.

However, OS-based encryption needs OS-specific drivers, which prevents simplified management of OSs. On the other hand, VMM-based encryption offers significant advantages such as OS independence and providing more secure environments. However, the deployment cost of VMM-based encryption into existing systems is high because it does not support background encryption and requires physical-to-virtual (P2V) conversions in order to run OSs on virtualized environment, which are originally running on physical machines. It also suffers virtualization overhead and limited device functionality.

We present a VMM-based encryption scheme that allows instant deployment of full disk encryption into existing systems without disturbing user's activities. To avoid waiting for encryption to be completed, VMMs perform background encryption that does not incur significant performance penalty on guest OSs by carefully watching guest OS activities and

moderating the degree of encryption speed. Our scheme does not require conversion of disk images or modification of OS configurations to install VMMs. This is achieved by exposing physical disks to guest OSs and avoiding P2V conversions to run guest OSs on VMMs.

1.6 Contributions

This thesis makes the following contributions:

- The design and implementation of a new systems management framework that allows simultaneous pursuit of the OS-transparent rich I/O controllability for flexible storage management and the elimination of virtualization drawbacks.
- The design and implementation of a novel mechanism for storage I/O control, storage I/O mediators, that perform polling-based device-interface-level I/O mediation by carefully monitoring, intercepting, manipulating and inserting data I/O requests in a manner conforming to device specifications.
- The demonstration and performance evaluation of the framework applied to some important systems management operations for large-scale distributed systems; network booting of OSs, caching, installing of OS images with high performance and full device functionality of guest OSs.
- The demonstration and performance evaluation of the framework applied to an advanced storage encryption scheme, easily-deployable and highly-secure background encryption of storage devices.

1.7 Thesis organization

This thesis demonstrates I/O mediation in the para pass-through VMM enables OS-transparent flexible storage management which are originally provided by virtualization while preserving performance and device functionality of physical machines. Chapter 2 provides a brief

background on system virtualization and motivates our work, discussing the pros and cons of the design of conventional VMMs and storage management mechanism that relies on virtualization. Chapter 3 presents the design and implementation of our framework that performs storage I/O mediation with the para pass-through VMM. Chapter 4 demonstrates the application of our framework to OS provisioning for large-scale distributed systems. Chapter 5 demonstrates the application of our framework to advanced storage encryption scheme that allows easily-deployable and highly-secure background encryption of storage devices. Chapter 6 presents the discussion on the constraints of our framework and future works. Chapter 7 concludes and summarizes the key points of this thesis.

Chapter 2

Background and Motivation

2.1 Background: virtualization merits and drawbacks

System virtualization brings great impacts on computer systems management today. Traditionally, OSs are the most privileged software that provides execution environment for applications, controlling hardware. However, virtualization has changed this architecture; higher privileged software called *Virtual Machine Monitors* (VMMs) runs underneath OSs, and they control and manage OSs as unprivileged software (*guest OSs*). Providing virtual execution environment emulated by software (*virtual machine*), virtualization decouples *virtual devices*, which are used by guest OSs, and *physical devices*, which are actually implemented to the real machine (*physical machine*). Decoupling has brought greater flexibility in computer systems management, enabling us to establish more advanced computing infrastructures such as clouds. On the other hand, the VMM-based architecture brought by virtualization abandons some important abilities that are originally available in the traditional architecture where OSs runs on bare-metal. In this section, we discuss the pros and cons of VMM-based architecture, comparing it to the traditional architecture.

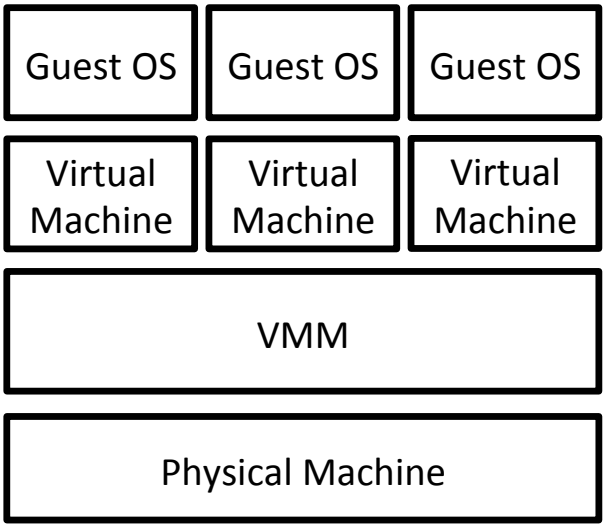


Figure 2.1: The VMM runs multiple virtual machines.

2.1.1 Virtualization merits

Virtualization has many beneficial features that significantly improve efficiency or flexibility of computer systems management. One of the most important advantages brought by virtualization is multi-tenancy. Decoupling of virtual and physical devices enables multiplexing of virtual devices on a single physical devices. Hence, VMMs successfully provide multiple virtual machines to run OSs concurrently on a single hardware (Figure 2.1). In addition, managing virtual machines in the highest privileged layer, VMMs can securely isolate each guest OS.

Thanks to multi-tenancy of VMMs, users who belong to different organizations or domains are now able to securely share physical machines by utilizing virtual machines isolated by VMMs. Because commonly-used VMMs today such as VMware, Xen and KVM hold OSs inside (*host OSs*) and rich functionality, having its own file systems, device drivers and network channels, system administrators can flexibly control virtual machines through the many convenient functions of VMMs even remotely. Therefore, relatively small number of system administrators can manage virtual machines for various users. This feature of VMMs help managing large-scale systems including hundreds or thousands of PCs in the

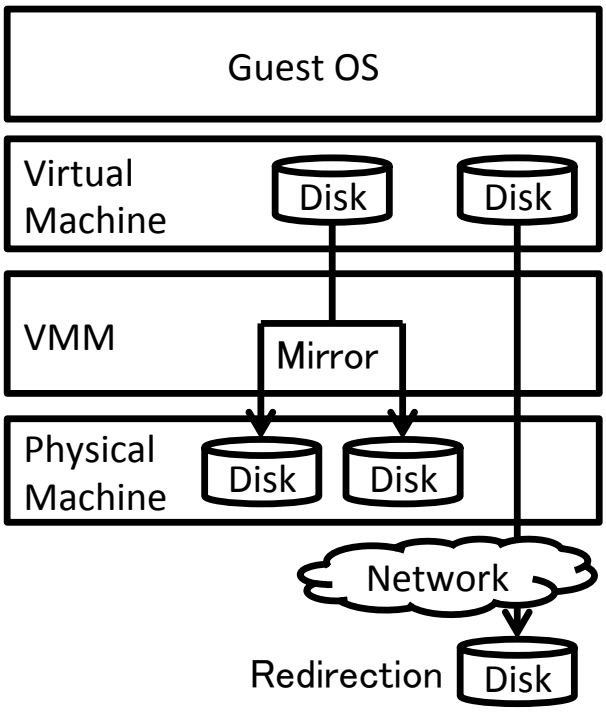


Figure 2.2: Storage virtualization improves manageability of disk images.

company or university. It also helps constructing large data centers for cloud computing that is popular platform today.

Muti-tenancy feature of VMMs also helps administrators improve the efficiency of hardware resource management. When each guest OS in a system does not require a lot of hardware resources (e.g. low I/O bandwidth or CPU utilization), administrators can aggregate multiple guest OSs in a single physical machine to efficiently utilize hardware resource, and they can power off the rest of physical machines to reduce energy consumption. Today, many powerful server hardware is shipped in the market, supporting many number of processors, large memory and I/O devices with broader bandwidth. Aggregation scenarios are therefore common today.

Decoupling also provides VMMs with the ability to control the state of virtual machines such as suspend and resume of virtual machines independently from the state of physical machines. Furthermore, VMMs achieve the ability to move a virtual machine between phys-

ical machines without stopping its operations (*live migration*). Live migration can further improve the efficiency of resource management by dynamically aggregate or distribute guest OSs depending on their loads. This feature also helps maintenance of physical devices that require reboot or shutdown by moving virtual machines in order to spare physical machines without disturbing the operations of guest OS users.

In addition to the capability of multi-tenancy, virtualization greatly improves system manageability, while preserving OS transparency; that is, virtualization enables us to easily implement management capabilities of OSs in the OS-independent manner without modifying OSs. This is very important feature because multiple types, versions and distributions of OSs can be uniformly managed with only a single implementation of the management features to VMMs. One typical example that virtualization improves manageability is *storage virtualization*. By intercepting I/Os to storage devices, VMMs can redirect disk reads and writes to local disks to other disks in the network, which achieves location transparency for OSs. Guest OSs for a certain virtual machine can be booted arbitrary place in the system (see "Redirection" in Figure 2.2). This allows centralized management of disk images of OSs, allowing system administrators single update of disk images, backup creation, and protection at the central server.

Another benefits of storage virtualization is that VMMs can transparently allow mirroring of storage by transforming a disk write from the guest OSs into replicated writes to multiple disks. In addition, VMMs can easily allow checkpointing of storage by intercepting and tracking the changes made to disks. They can rollback the state of the guest OS when some troubles or data corruption occurs with the current version of the disk images (see "Mirror" in Figure 2.2). Also, VMMs transparently trap storage or network access by the guest OS and perform operations for security enhancement of the system such as data encryption (e.g. full-disk encryption or VPN), file access control, firewalls, intrusion-detection or malware analysis. Since OS kernels today consist of large number of code and tend to have many bugs and vulnerability, implementing security functions in the OS-independent

components are regarded as more secure approach.

2.1.2 Virtualization drawbacks

While virtualization provides beneficial features to computer systems management such as multi-tenancy and manageability while preserving transparency, the VMM-based architecture with virtualization abandons some important benefits that can be achieved by running OSs on bare-metal. In this section, we mention two important problems on virtualization: performance problems and physical device functionality problems. We describe the current state-of-the-art technologies to mitigate the drawbacks.

Performance problems

One of the important drawbacks of virtualization is performance penalty. The source of virtualization overhead can be divided into three: CPU virtualization, memory virtualization and I/O device virtualization. The drawback caused by virtualization of each component is widely recognized and many researches and techniques to improve performance are introduced. However, completely eliminating the performance overhead (achievement of *bare-metal performance*) in virtual machines is still a open problem. We here describe efforts on performance improvement for virtualization of each component.

Traditionally, CPU virtualization is achieved by *CPU emulation* techniques, by which VMMs trap and emulate every CPU instructions in the programs of guest OSs. However, trap-and-emulate operations by software incur significant overhead and shows impractical performance in many applications. Hence, many software-based techniques (e.g. binary translation, better use of protection rings) are proposed to efficiently reduce the number of trap-and-emulate operations. However, software-based approaches have limitations in reducing trap-and-emulate operations. Hence, hardware-assisted virtualization (e.g. Intel VT-x, AMD-V) has been introduced. Hardware-assisted CPU virtualization enables VMMs to flexibly configure to trap only key instructions while allowing most instructions to be

directly executed by CPUs. Thanks to this hardware-based technology, CPU virtualization has succeeded in showing reasonable performance and even bare-metal performance in most applications today.

However, hardware-assisted CPU virtualization does not achieve pure bare-metal performance. Even with hardware-assisted CPU virtualization, VMMs still need to trap key instructions for virtualization (e.g. interrupt handling, memory management). Performance-intensive applications are affected by the cost of trapping even a single operations because the context switch from the context of guest OSs to that of VMMs pollutes CPU caches and potentially increases branch mis-predictions. Lock-holder preemption [84] issue can also cause overhead or jitter, which makes a thread in the guest OS delay releasing a shared resource due to the trap operations by the VMM occurred in the middle of its critical section.

Additionally, the cost of memory virtualization is another source of virtualization overhead. OSs today basically manage memory with the *paging* operation, with which OSs allocate memory to their applications in a unit of fixed-size regions called *page*. In the same way, the VMMs also perform paging to allocate memory to their guest OSs. Therefore, VMMs require implementation of the mechanism for nested paging operations. Traditionally, the nested paging is implemented by software (e.g. shadow paging). However, software-based nested paging requires very complicated operations with frequent trap-and-emulate operations of memory management instructions. Hence, as is the case with CPU virtualization, memory virtualization today is assisted by hardware (which is also integrated into Intel VT-x or AMD-V, being called EPT or NPT), in which most paging operations to control memory of guest OSs is directly performed in CPUs instead of software. This technology has greatly reduced the virtualization overhead of memory management. However, because the paging operations are costly even if they are implemented in the VMMs (requiring two-dimensional page walks) [], they fail to achieve bare-metal performance in memory-intensive workloads.

Finally, the performance improvement of I/O virtualization is very important and challenging issue today. The most basic implementation of I/O virtualization is *device emulation*. To perform device emulation, VMMs trap and emulate every I/O instruction to the target I/O device. VMMs also perform DMA operations and emulate generating interrupts. However, emulating every I/O instruction can cause many costly trap-and-emulate operations, and emulating DMA operations is often very costly increasing the number of times of memory copy. To mitigate the overhead, *para-virtualization* becomes a more popular solution today, which requires the guest OS to support special device drivers that is optimized for virtualization; that is, the driver is designed to reduce the number of I/O operations and interrupts, and to avoid redundant memory copy. However, para-virtualization is a compromised solution that requires the modification of guest OS and abandons the important virtualization merits we mentioned in the previous section, OS transparency. The limits the types or versions of available guest OSs in virtual machines. Furthermore, even the para-virtualization solution still causes overhead because it does not completely remove trap-and-emulation.

Moreover, in addition to the storage stack (file systems and device drivers) of guest OSs, the host OSs in VMMs also have its own storage stack. I/O requests issued by the applications on guest OSs are first served by the storage stack in the guest OSs. After that, the I/O requests in the device drivers in the guest OSs are intercepted by the host OSs and then handled by the software stack in the host OSs. Consequently, with either emulation or para-virtualization, a single I/O request needs to be handled by two nested software stacks in the VMM-based architecture. This causes significant overhead even though the number of trap-and-emulation operations is successfully reduced.

Physical device functionality problems

Virtualization prevents the guest OSs from directly controlling the physical hardware. While this feature provides the benefits such as high security by protecting system from malfunc-

tions in the guest OSs, the feature can be limitations to the guest OSs and their users who need to use full functionality of physical devices implemented in the machine. For example, advanced I/O devices such as GPUs or RAID controllers are difficult to be virtualized and thus utilizing full functions of GPUs from the guest OSs are known to be challenging today. In addition, WiFi devices are also known to be difficult to be virtualized. Furthermore, some CPU features are also hidden by the host OSs and difficult to make them identical to the physical ones on virtual machines.

2.2 Scenarios where virtualization is prohibitive

In spite of its great management benefits, virtualization is often avoided in computer systems management because their drawbacks are fatal and prohibitive to meet their requirements: bare-metal performance and flexible support of various hardware.

2.2.1 Scenarios in desktop management

System administrators in various organizations (e.g. companies, schools, hospitals, governments) need to manage hundreds or thousands of desktop machines properly. Virtualization is a good technology for desktop management today, greatly reducing management cost. However, performance penalty due to virtualization are often unacceptable.

A typical form of desktop management with virtualization is *local desktop virtualization* to place VMMs on each client machine and assign virtual machines to users. System administrators can simplify their steps to manage each virtual machines by use of VMMs, broadcasting management commands from the center in order to deploy, update and backup disk images for each virtual machine. Another common form is *remote desktop virtualization* or often referred as *thin client*, where virtual machines run on the central servers and their execution screens are redirected to the client machines. When users control these machines, their inputs via keyboards or mouses are redirected to corresponding virtual machines.

In both forms, virtualization overhead or jitter often becomes trouble on client machines.

While relatively light-weight applications (e.g. mailers, notes, spreadsheet applications, word processing applications) can accept the overhead, performance-intensive applications (e.g. video applications, design tools, computer-aided design (CAD) software, 3D games) often have trouble with performance impacts caused by virtualization. For example, video streaming relies on storage performance, each of which I/O requests are served in the deep storage stack that includes both virtual machines and host OSs. CAD, 3D games and even web browsers today (e.g. WebGL, Stage3D) require high-performance 3D rendering, which also follows complicated path; the VMMs first manage 3D renderings in the virtual machines and then proxy them to graphic libraries in the host OSs. Even with I/O pass-through, the overhead does not become zero due to IOMMU impacts. Especially in the case of thin client, redirecting execution screens and inputs adds further latency and does not suits performance-intensive workloads. As the result, design offices or desktops in art colleges where heavy 3D design tools is important often fail to accept virtualization. Internet cafes where 3D gaming is their sales point could also avoid virtualization.

Limited physical device functionality in virtual machines also becomes a critical issue, especially in the case that system administrators manage laptop or mobile devices by use of desktop virtualization methods. Most of laptop and table PCs use WiFi devices to connect to the Internet. However, virtualization of wireless network devices needs complicated operations in VMMs and is technically challenging [90]. Guest OSs also cannot use the handover technique that switches network connections among WiFi or 3G network depending on the state of wireless connection.

Battery performance is also limited by virtualization. Normally, when OSs are not at high utilization, they enable the lower-power states of CPUs and devices frequently to save the battery. However, since virtual machines cannot directly control power management capabilities, they unintentionally stay in the active states, wasting the battery.

Today, many companies start using the policy of bring your own device (BYOD), which organization members bring their own laptop or mobile devices into their companies. Hence,

physical devices are very various in this policy, and limited physical device functionality of virtualization becomes a critical problem.

2.2.2 Scenarios in server management

Virtualization is prevailing at the server-side systems management, diffusing cloud computing that supports today's computer systems. However, while most server applications accept virtualization, the recent trends in server applications often create conditions that virtualization is prohibitive for sufficient services.

Today, clouds are platforms for various applications. Not only traditional server-side applications such as web servers and mail servers but also client-side applications such as mailers, notes, spreadsheets and other productivity tools begin to run on servers. Even highly-interactive applications (e.g. games, graphic tools) are often implemented on servers, being accessed by many users via network. Hence, low latency or responsiveness is becoming a more important factor for servers today. Demands on throughput are also increasing because of the emergence of public clouds that handle requests from millions of users around the world. The recent trend of big data and high performance computing in clouds increase the level of server throughput and latency furthermore.

Virtualization therefore often fails to meet requirements for performance. For example, performance requirement for servers for world-wide social networking services, gaming services, and streaming services already reaches the one for high performance clusters. They therefore tend to suffer virtualization overhead [22, 66, 16, 69]. Consequently, many companies such as Google, Facebook and Microsoft does not use virtualization to provide their core applications.

Clusters for high performance computing today are constructed both on premise and clouds. In both forms, virtualization is often used for improving manageability. Virtualization overhead, however, becomes a problem. The performance scalability of HPC clusters is limited due to the network latency added by operations in VMMs, especially when a

number of virtual machines communicate each other [47].

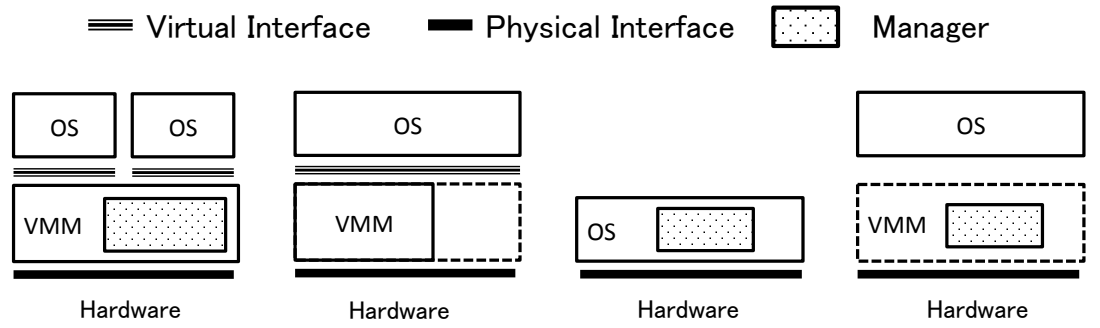
Not only performance but also limited device functionality also becomes problems in server machines. It is inconvenient for developers who test prototype software designed to exploit specific hardware features of CPUs, RAID controllers, and SSDs [11, 8, 15]. It is also inconvenient for computer scientists who evaluate the performance characteristics of their software because virtualization incurs indeterministic, complicated side effects on guest performance.

As the results, leading-edge cloud providers including IBM, have started to introduce bare-metal instances [12, 2, 6, 3, 13, 8]. start providing bare-metal clouds that supply physical machines instead of virtual machines to users in order to avoid the concerns caused by virtualization. Removing virtualization layers eliminates the performance overhead, allowing direct access to physical hardware eliminates the functionality limitations, and removing virtualization software eliminates the security concern.

2.3 Requirements and goals

Despite many efforts to mitigate virtualization drawbacks, there are many applications in the both client side and server side that cannot tolerate virtualization drawbacks imposed by conventional VMMs. However, some of systems management capabilities such as storage virtualization provided by the VMMs are still attractive for better management of OSs and applications.

To mitigate this dilemma, our research goal is to design and implement a new systems management framework that provides OS-transparent storage management features provided by conventional VMMs without performing virtualization. The design goal of our framework is to achieve the two conflicting merits: (1) the merit of storage virtualization of conventional VMMs and (2) the merit of high performance and full device functionality.



(a) Conventional VMMs (b) I/O pass-through (c) Bare-metal OS (d) Management with exposing physical interface

Figure 2.3: Design comparison of existing approaches

2.4 Existing approaches

There are many researches and techniques to avoid or mitigate limitations caused by virtual interface while enabling features of storage virtualization such as location transparency or storage protection. They are divided into three approaches: conventional VMM-based storage management, OS-based storage management and hardware-based storage management.

2.4.1 Conventional VMM-based approaches

As shown in Figure 2.3a, conventional VMMs expose virtual interface to OSs, which causes limitations on performance and device functionality. To mitigate the limitations, recently the VMMs optimize its design so that they expose physical interface to OSs instead of virtual interface (see Figure 2.3b).

For performance improvement of the conventional VMMs, there are number of researches to eliminate the storage stack in VMMs such nested file systems [38] and nested I/O schedulers [67]. Especially, the one that largely improves I/O performance in virtual machines is *I/O pass-through* [57]. This technique allows guest OSs direct control on physical I/O devices, completely eliminating software stack of host OSs from the I/O path. Since I/O pass-through minimizes interposition of VMMs, overhead is greatly reduced. However, even

I/O pass-through is not complete solution for pure bare-metal performance because host OSs cannot allow pass-through of all device operations such as interrupts [42], memory management [26] and I/O protection mechanism such as IOMMU [25], and these factors still cause overhead. I/O pass-through is basically applicable to only some of PCI devices, and pass-through of some I/O devices (especially, legacy I/O devices) are difficult to be supported in today's VMM architecture.

I/O pass-through also can help utilizing physical device functionality. However, I/O pass-through is not applicable all devices today because the basic design of conventional VMMs today is that host OSs basically controls the physical machines. Therefore, in order for the entire system to work properly, the host OS cannot allow direct control of the guest OSs on some fundamental and critical devices. For example, interrupt controllers cannot be directly controlled by the guest OSs in order to properly execute host OS operations (e.g. scheduling based on timers). Power management mechanism such as ACPI cannot be directly exposed to the guest OS because the host OS heavily relies on ACPI for the basic operations [44].

2.4.2 OS-based storage management

Since the conventional VMMs cannot completely eliminate the virtualization overhead and device functionality limitations, OS-based storage management is often used 2.3c. In OS-based storage management, storage management functions (e.g. redirecting or protecting storage access) are implemented in the layer of file systems or device drivers. For example, storage access from applications are intercepted and redirected to another storage or encrypted/decrypted by special device drivers or file systems implemented in the OSs. However, this design of OS-based approach abandons OS transparency; that is, it depends on the types or versions of OSs, which raises the complexity of implementation and management. For example, OS-based storage management forces systems administrators to setup and configure different drivers for different OSs, while users today often request various OSs

for their platforms. In addition, if the versions of OSs change, system administrators need carefully update or re-configure drivers.

2.5 VMM-based systems management exposing physical interface

Our approach is one of the VMM-based approaches. However, we re-design the architecture of the VMM, abandoning virtual interface and exposing physical interface directly to OSs. Hence, the OSs can utilize the full performance and device functionality of the physical devices. At the same time, the VMM properly arbitrates the I/Os from the OS to the physical interface and provides storage manageability. To achieve such architecture, we leverage the para pass-through VMM [77], which is the enhanced version of pass-through VMMs.

Recent years, the new type of VMMs, *pass-through VMM*, are proposed [75] which does not perform virtualization of devices, allowing guest OSs direct control on physical devices. They therefore no longer impose virtualization overhead, unlike conventional VMMs that rely on virtualization. The pass-through VMM becomes a higher-privileged layer that has controllability on the guest OS but without virtualization overhead. However, they only have mechanisms that control over CPUs and memory (for the purpose of ensuring code integrity) but do not have any mechanisms that manipulate I/Os of the guest OSs. Hence, their design does not allow an ability to manage storage access of the guest OSs. While exposing physical interface, the para pass-through VMM provides I/O controllability by intercepting I/Os.

However, the original design of the para pass-through VMM does not provide rich I/O functionality that is provided by the conventional VMMs today. The para pass-through VMM aims at only storage encryption and only converts storage I/Os to encrypt disk access of guest OSs. We extend the design of the para pass-through VMM and achieve the

rich I/O functionality of conventional VMMs. By use the functionality, we demonstrate advanced storage encryption applications and OS provisioning applications.

Chapter 3

Storage I/O Mediation in the Para Pass-through VMM

3.1 Para Pass-through VMM

Para pass-through VMM [77] is another new VMM recently proposed for security enforcement. Like the pass-through VMM, the para pass-through VMM also allows a single guest OS direct access to physical devices without performing any virtualization and indirection. On the other hand, the para pass-through VMM is design to be a framework that selectively intercepts I/Os of the guest OSs to perform security enforcement. For example, the para pass-through VMM intercepts only disk I/Os to encrypt or decrypt the data, while the VMM allows I/O pass-through for the other devices (e.g. keyboards, displays and network devices) that do not have to protect. In addition, the para pass-through VMM can decide the rule about whether to allow pass-through or not in the granularity of I/O instructions instead of devices; that is, the VMM also intercepts only disk reads and writes while allowing pass-through of other device operations such as device initialization and power management operations for disks.

3.2 Design extensions to the para pass-through VMM

Our goal is to provide storage management features of conventional VMMs that transparently redirects or manipulates disk access from the guest OSs while avoiding virtualization drawbacks such as performance degradation and limited device functionality. To achieve this goal, inheriting the design of para pass-through VMM in order to design our framework is a promising approach because the para pass-through VMM itself does not perform virtualization but provides I/O controllability, with which we can manipulate storage access of the guest OS.

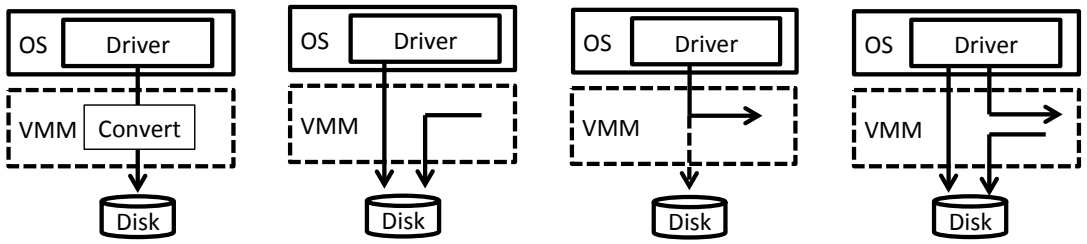
However, the original design of the para pass-through VMM focuses on storage data encryption and thus does not assume rich I/O controllability such as I/O redirection or multiplexing. Firstly, it works passively in response to each I/O issued by guest OSs and converts the data being transferred. However, we have to perform I/O redirection that needs to perform I/O operations actively in the VMM in order to send data via network devices. In the same way, I/O multiplexing needs the VMM to be able to actively read and write the local disks. Secondly, since the original design of the para pass-through VMM does not consider maximized performance, some of the design principles add the interposition overhead of the VMM. To achieve our goal, the original design of the para pass-through VMM needs to be extended.

In this work, we extend the para pass-through VMM so that the VMM can provide rich storage I/O controllability that conventional VMMs today provide. Our design extension follows the three steps shown in Figure 3.1. The original para pass-through VMM only supports I/O conversion that converts the contents of data I/Os 3.1a. However, the functionality is not enough for rich storage I/O management of conventional VMMs today. Therefore, in the first step, we extended the design of the para pass-through VMM so that it can perform I/O multiplexing 3.1b. I/O multiplexing allows the sharing of physical disks between the OS and the VMM; that is, while allowing the OS to use the local disk of the machine, the VMM also reads and writes the local disk in the transparent manner. By

using this function, we demonstrate background encryption, which is an advanced encryption features provided in conventional VMMs today for more efficient encrypted storage management.

Then, we extended the para pass-through VMM so that it can perform I/O redirection 3.1c. I/O redirection allows the VMM to change the direction of I/Os that are issued by the OS. The VMM intercepts the I/Os and directs them to the different disks that are different from the original targets. By I/O redirection, the VMM redirects local disk access of the guest OS to the network disks and performs network booting of OSs, another advanced management feature of the conventional VMM.

Then, we allow the VMM to simultaneously provide I/O redirection and I/O multiplexing as well as the existing operations of I/O conversion and pass-through. In this step, the VMM can perform rich storage management functions provided by conventional VMMs today, allowing location transparency of storage. As the demonstration, we applied this function to OS provisioning in bare-metal clouds.



(a) I/O conversion only in the original para pass-through VMM

(b) Implementing I/O multiplexing

(c) Implementing I/O redirection

(d) Combining I/O redirection and multiplexing

Figure 3.1: Extending para pass-through VMM.

3.2.1 Storage I/O mediators

The device mediators are the key components required to achieve device sharing with the guest OS while exposing the physical hardware interface to the guest OS directly. Device mediators perform I/O mediation; they mediate access to devices from the guest OS and

VMM. Storage I/O mediation involves three tasks: *I/O interpretation*, *I/O redirection*, and *I/O multiplexing*. I/O interpretation involves monitoring I/O sequences from the guest OS and determining their context. I/O redirection involves intercepting I/O requests from the guest OS and redirecting them to the server, and I/O multiplexing involves inserting I/O requests from the VMM to devices. Note that I/O interpretation is the basis of I/O redirection and I/O multiplexing. I/O redirection is used in the copy-on-read, and I/O multiplexing is used in the background copy.

I/O interpretation

I/O interpretation determines the context of I/O sequences to properly control I/O access from the guest OS. For example, in disk controllers, device mediators interpret three types of contextual information: *command*, *status*, and *data*. The command information contains the operation type (read or write), logical block address (LBA), and sector count. The status information determines whether the device is idle or busy, which is required to determine when operations terminate. The data information is about the actual data transfer, such as the DMA buffer address of the guest OS.

Device mediators can easily capture this information by monitoring programmable I/Os (PIOs) and memory-mapped I/Os (MMIOs) based on device specifications. Although actual I/O sequences are device-specific, the basic concept can be applied to many different types of disk controllers. In addition, device mediators only need to determine these basic I/O sequences and can ignore other irrelevant sequences, such as device initialization and vendor-specific configurations. Therefore, device mediators are simpler and smaller than conventional full-spec device drivers.

I/O redirection

I/O redirection is used in the deployment phase to redirect read access of empty blocks to the storage server. Device mediators must (1) capture the block information, (2) retrieve

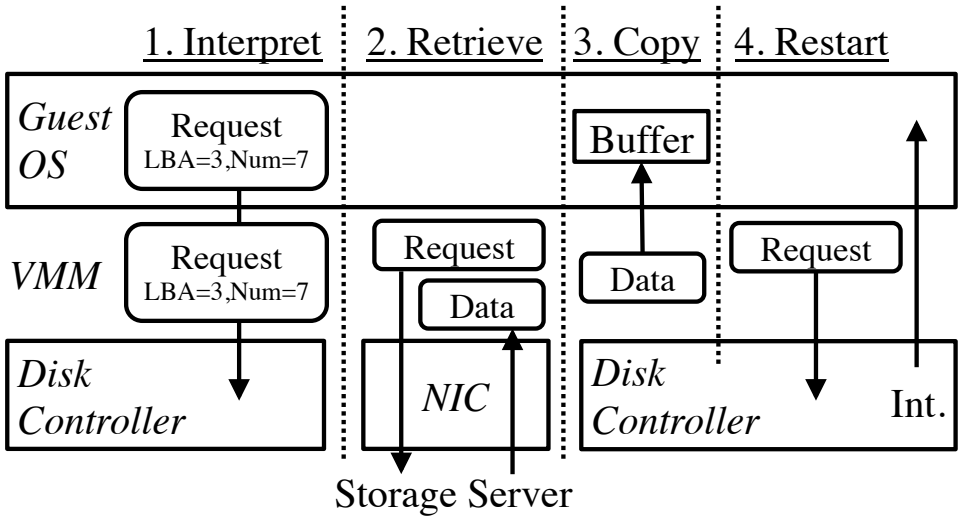


Figure 3.2: I/O redirection

the corresponding data from the server, and then (3) pass the data to the guest OS.

Figure 3.2 illustrates the basic operations of I/O redirection. Device mediators use I/O interpretation to capture the command information ("1. Interpret" in Figure 3.2). By interpreting I/O requests, device mediators can determine whether an access is read or not, and if read, its location (LBA and sector count). Based on this information, the VMM determines whether the blocks are empty or filled.

When blocks are empty, device mediators must retrieve the corresponding data from the server ("2. Retrieve" in Figure 3.2). First, device mediators temporarily block I/O access to the device so that the device does not start the data transfer. Device mediators then send the command information over the network and retrieve the data from the server. While retrieving data, device mediators emulate the status information so that the guest OS can determine that the device is busy. After retrieving the data, device mediators pass the data to the guest OS by functioning as a virtual DMA controller; i.e., copying the data to the guest DMA buffer ("3. Copy" in Figure 3.2). Device mediators obtain the address of the guest DMA buffer using I/O interpretation.

At this point, device mediators must generate an interrupt to indicate that the opera-

tion has been completed. The problem here is how to generate this interrupt. One possible approach is to virtualize interrupt controllers; however this complicates de-virtualization. Another possible approach is to share interrupt controllers with the guest OS transparently using techniques that are similar to I/O multiplexing, which is described below. However, sharing interrupt controllers is very complicated, especially when an advanced programmable interrupt controller is used. Keeping track of the interrupt numbers assigned by the guest OS to the devices is also difficult because it depends on the platform hardware such as low pin counter controllers and PCI bridges in the chipset. Therefore, this approach decreases portability drastically.

Rather than sending virtual interrupts, device mediators simply restart the blocked I/O access to the device so that the device itself generates an interrupt ("4. Restart" in Figure 3.2). To prevent the device from overwriting the guest DMA buffers, device mediators configure the device to transfer the data to dummy buffers. Device mediators also manipulate the command information (LBA and sector count) so that the device reads a single dummy sector that hits the disk cache. This technique allows easy and transparent generation of interrupts.

I/O multiplexing

I/O multiplexing is used for background copy. I/O multiplexing allows the VMM to issue its own I/O requests to devices while the guest OS controls those devices. Device mediators must (1) find proper timings to insert I/O requests, (2) emulate device status and handle the requests, and (3) manage request queues.

Figure 3.3 illustrates the basic I/O multiplexing operations. First, device mediators find proper timing to safely share a device in a time-sharing manner ("1. Find" in Figure 3.3). If the device is processing an I/O request from the guest OS, the device mediators wait for completion of the request. Device mediators detect this using I/O interpretation.

When the request is completed, the device mediators send the I/O request from the

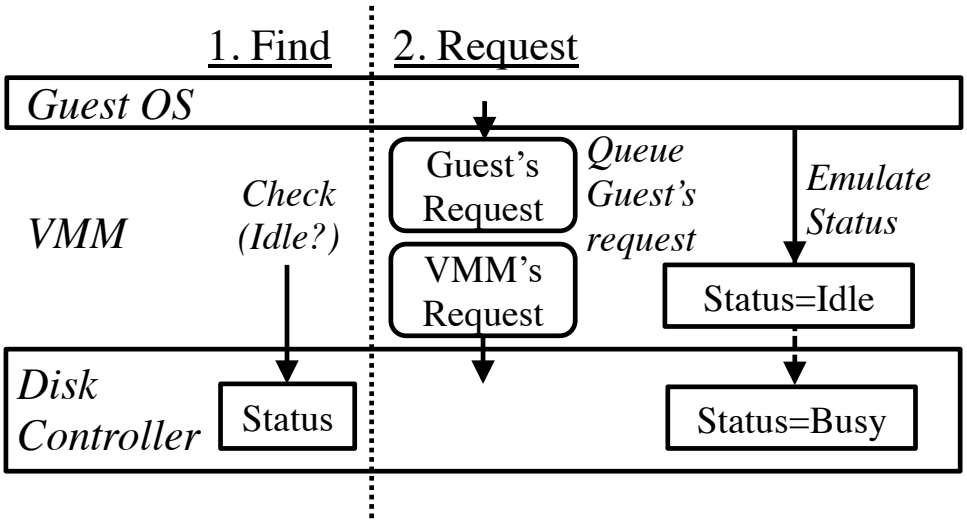


Figure 3.3: I/O multiplexing

VMM to the device ("2. Request" in Figure 3.3). To preserve consistency, device mediators emulate the status of the device as if the device is not busy even though the device is actually processing the request from the VMM. Therefore, at this time, the guest OS may attempt to send its own I/O request. To prevent conflicts with the guest OS, the device mediators intercept the request and keep it in a queue. After the request from the VMM is completed, the device mediators stop emulating device status and send queued requests to the device.

Sharing devices causes another problem with interrupts. Keeping track of the interrupt number assigned by the guest OS is difficult. As a result, the device mediators cannot detect the exact interrupts generated for the requests from the VMM. In addition, the interrupts for the VMM are also delivered to the guest OS because interrupt controllers are not virtualized. Although device drivers may safely ignore unknown interrupts to support interrupt sharing, extra interrupt delivery should be avoided to preserve portability. Therefore, device mediators temporary disable interrupts and detect completion of requests by polling threads with fine-grained scheduling. The details of this process are described in Section 4.4.1.

3.2.2 Thread scheduling

The para pass-through VMM is designed to be completely passive; that is, the VMM takes the control on physical machines only when the VMM traps I/O instructions issued by the guest OS. In other words, when the guest OS does not perform any I/O operations, the VMM is no longer invoked and scheduled. This design is reasonable for relatively simple I/O transformation; e.g. storage encryption that requires encryption or decryption operations only at the timing when the guest OS reads and writes the disks.

However, our goal is to provide more complicated I/O operations including I/O redirection and multiplexing. To perform I/O redirection, the VMM needs to run its own threads to actively send and receive packets to transfer disk access trapped by the VMM. To perform I/O multiplexing, the VMM needs to poll the completion of disk access so that the VMM actively reads and writes disks. Hence, the scheduling limitation of the para pass-through VMM that cannot execute its own thread without I/Os from the guest OS cannot actively schedule threads for I/O redirection or multiplexing. Therefore, we need to design and implement the appropriate scheduling mechanism for the para pass-through VMM.

We first need to implement scheduling mechanisms for the para pass-through VMMs in order to perform advanced I/O mediation. Regardless of I/O operations of the guest OSs, the VMM needs to determine the timeslice for a thread and precisely schedule the thread. Such thread scheduling is easily achievable in the conventional VMMs because they are based on the host OSs that originally contain full implementation of process and thread schedulers. However, importing the full implementation of schedulers of conventional VMMs conflict our goal of performance. For example, the host OSs of conventional VMMs rely on hardware interrupts to enable thread scheduling; the VMM configures interrupt controllers and timer devices properly so that a current thread is properly preempted by a hardware interrupt and switched to the next thread at the proper timing. The conventional VMMs therefore needs to have control over physical interrupt controllers, which requires virtualization of interrupt controllers in order to allow sharing of the interrupt controllers

among the VMM and the guest OS. In order to virtualize interrupt controllers, all hardware interrupts are forced to be trapped in the VMM layer for consistent operations and thus causes significant overhead as we mentioned in Chapter 2, even though most of interrupts are not important just for the implementation of storage I/O mediation.

Hence, the challenge here is to find alternative designs to perform thread scheduling that does not require virtualization of interrupt controllers. To this end, we leverage another hardware mechanisms that allow configurable preemption for threads. We can use of the these two mechanisms: *VMX preemption timer* feature or the *performance counter* feature. The former feature is the Intel-specific feature but allows the VMM to cause unconditional preemption of the operations in the guest OS and to pass the control on CPUs to the VMM. The interval between each preemption event is configurable in the granularity of CPU cycles. With this feature, the VMM can configure to cause preemption and scheduling of threads at the proper timing.

On the other hand, the other preemption mechanism, a performance counter, is available in both Intel and AMD CPUs. Performance counters are hardware counters implemented inside CPUs that are designed to measure some hardware events such as branch mis-predictions and TLB misses, and the OS can force hardware interrupts when the number of targeted events reaches the specified thresholds. CPUs today have multiple performance counters, and hence the VMM can occupy and conceal one of the performance counter to use to cause interrupts for preemption of threads.

However, in older CPUs, the VMM needs to intercept MMIOs to a few device register in interrupt controllers in order to occupy a performance counter. Since MMIOs force the interception in the granularity of pages, the interception of a few device register unfortunately requires trap for other device registers, which causes overhead since some of the device registers are accessed by the guest OS very frequently (e.g. the register for end-of-interrupt (EOI)). Recent CPUs have improved the unfavorable conditions by allowing per-register interception of interrupt controllers by mapping some device registers to

model specific registers (MSRs), and the VMM can avoid unnecessary trap to occupy a performance counter.

3.2.3 Driver implementation

The para pass-through VMM needs to control network devices in order to send and receive packets to network for I/O redirection. However, the para pass-through VMM does not have its own device drivers. Therefore, we also need the implementation of device drivers for the pass-through VMM in the appropriate way so as not to impose the virtualization drawbacks.

In order to allow redirection of I/O operations, the VMM implements its own device drivers to control network devices by itself. To control network devices, using hardware interrupts are usually used. However, the use of interrupts needs the VMM to use physical interrupt controllers while allowing the guest OS to use it. This requirement again forces virtualization of interrupt controllers and impose virtualization overhead. We therefore adopt polling-based device handling rather than relying on interrupts.

When the latency of requests are very short or predictable, device handling based on polling can often achieve better performance and CPU utilization than interrupts. In the case of I/O redirection, polling is not always necessary because the requests are only expected just after redirection. In addition, the response time from the server is often predictable by sampling the recent response time of requests. Based on these characteristics, we can reduce the frequency of polling operations and optimized performance.

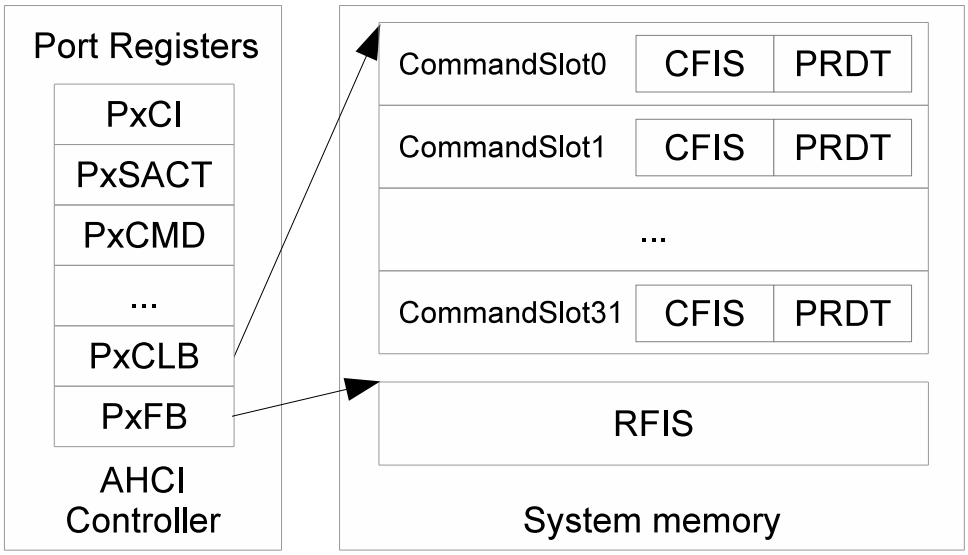


Figure 3.4: A simplified version of AHCI controller and its system memory structure.

3.3 Implementation example: I/O mediation for AHCI controller

I/O redirection

Figure 3.4 shows a simplified version of AHCI controller and system memory structure. An AHCI controller has multiple ports and each port is connected to a single ATA device. A port has a set of port registers and, among others, a PxCLB register points to a list of memory structures called CommandSlot (see the left side of Figure 3.4). A CommandSlot has information of a single read or write transaction, that is, *ATA command* to the device. An ATA command contains parameters to read or write an ATA device such as a block address on a disk and sector counts to be transferred. These parameters are embedded in a memory structure called Command FIS (CFIS) of a CommandSlot (see the right side of Figure 3.4), and passed to the target ATA device by setting a bit of PxCI register and PxSACT register of an AHCI host controller.

When the VMM detects that these bits are set, the VMM scans CFIS and interprets the content of the ATA command and extracts the block address and sector counts. If the block

to be accessed is empty, the VMM sends the ATA command to the storage server. While the ATA command is handled in the server, the VMM intercepts access to PxCI, PxSACT, and PxCMD registers in the AHCI controller, and the return the register values to the guest OS after setting some bits to indicate as if DMA is still running. When the ATA command has been completed and the VMM has received the response from the server, it stores the status value to Received FIS region pointed by PxFB register (see the bottom half of Figure 3.4). The data read from or written to an ATA device is stored in a memory region specified by Physical Region Descriptor Table (PRDT) of a CommandSlot. The VMM copies the data transferred from the server to the memory region.

The completion of an ATA command is ordinary notified with an interrupt from a host controller. A straightforward method is to assert a virtual interrupt. However, the VMM avoids virtualization or interception of interrupt controllers such as APIC and PCI bridges and does not manage interrupt information. Therefore, the VMM does not know to which interrupt vector it should assert a virtual interrupt. Hence, when the redirection completes, the VMM sends a command to the host controller to generate a actual interrupt. Here the VMM issues a small read command (read only a single sector) to the ATA device to avoid emulation of the register state on read completion.

To redirect ATA commands, the VMM uses a dedicated network card and hides it from the guest OS. Since we do not virtualize interrupt routing controllers, we cannot use interrupts to receive packets. Instead, we use the preemption timer capability to cause periodic VM exit and poll the device status. The VMM calculates the recent average RTTs to the server and estimates the appropriate timeout value. The VMM sets this timer only while waiting for response from the server to reduce unnecessary VM exits.

For the network card, we currently support Intel PRO/1000 and RTL8169x network card. To allow the disk I/O redirection before the guest OS boots, the VMM initializes the network card at its initialization phase. To hide the network card, the VMM intercepts I/O accesses to the PCI configuration registers of the network card and ignores write operations

and returns zero for read operations. After the de-virtualization of the VMM, the guest can recognize the network card and can optionally use it. However, the VMM does not notify if of the guest OS and a normal OS will not notice that. Therefore, it is still transparent from the guest OS.

To convert ATA commands to a form transmittable over network with minimal steps, we exploit *AoE protocol*. AoE protocol is designed for sending ATA commands over Ethernet. This protocol has a header which exactly contains the set of values of an ATA command and the VMM can easily convert from the ATA register set to an AoE header. Data to be read is transferred as a payload of a packet with the header. If the transferring data is too long for a single Ethernet packet, the VMM split the data into multiple AoE fragments. In this case, the VMM set the tag field in an AoE header to know the offset of received fragment. AoE protocol is originally designed for loss-less cables. Therefore, we implement a re-transmission capability in case of packet loss.

I/O multiplexing

To store the OS image to the local disk in the background, the VMM needs a mechanism to read and write the ATA device in parallel with the guest OS.

AHCI controller has multiple command slots (up to 32), and each slot can issue a single ATA command at a time. Whether each command slot is in use or not is shown in PxCI and PxSACT register. To safely share the command slots with the guest OS, the VMM must schedule the use of the slots. If a command slot is used by the guest OS, the VMM can avoid to use the slot. On the other hand, even if a command slot is used by the VMM, the guest OS may try to use the slot because the guest OS is not aware of the VMM using the slot. Therefore, the VMM keeps the command from the guest Os in a queue and virtualizes the status registers to indicate that the command is being processed at the port. After the command of the VMM has completed, the VMM gets a command from the queue, issues the command to the slot, and stop virtualizing the status registers.

If interrupts from the AHCI controller were enabled, when a command issued by the VMM has completed, an interrupt would be asserted from the controller and passed to the guest OS. However, the guest OS should not be aware of the command. Therefore, when the VMM starts a command, it temporary disables interrupts for the port being used, and enables it again after the command finishes. The VMM knows the completion of its own command by polling. The VMM again uses the preemption timer and estimates the timeout value from the recent average response time.

To fill the entire image on the local disk, the VMM actively transfers the image in the background. The VMM uses a bit-map to manage the empty and filled blocks.

Since the download rate and disk-write rate can be different, the VMM uses a pair of threads, the *downloader thread* and *writer thread*, which are mutually connected with a single FIFO queue. The downloader thread downloads data for an empty block from the server and pushed it to the queue. Then the writer thread pops the block data and writes them to the local disk. To minimize the disk seek latency, both threads try to access blocks adjacent to the last-accessed block. In case of shutdown or reboot, the VMM saves the bit-map to the local disk. The VMM detects such events by intercepting INIT signal, and blocks the termination process to ensure that the bit-map is saved. Currently the VMM uses unused region on the local disk (such as a chink between two partitions) to save the bit-map. To prevent overwrite of the bit-map by the guest OS, the VMM hides and protects them.

To avoid performance interference, the speed of background access should be moderated. To do so, the VMM monitors frequency of disk I/Os of the guest OS. If the rate is higher than a certain threshold, then the VMM waits until it becomes lower than the threshold. In our current implementation, we heuristically determined the threshold based on our experiments. The VMM checks the number of commands from the guest OS within 200msec interval. If the number is higher than 20, it temporary suspends background copy for 500msec.

Chapter 4

Application: OS Provisioning

In this chapter, we apply storage I/O mediation to OS provisioning of bare-metal clouds, a emerging Infrastructure-as-a-service (IaaS) clouds where physical machines instead of virtual machines are supplied, targeting customers who cannot accept virtualization overhead and limited device functionality of traditional virtual IaaS clouds. The problem of bare-metal clouds are slow OS deployment that compromises the original benefits of clouds: agility and elasticity. With our framework that provides flexible I/O controllability and low overhead, we improve the rapidity of OS deployment process.

This chapter is organized as follows. We briefly explain the background of bare-metal clouds and then motivate our work in Section 4.1. Section 4.3 presents our approach and Section 4.4 describes the implementation of the prototype VMM based on BitVisor. Section 4.5 shows the result of performance evaluation. Section 4.2 discusses related works and Section 5.6 concludes the paper.

4.1 Background and motivation

Bare-metal clouds allow the customers to directly run their OSs on the physical hardware without virtualization overhead and functionality limitations. Bare-metal instances are an attractive platform for customers who run gaming servers, media servers, database servers

and HPC applications where CPU & I/O overhead and jitters caused by virtualization software significantly affect the performance. Bare-metal cloud also has greater flexibility in hardware configurations, such as using dedicated GPU accelerators [11], RAID devices with specific configurations [8], and specific SSD products [15]. In addition, it can avoid security vulnerabilities posed by virtualization software, which is increasingly becoming a crucial problem [81, 91, 34]. Recently, several leading-edge providers are offering bare-metal instances [12, 2, 6, 3, 13, 8], and they are becoming widely available.

However, bare-metal instances take long time for initial deployment of OS images. A straightforward way to deploy OS images to bare-metal instances is *image copy*, copying the entire OS image from the storage server to the local disk of the bare-metal instances over network before starting-up the OS. Hence, the customer must wait for up to tens or hundreds of minutes, depending on the network bandwidth and image size, until the image copy has completed. The total waiting time could become even longer to decide server locations and reboot the instance. This long OS deployment time impairs beneficial features of traditional IaaS clouds such as agility and elasticity; it becomes obstacle for customers to temporarily use instances for testing, quickly scale up the number of instances on demand, or pay as you go on hourly basis.

OS streaming deployment [37] is one of the approaches to this problem; it first boots a target OS over network, then copies the entire OS image to the local disk in the background. This approach allows fast startup of instances and eventually achieves bare-metal performance after the OS deployment has completed. Unfortunately, it requires customized device drivers and depends on the OS types, versions and configurations of both the client and server, sacrificing another important feature of IaaS clouds, namely, OS transparency. It is crucial for customers who want to choose arbitrary OS images and customize them freely. Using conventional virtual machine monitors (VMMs) such as Xen [21] and KVM [51] achieves OS transparency and can provide various useful features such as live storage migration [61, 29, 63] and VM streaming [14]. However, conventional VMMs

incur continuous virtualization overhead and have difficulty in providing pure bare-metal performance [42, 53, 76, 76, 86, 79].

Our goal is to achieve agility and elasticity in bare-metal cloud by providing fast startup of instances and eventual bare-metal performance without sacrificing OS transparency. To this end, we extend the design of the para pass-through VMM so that the VMM first performs the streaming deployment of OSes, making the instances to be quickly ready to use. The OS deployment is performed in a OS transparent manner by utilizing virtualization technologies, while incurring as little overhead as possible. After the completion of OS deployment, the VMM turns off the virtualization and completely disappears to eliminate the virtualization overhead.

The key challenge of this approach is how to achieve seamless and transparent de-virtualization. Before de-virtualization, VMMs normally need virtual devices and device drivers to handle complex management of device access. It also needs virtual interrupt controllers to handle interrupts from devices. However, exposing virtual devices to the guest OS makes de-virtualization complicated because the device interfaces visible to the guest OS become different before and after de-virtualization. Even if the guest OS supports plug-and-play, the device changes may require temporal suspension of OS functionaries [53], or require OS reboots. Moreover, interrupt controllers are usually configurable only at boot time, and synchronizing the internal device states of virtual and physical devices is difficult especially when the device specifications of the devices are different.

To address this challenge, we designed the VMM to directly expose almost all the physical devices, including storage devices and interrupt controllers, to the guest OS. This design makes the device interfaces visible to the guest OS identical before and after de-virtualization and contributes to reduce the virtualization overhead. To manage the device access and track the internal physical hardware states, the VMM performs polling-based device-interface-level I/O mediation that carefully intercepts, monitors, manipulates and inserts I/O requests to the devices in a manner conforming to the device specifications. This

technique allows device sharing with the guest OS while achieves seamless and transparent de-virtualization. We also designed the VMM to use a network storage protocol that can convert storage device access to network packets with minimal efforts to reduce the overhead of storage virtualization. Note that we specialized the design of the VMM for fast instance startup in bare-metal cloud and abandoned the support of general VMM functionaries such as running multiple OSes.

We have implemented a prototype VMM based on BitVisor [77]. We changed the core of BitVisor 1.4 by only 3,576 LOC. The prototype supports x86 environments with IDE and AHCI disk controllers, and Intel PRO/1000, x540, Realtek RTL816x, and Broadcom NetXtreme network interface controllers (NIC). We also designed and implemented a network storage protocol that extends the ATA-over-Ethernet (AoE) protocol [73] to improve network storage performance. Although sharing an NIC with the guest OS is technically possible, our current implementation uses a dedicated NIC for streaming OS deployment to avoid performance interference. We believe using a dedicated NIC for management is a reasonable configuration because modern server machines typically have multiple NICs.

The experimental results confirmed that our VMM can deploy Windows (Vista, 7, 8.1, Server 2008) and Linux (Ubuntu 10.04 and later, and CentOS 6.3 and later) without any modifications. The VMM started up a bare-metal instance 8.6 times faster than image copying. The average database throughput on our VMM was comparable to that on a state-of-the-art VMM, i.e., kernel-based virtual machine (KVM) with exit-less interrupts (ELI) [42, 41] even though our VMM was performing streaming OS deployment while KVM with ELI did not. After de-virtualization, our VMM incurred zero overhead.

4.2 Related work

We first present OS deployment approaches and then discuss VMM overhead from an OS deployment perspective.

OS deployment: Image copying, such as in OpenStack Nova [9], is an OS-transparent but time-intensive approach to OS deployment. For example, copying a 30-GB image (default Windows Server 2008 on Amazon EC2) at 130 MB/sec (e.g., transfer from a 1.5 K rpm SAS disk over 10-Gb Ethernet) takes approximately 5 minutes. Using SSDs may reduce the copy time; however, the server or network may saturate when multiple instances are deployed simultaneously. Rebooting machines after the copy further add several minutes due to long firmware initialization time. Network installation, such as Kickstart in Linux, is a similar approach that copies and installs system files from the server to the local disk over the network. However, it is OS-specific and takes tens of minutes to complete the copy operation.

Network booting boots up an OS quickly; however, it does not deploy the OS image to a local disk, which causes continuous overhead as a result of redirecting every disk I/O over the network. Caching data on the local disk [32] reduces I/O overhead but must check cache expiration for every disk access, which increases disk access latency. OS streaming deployment [37] is a hybrid approach to network booting and image copying that allows fast OS startup and deployment to the local disk. However, it compromises OS transparency.

Our approach enhances OS streaming deployment by exploiting a de-virtualizable VMM to provide OS transparency while retaining low overhead during deployment and eventual bare-metal performance.

VMM overhead: Leveraging conventional VMMs, such as Xen [21] and KVM [51], is an easy approach to start up an OS quickly while preserving OS transparency. However, despite efforts to reduce virtualization overhead, such as para-virtualization [21] and I/O pass-through [70, 88], such VMMs do not achieve bare-metal performance in certain compute-intensive and I/O-intensive workloads [22, 66, 47] due to the lock-holder preemption problem [84], cache pollution, nested paging, and interrupt handling overhead [26, 42].

Another possible approach is to uninstall VMMs after OS deployment. Several recent

VMMs support raw disks and virtual-to-physical conversion. Therefore, using VMMs during steaming OS deployment and then converting virtual instances to physical instances is possible. However, uninstalling VMMs requires OS reboots, which incurs downtime. A recent study has shown that a more seamless conversion is possible by exploiting OS hibernation [53]. Unfortunately, in addition to slight modifications to the OS, this requires 90 seconds for physical-to-virtual conversion.

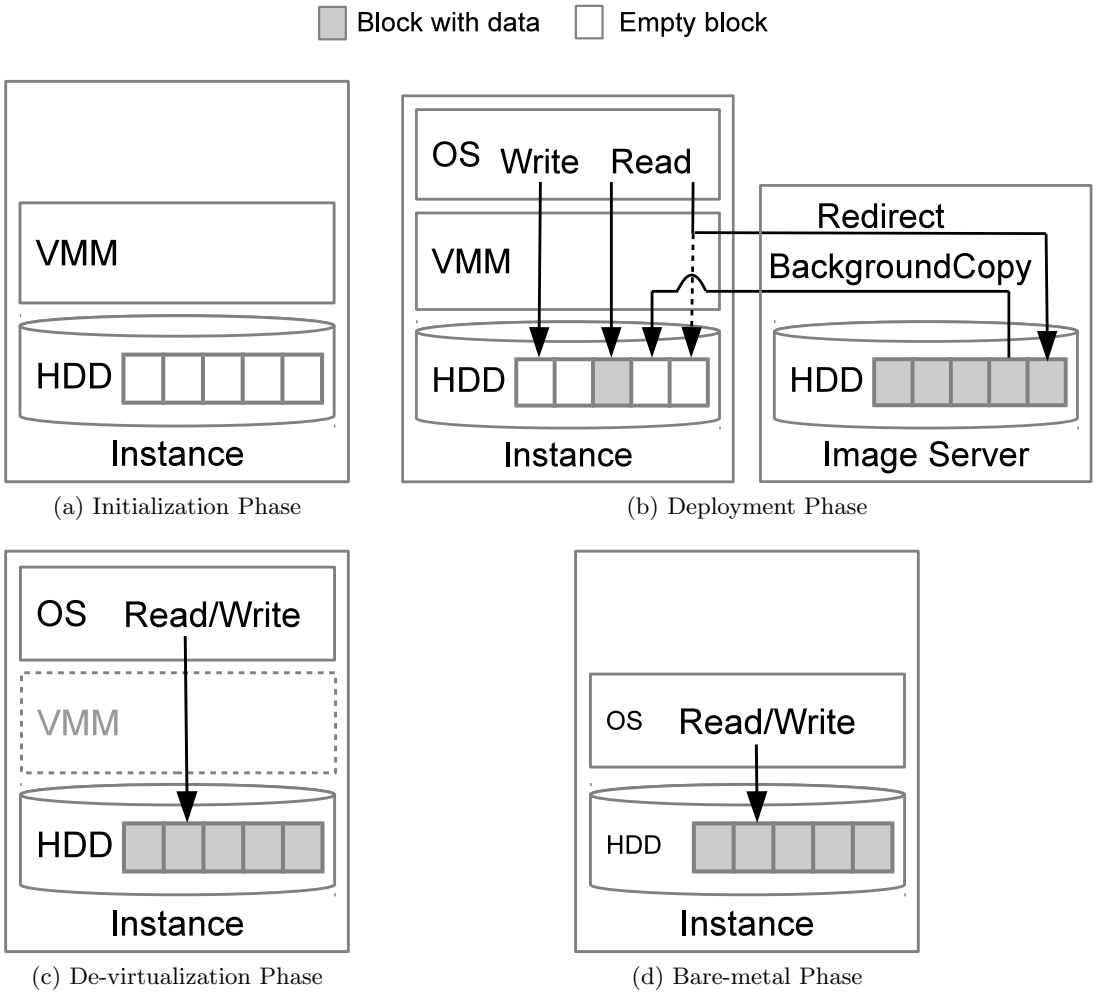
Creating virtual devices with the same device interfaces as those of physical devices will ease de-virtualization while preserving OS transparency. However, emulating a machine with an identical hardware interface as the underlying physical machine, including all devices (e.g., chipset and ACPI functions) is costly. In addition, synchronizing the internal states of virtual devices with physical devices is an open problem. Therefore, seamless and transparent conversion from virtual instances to physical instances is difficult.

NoHype [50] allows the removal of the virtualization layer to eliminate attack surfaces after booting guest OSs. Microvisor [60] demonstrates run-time de-virtualization for online maintenance of servers. Pass-through-based VMMs that allow direct control of I/O devices from the guest OS [75, 77] can also achieve de-virtualization. However, current de-virtualizable VMMs do not support device sharing between the guest OS and VMM. Therefore, the VMMs do not have access to the local disk to copy the OS image.

The proposed VMM achieves seamless de-virtualization and eventual bare-metal performance, which differs from conventional VMMs. The proposed VMM also achieves low-overhead device sharing with the guest OS, which differs from existing de-virtualizable or pass-through VMMs.

4.3 Design

In this section, we first explain the deployment process and then describe the VMM functionalities, such as I/O mediation, background copy, and de-virtualization.



(a) Initialization Phase (b) Deployment Phase

(c) De-virtualization Phase (d) Bare-metal Phase

Figure 4.1: The four phases to deploy an OS.

4.3.1 Deployment process

The deployment process is divided into four phases: *initialization phase*, *deployment phase*, *de-virtualization phase*, and *bare-metal phase*.

Initialization phase: In this phase, the VMM itself boots up on a target machine. It can be either network boot or local boot from the secondary local disk. We assume that the processors support hardware-assisted virtualization such as Intel VT-x and AMD-V [46, 17], and the VMM in this phase initializes processors and data structures on memory to turn on

the virtualization functionalities to start as a VMM. Unlike conventional VMMs, the VMM does not initialize most of hardware devices and leaves them for the guest OS so that the guest OS recognizes the hardware devices and initializes them by itself.

We assume that each physical machine has a dedicate network channel for OS streaming deployment. Although it is possible to share the network with the guest OS, the network performance may significantly be affected by the network traffic of the disk image. Therefore, the VMM occupies a network card connected to the storage server and initializes it in this phase. The VMM also hides the network card from the guest OS so that the guest OS does not recognize and try to initialize it.

In this phase, the (primary) local disk is uninitialized: all blocks of the disk are empty and not filled with the disk image of the guest OS (see Figure 4.1a).

Deployment phase: In this phase, the VMM boots the guest OS over network so that the guest OS immediately becomes ready to use. The VMM first fetches the boot sector of the target OS from the disk image on the storage server, and stores it on the local disk. The VMM also copies the boot sector to the memory location as if it is loaded by BIOS, and pass the control to it. Since the boot loader assumes that it is booting from a local disk, it tries to load OS image from the local disk via BIOS.

Since then, the VMM intercepts disk access and performs copy-on-read. When the guest OS attempts to read empty blocks of the local disk, the VMM redirects the read access to the storage server (see "Redirect" in Figure 4.1b). The data read from the server are returned to the guest OS, and written to the local disk as well for future use. The written blocks are marked as filled. If the guest OS attempts to access filled blocks, the access passes-through the VMM and are served by the local disk (see the solid-line arrow from "Read" in Figure 4.1b). Note that write accesses always pass-through the VMM and are not redirected to the server (see the solid-line arrow from "Write" in Figure 4.1b), because the disk image on the server is a template and the local disk is an instance. With this

behavior, the guest OS can startup by only copying, from the server, minimal set of disk blocks which are necessary for OS boot.

In parallel with the copy-on-read behavior, the VMM actively fills empty blocks with background copy (see "Background Copy" in Figure 4.1b). The VMM moderates the speed of background copy to reduce the performance interference. With this behavior, the local disk is gradually filled with the disk image and the guest OS eventually gains the near-bare-metal disk performance. Note that we use the identical block address space for both the local disk and the disk image on the server: the first sector of the disk image corresponds to the first sector of the local disk. It allows seamless elimination of the VMM.

De-virtualization phase: After the entire image of the OS has been copied to the local disk, the VMM no longer needs to intercept I/Os and all disk accesses just pass-through the VMM (see Figure 4.1c). In this phase, the VMM disables I/O interception and all related functions. Then, the VMM stops all virtualization functions and disappears from the beneath of the guest OS. Since the VMM directly exposed the physical hardware interface to the guest OS, the guest OS are not aware of the de-virtualization (unless intentionally try to detect it), and continues its operation without interruption.

Bare-metal phase: In this phase, the OS directly runs on a physical machine without virtualization overhead (see Figure 4.1b). The disk state is same with the result of the local installation of the OS.

4.3.2 Background copy

To fill the entire local disk with the OS image, the VMM actively retrieves the data for empty blocks in the background. Since the retrieval and disk-write rates will differ, the VMM uses a pair of threads, *retriever thread* and *writer thread*, that are connected by a FIFO queue. The retriever thread retrieves data for empty blocks from the server and pushes them to the queue. The writer thread then pops and writes the data to the local

disk. The VMM fills blocks in order from low to high LBA. However, to minimize seek, the VMM changes the address adjacent to that of the last-accessed block if the guest OS accessed the disk.

In devices that have multiple request queues, a consistency problem may occur if a local disk is shared by the VMM and guest OS. To illustrate this problem, let us consider the following example. The VMM attempts to fill an empty block and send a request to the server. Before the response arrives, the guest OS issues a write request to the same block. In this case, the data from the guest OS is the most recent and should remain in the local disk. However, the response comes after the write from the guest OS; therefore, simply putting write requests into a queue in a FIFO manner breaks the consistency.

To mitigate this problem, the VMM holds a bitmap to manage the status of each disk block and atomically checks the status to prevent the VMM from writing to a filled block. In case of shutdown and reboot, the VMM saves the bitmap on the local disk. The VMM uses an unused region on the local disk (such as unallocated space between two partitions) to save the bitmap. To prevent accidental overwrite of the bitmap by the guest OS, the VMM hides and protects the region.

To avoid performance interference, the background copy speed should be moderated. If the write frequency is too high, guest storage performance degrades significantly. If it is too low, the deployment phase takes a long time. To solve this problem, the VMM adjusts the write frequency based on the guest OS load and three configurable parameters: *guest I/O frequency threshold*, *VMM-write interval*, and *VMM-write suspend interval*. If the disk I/O frequency becomes higher than the value of the guest I/O frequency threshold, the VMM waits for the time specified by the VMM-write suspend interval. Otherwise, the VMM writes blocks at the interval specified by the VMM-write interval.

Due to this moderation, the VMM will not perform excessive background copy operations during OS startup. As an optimization technique, we could configure the moderation function to prefetch the disk regions required for OS startup via a background copy opera-

tion, which would potentially boost OS startup time. However, we do not assume that this prefetch information is generally available.

4.3.3 De-virtualization

To make de-virtualization easy, we directly expose physical hardware to the guest OS. For example, all physical CPU cores and I/O devices, including PCI devices and interrupt controllers, and other hardware functionalities such as ACPI are exposed to the guest OS.

Memory address space is primarily identity-mapped; a guest physical address is identical to the machine physical address. One exception to this is the physical memory region for the VMM. The VMM requires several tens of megabytes of memory and reserves the required region by manipulating the BIOS function such that the guest OS does not allocate this region for itself. The VMM also uses nested paging to prevent the guest OS from accidentally corrupting the memory region.

In the de-virtualization phase, the VMM turns nested paging off to eliminate the paging overhead. At this time, the VMM needs to invalidate the TLBs on all CPUs. Unfortunately, the VMM cannot use inter-processor interrupts (IPI) for TLB shootdown because it does not manage interrupt controllers. Fortunately, page mapping is constant over the life time of the VMM and does not cause a consistency problem among CPUs. Therefore, the VMM eventually turns paging off and invalidates TLBs on all CPUs at different timings. After all CPUs turn paging off, the VMM terminates virtualization.

4.4 Implementation

In this section, we describe our prototype implementation. First, we describe the CPU virtualization and network storage protocol. Then, we discuss the implementation status.

4.4.1 CPU virtualization

We assume that the CPU is equipped with a hardware-assisted virtualization feature (Intel VT-x or AMD-V). To minimize overhead, we make the CPU to run the guest OS as much as possible and switch to the VMM only when the minimum required events occur. The switch from the guest OS to the VMM is called *VM exits*.

We identified several events that require VM exits. To implement I/O mediation, PIO and MMIO instructions for the storage devices need to trigger VM exits. To detect boot, Startup IPIs and INIT Signals need to trigger VM exits. To detect changes in paging and the processor mode of the guest OS, CR0 (PE, ET, WP, AM, NW, CD, PG) and CR4 (PSE, PAE, PGE, VMXE, SMXE, PCIDE, SMAP, SMEP) bit changes should also trigger VM exits. To trigger VM exits on MMIO, the VMM uses nested paging (EPT on Intel VT-x or NPT on AMD-V) and keeps the target memory regions unmapped. To cause other VM exits, the VMM configures the data structure to control the virtual machine (VMCS on Intel VT-x or VMCB on AMD-V). Note that the CPUID instruction unconditionally causes VM exits; however, this instruction occurs infrequently.

To poll the devices in I/O mediation, the VMM needs to be scheduled periodically. With Intel VT-x, we exploit *preemption timer* to schedule threads. The preemption timer is a timer feature supported by latest Intel processors that unconditionally causes VM exits at a specified interval. It allows fine-grained control of the timing of VM exits with CPU clock cycle granularity. Polling intervals are estimated from recent average network round trip times and I/O latency times. This achieves reasonable performance. If the preemption timer is not available, the VMM enables VM exits on hardware interrupts and uses a technique similar to soft timers [18].

4.4.2 Network storage protocol

The VMM requires a network storage protocol to redirect I/O requests to the storage server. To reduce overhead and improve transparency, we should select a protocol that allows the

conversion of I/O requests to network packets with minimal effort. File-level protocols, such as NFS and CIFS, are therefore not suitable, and a block-level protocol is preferable.

We extended the AoE protocol [73], which has greater affinity with ATA devices, although other protocols could be used. This protocol has a header that contains the set of device registers, and the VMM can easily convert the device register set to an AoE header. Data to be read is transferred as a payload of a packet with the header. If the transferred data is too long for a single Ethernet packet, the VMM splits the data into multiple AoE fragments. In this case, the VMM sets the tag field in an AoE header to determine the offset of a received fragment.

To improve performance, we modified the AoE protocol to support jumbo frames. We also designed and implemented a retransmission capability to tolerate packet loss. We use *vblade* [1] as the basis of our AoE server implementation. However, the original vblade cannot fully utilize network bandwidth because it is single-threaded and becomes a performance bottleneck when the VMM sends a significant volume of read requests. Therefore, we implemented a thread pool to vblade.

4.4.3 Implementation status

We implemented a prototype VMM based on BitVisor [77]. It supports x86 environments with Intel VT-x or AMD-V processors. We implemented storage I/O mediators for IDE and AHCI disk controllers and the extended version of the AoE protocol in the VMM and server. We confirmed that the VMM can deploy both Windows (Vista, 7, 8.1, Server 2008) and Linux (Ubuntu 10.04 and later, and CentOS 6.3 and later).

The sizes of storage I/O mediators are 1,472 LOC for IDE and 2,285 LOC for AHCI. We assume that we can use a dedicated NIC for streaming deployment to avoid performance interference and implemented small network drivers for Intel PRO/1000, x540, Realtek 816x, and Broadcom NetXtreme network adapters. The implementation cost is limited because we need minimal functions to send and receive packets with polling: the PRO/1000 driver

has 718 LOC, x540 driver has 614 LOC, RTL816x has 757 LOC, and NetXtreme has 620 LOC. Our implementation is based on BitVisor 1.4 and modified the core part of BitVisor by only 3,576 LOC. The total is approximately 27 KLOC. When adding storage I/O mediators for new devices, the core part does not need to be modified.

Our current implementation has some limitations. First, the memory region used for the VMM is not released back to the guest OS after de-virtualization. We can mitigate this by implementing memory hot-plug features. Second, we hide hardware-assisted virtualization features from the guest OS because we have not implemented nested virtualization. However, nested virtualization is known to be implementable [24]. Third, we do not support VMXOFF (disabling the VMM mode). Supporting VMXOFF requires some implementation effort in the VMM to restore the guest processor state in memory (VMCS or VMCB) to the real processor without using the VMExit instruction. However, it is theoretically possible. Therefore, these limitations are not essential and we plan to implement them in the commercial version of our system.

4.5 Performance evaluation

In this section, we show the experimental results of evaluating the performance of our system. We first show the result of measuring the OS startup time to demonstrate that our VMM achieved quick startup of bare-metal instances. We next explain the result of database benchmarks to demonstrate that our VMM achieved low-overhead streaming deployment and eventual bare-metal performance. Then, we show the result of a kernel compile benchmark to illustrate the overall performance of our system. After that, we show the result of micro benchmarks to reveal the effects on threads, memory, storage and network performance. Finally, we show the behavior of the moderation in background copy.

We used a cluster of machines (originally used for HPC applications in practice), each of which was a FUJITSU PRIMERGY RX200 S6 with two Intel Xeon X5680 processors (3.33 GHz, 2 $\times$ 6 cores, hyperthreading disabled), 96-GB memory, a Mellanox MT26428

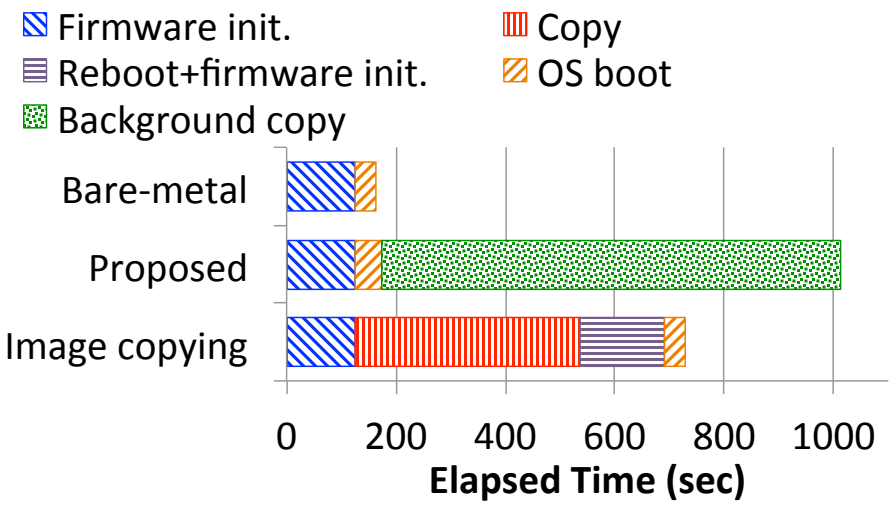


Figure 4.2: OS startup time

Infiniband card (4X QDR), and a Seagate Constellation.2 ST9500620NS (500GB/7200 rpm) SATA hard drive. It also had two Intel 82575EM gigabit NICs, one of which was dedicated to the VMM. These machines were connected via a FUJITSU SR-S348TC1 Gigabit Ethernet switch whose maximum transmission unit (MTU) was 9000 bytes, and a Mellanox Grid Director 4036E Infiniband switch. In all experiments, we deployed Ubuntu 14.04 with Linux kernel 3.13.0. For comparison, we used KVM (Linux kernel 3.9.0) with the ELI patch [41].

4.5.1 OS startup time

We first evaluate the OS startup time. We measured the startup time of an instance on a bare-metal machine (Baremetal) and that on our VMM (Proposed). For comparison, we also measured the time of image copying over iSCSI (Image Copy). In addition, we measured the startup time of network boot using NFS (NFS Root), and that of a guest OS on KVM using a disk image over either NFS or iSCSI. In all experiments, the network used was the gigabit Ethernet and the size of OS image deployed was 32-GB.

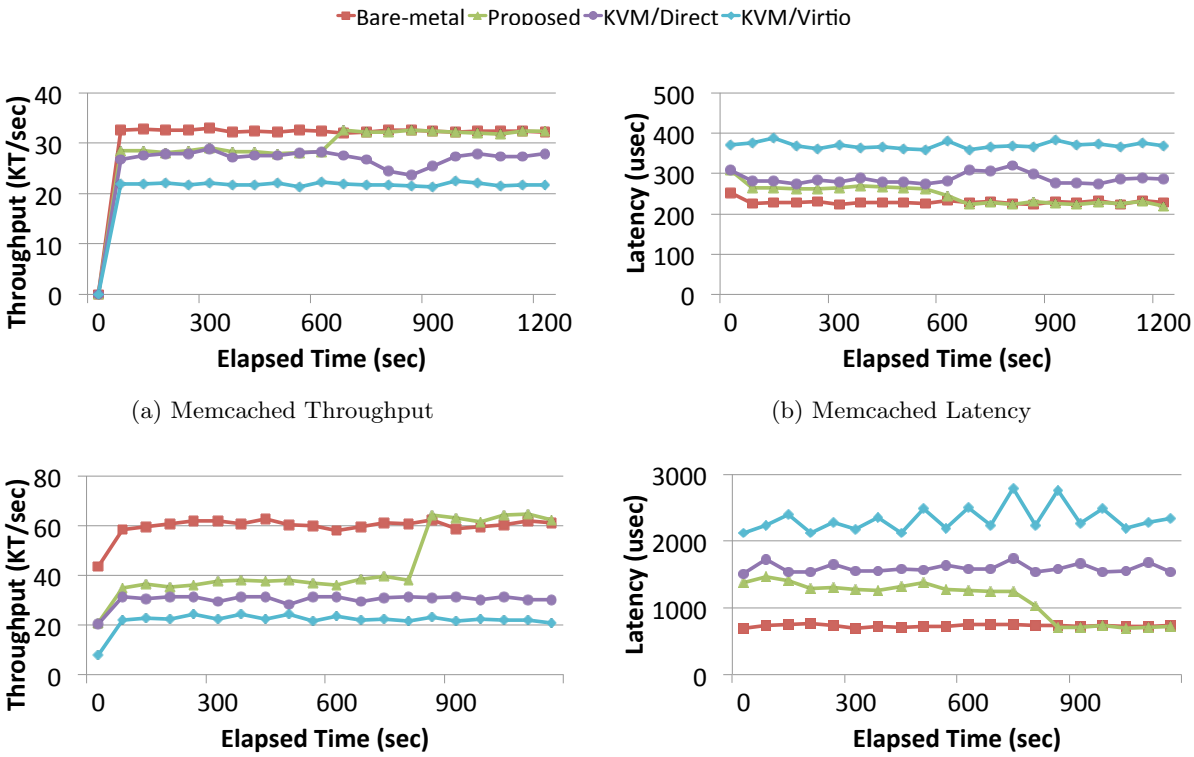
Figure 4.2 shows the results. The firmware initialization on our machine took 133

seconds, and the OS boot on the bare-metal machine took 29 seconds. On our VMM, the startup time of a bare-metal instance was 63 seconds (including 5 seconds to boot our VMM). On the other hand, image copying took 544 seconds (50 seconds to boot the installer OS over the network, 320 seconds to transfer the disk image, 145 seconds to restart, and 29 seconds for the OS boot from the local disk). Therefore, our VMM started up a bare-metal instance 8.6 times faster (excluding the first firmware initialization) or 3.5 times faster (including the first firmware initialization) than image copying.

The network transfer rate in the image copying was approximately 100 MB/sec, being limited by the bandwidth of gigabit Ethernet. Therefore, using SSDs will not speed up the image copying in this case. Using a faster network such as Infiniband or 10Gbit Ethernet may reduce the transfer time. However, when multiple instances are started-up at the same time, the performance of the storage server will be saturated and the transfer time will not be reduced so much. On the other hand, our VMM transferred only 72MB of the disk image while booting the OS in 58 seconds, so the average rate was 1.2 MB/sec. This means that there is more room to scale-up the number of instances booted simultaneously. Therefore, our VMM will keep the advantage of fast startup time even if SSDs or faster networks are available.

In Figure 4.2, the OS startup time of network boot was 49 seconds, which was slightly faster than that on our VMM. However, it did not deploy the OS image to the local disk and took continuous overhead in a database workload. KVM took 30 seconds to boot itself. Our VMM is designed to boot fast, as described in 4.3.1, and its boot time (5 seconds) was 6 times faster than that of KVM. The startup time of the guest OS on KVM was 42 seconds in the NFS case and 55 seconds in the iSCSI case. The startup time of the OS on our VMM, 58 seconds, was comparable to that in the iSCSI case. Moreover, when we compared the startup time including that of VMMs, our VMM was 1.14 times and 1.35 times faster than KVM with NFS and iSCSI.

From these results, we confirmed that our VMM achieved quick startup of bare-metal



(a) Memcached Throughput

(b) Memcached Latency

(c) Cassandra Throughput

(d) Cassandra Latency

Figure 4.3: Throughput and latency of database benchmark

instances.

4.5.2 Database benchmark

To evaluate the performance throughout the deployment and de-virtualization phase, we simulated a situation in which a user launches a new instance with a NoSQL database that serves data to clients, and traced the performance shift.

We tested two databases widely used in cloud computing: *memcached* and *Cassandra* [56]. Memcached is an in-memory database often deployed to improve data-serving latency for read-intensive workloads. Cassandra is a database that allows high throughput write access for update-intensive workloads. Both databases are designed to be scalable, spanning multiple instances depending on the amount of data to be handled. We used

Yahoo! Cloud Serving Benchmark (YCSB) [35] for the performance evaluation by sending continuous requests from another instance to the target database instance. For memcached, we recreated a read-intensive workload with 95% reads and 5% writes ratio and, for Cassandra, a write-intensive workload with 30% reads and 70% writes. We created a 32-GB OS image with the database configured.

Figure 4.3 shows the throughput and latency results of memcached and Cassandra benchmarks. The horizontal axis in each figure indicates the elapsed time from the beginning of the YCSB test. The vertical axis indicates the ratio to the average performance on the bare-metal machine. We performed the benchmarks on our proposed VMM while streaming OS deployment was in progress and after de-virtualization (Proposed), and on a KVM instance with pass-through access to infiniband cards and para-virtual storage devices (KVM). Note that KVM did not perform streaming OS deployment and the overhead of sharing devices were not incurred.

In the memcached benchmark, even while our VMM was in the deployment phase and performing background copy, it showed slightly higher throughput than KVM (see Figure 4.3a); the average throughput was 34.6 kilo-transactions per second (KT/sec) on our VMM, which was 94.8% of the bare-metal performance and 102% of the KVM performance. As for the average latency during the deployment phase, our VMM showed lower latency than KVM (see Figure 4.3b); 291 μsec on our VMM, which was 7% slower than the bare-metal performance and 14.8% faster than the KVM performance.

The primary reason of the performance degradation by our VMM is TLB pollution; the number of TLB misses increased up to 5 times and the latency on TLB misses doubled due to the two dimensional page walks of nested paging. The VMM also consumed 6% of the total CPU time: 5% was for handling threads in OS streaming deployment and 1% was for the VMM core itself.

In our measurement, the deployment phase took 16 minutes. Therefore, in Figures 4.3a and 4.3b, the throughput increased and latency decreased after 990 seconds have elapsed;

the throughput reached 34.6 KT/sec and the latency reduced to 281 μsec, which were identical to those of the bare-metal performance. There was no suspension or performance degradation during the phase shift. Therefore, we confirmed that our VMM achieved seamless de-virtualization and eventual bare-metal performance.

The performance of the Cassandra benchmark on our VMM was slightly lower than that on the memcached benchmark. In the deployment phase, the throughput on our VMM was lower than that on KVM (see Figure 4.3c); the average throughput was 51.4 KT/sec, which was 91.4% of the bare-metal performance and 98.7% of the KVM performance. The average latency on our VMM was also longer than that on KVM (see Figure 4.3d); it was 2,609 μsec on our VMM, which was 7% slower than the bare-metal performance and 3% slower than the KVM performance. The deployment phase took 17 minutes (1020 seconds), which was longer than that in the memcached benchmark because the Cassandra benchmark was more write-intensive. However, after de-virtualization, the throughput on our VMM increased to 60.0 KT/sec and latency decreased to 2,443 μsec, which were almost the same with that of the bare-metal performance.

4.5.3 MPI benchmark

To evaluate the performance effects of our VMM on cluster computing, we ran micro benchmarks of basic MPI operations on the HPC cluster. The cluster we used consists of 10 machines connected via a Infiniband switch and OSs are configured to use MPICH2. We used OSU Micro-Benchmarks [10] and measured the latency of MPI collective communications among all machines. We ran the tests on the cluster with all OSs running on our VMM (Proposed). We ran the same tests with all OSs running on KVM (KVM). We compared the result with that on the cluster of bare-metal machines.

Figure 4.4 shows the results. While most results on our VMM were almost the same with that on bare-metal machines, KVM incurred large overhead on many tests. On Allgather, the latency on KVM was 235% of that on bare-metal machines, while that on our VMM

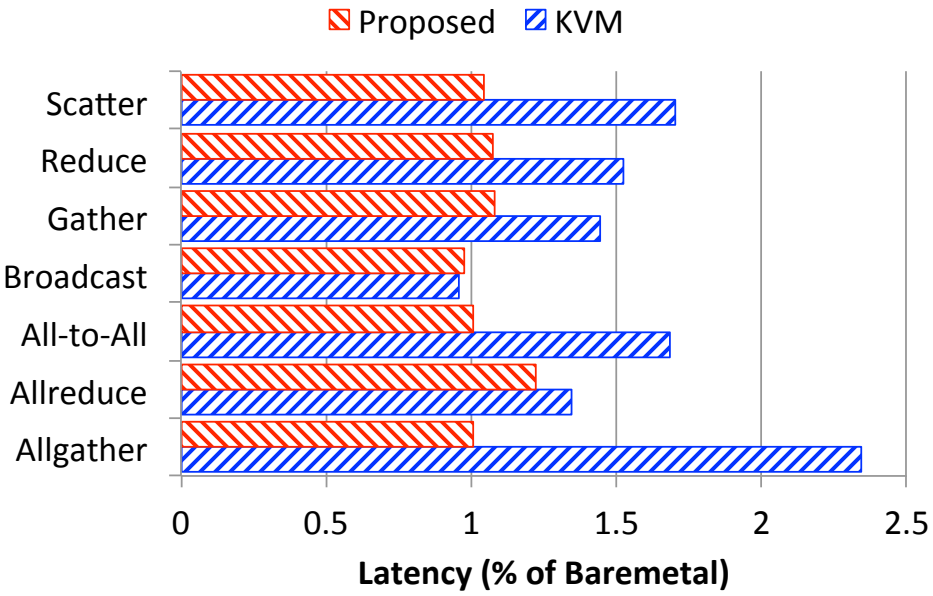


Figure 4.4: MPI benchmark

was almost identical to that on bare-metal machines. On Allreduce, our VMM incurred 22% overhead but KVM incurred 35% overhead. We guess the latency overhead of Inifiniband (see Section 4.5.5) largely contributes to this severe overhead on the MPI benchmarks because cluster machines need to exchange messages very frequently.

4.5.4 Kernel-compile benchmark

To illustrate the overall performance of our system, we used *kernbench*, which compiled a Linux kernel version 2.6.32 with 12 running jobs in parallel (`make -j 12`), and measured the total elapsed time on the bare-metal machine (Baremetal), on our VMM while deploying an OS image (Deploy), on our VMM after de-virtualization (Devirt), and on KVM (KVM).

The benchmark results are shown in Figure 4.5. The kernel compile took approximately 16 seconds on the bare-metal machine. While OS deployment was in progress, our VMM increased the compile time by 8%. The main reason of this overhead was the cost of sharing the storage device between the guest OS and the VMM by I/O multiplexing. However,

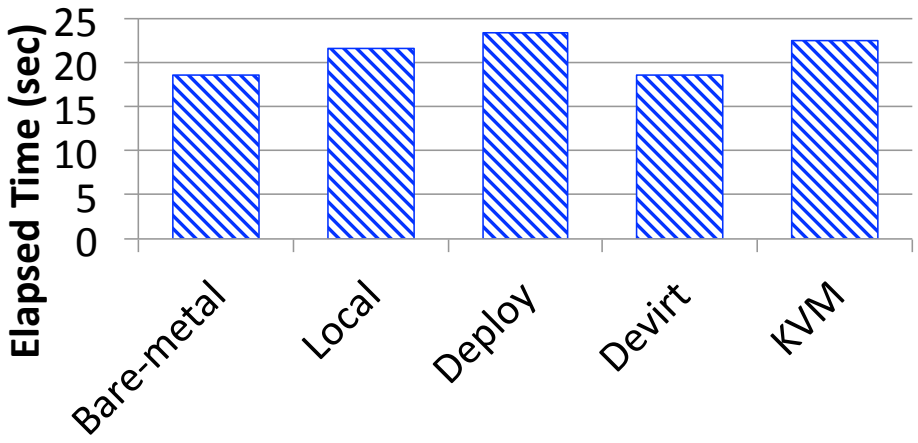


Figure 4.5: Kernel compile benchmark

its performance impact was limited because of the effects of the moderation of background copy described in Section 4.3.2.

KVM (KVM) increased the compile time by 3%. This result does not include the cost of streaming deployment because KVM did not perform it. Therefore, this was pure virtualization overhead of KVM. After de-virtualization on our VMM, the compile time became identical to that on the bare-metal machine. These results show that the moderation of background copy is effective, and de-virtualization has the benefit of reducing the overhead of virtualization.

4.5.5 Micro benchmarks

We performed three micro benchmarks to measure threads, memory, storage and network performance.

Thread and memory benchmark

To evaluate the effect on threads and memory performance, we used SysBench [54]. The thread benchmark repeatedly performed a sequence of acquire-yield-release operations 1,000 times on 8 mutex locks from multiple threads and measures the average execution time. We

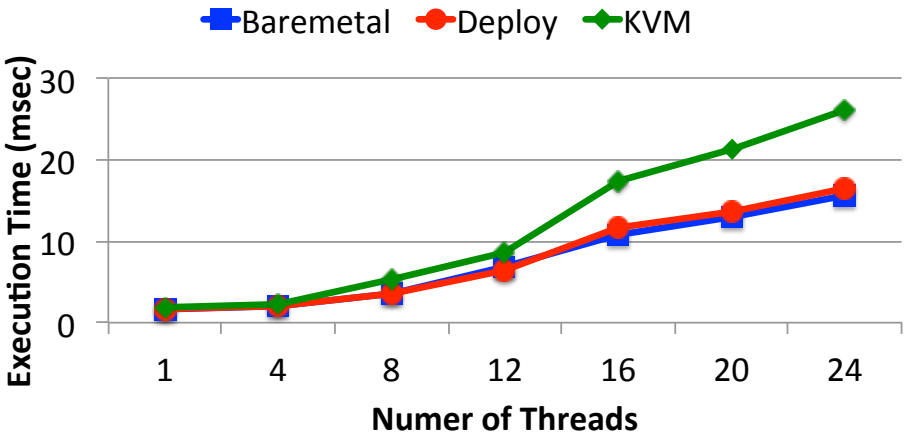


Figure 4.6: Threads performance

changed the number of threads from 1 to 24. The memory benchmark repeatedly allocated a memory block and wrote data to the block until the total amount of data written reached 1 MB. We changed the size of memory block from 1 KB to 16 KB. We performed the benchmarks on the bare-metal machine (Baremetal), on our VMM while deploying an OS image (Deploy), and on KVM (KVM). We configured KVM to use processor pinning to avoid overhead of scheduling virtual processors, and use 2-GB huge paging to reduce overhead of nested paging.

Figure 4.6 shows the results of the thread benchmark. As the number of threads increased, the overhead of KVM significantly increased (68% on 24 threads). We guess this overhead was incurred by the lock-holder preemption problem [84]; a thread holding a lock was scheduled off from a virtual CPU and other threads must wait until the thread was scheduled back. This virtualization overhead could become a significant problem for highly-concurrent applications. On the other hand, our VMM incurred only 6% overhead on 24 threads even while streaming deployment was in progress. This was because our VMM traps only minimum events required for streaming deployment, and the frequency of VM exits were much lower than conventional VMMs.

Figure 4.7 shows the results of the memory benchmark. While the throughput on our

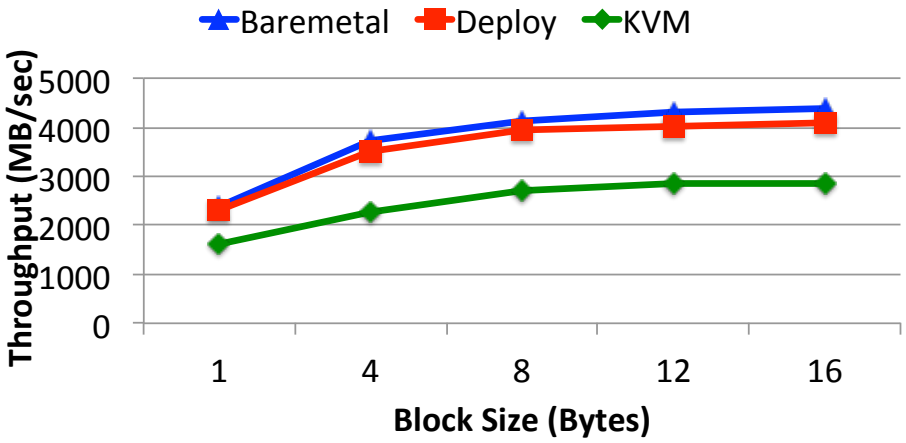


Figure 4.7: Memory performance

VMM incurred only 6% of overhead, KVM incurred 35% of overhead on 16-KB block size. We guess this was caused by the cost of nested paging and cache pollution by the VMM including the host OS. Note that after de-virtualization, the performance on our VMM became identical to the bare-metal case in both benchmarks.

Storage benchmark

To evaluate the effects on storage performance, we measured the throughput and latency of disk access on the guest OS. To measure storage throughput, we used Flexible IO Tester (fio) [19] and read 200-MB of data with 1-MB block size using direct I/O and Linux native asynchronous I/O engine (libaio). To measure storage latency, we used ioping [7] and read 1 MB of data 100 times with 4K byte block size. We measured the performance on the bare-metal machine (Baremetal), on our VMM during OS streaming deployment (Deploy), and that after the de-virtualization (Devirt). For comparison, we also measured the performance on a network-booted OS (Netboot), the guest OS on KVM with local disk (KVM/Local) and NFS (KVM/NFS).

Figure 4.8 shows read & write throughput. On the bare-metal machine, the read and write throughput was 116.6 MB/sec and 111.9 MB/sec, respectively. The read throughput

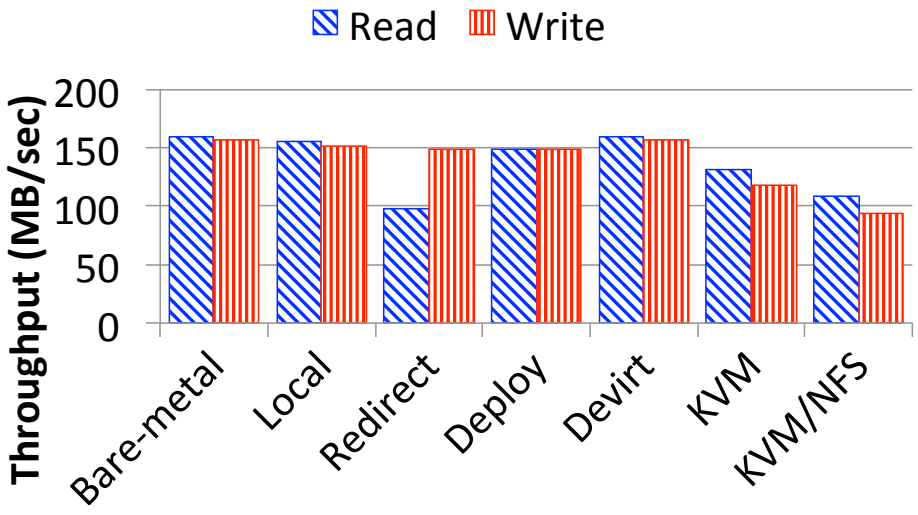


Figure 4.8: Storage throughput

decreased by 4.1% in the Deploy case and 1.7% in the Devirt case. The write throughput was almost the same as the bare-metal case. This result suggests that the moderation of the speed in background copy worked effectively and de-virtualization achieved almost bare-metal performance. On the other hand, KVM decreased the read throughput by 10.5% in KVM/Local and 12.3% in KVM/NFS, and decreased write throughput by 13.6% in KVM/Local and 15.3% in KVM/NFS, respectively. These overhead would be mainly caused by virtual I/O devices.

Figure 4.9 shows storage latency. Our VMM in the Deploy case increased the average latency by 4.3ms. This increase in time was for the blocking time in accessing storage devices. If I/O requests from the VMM are being handled, the requests from the guest OS are queued and blocked. This blocking time is measured as the overhead in the latency. However, in the Devirt case, there was no overhead in the latency (actually slightly faster). In this case, no instructions, except for CPUID, triggered VM exits. The intervals of the CPUID exits ranged from a couple of seconds to minutes, and their overhead was negligible. These results confirmed that the VMM achieves bare-metal performance.

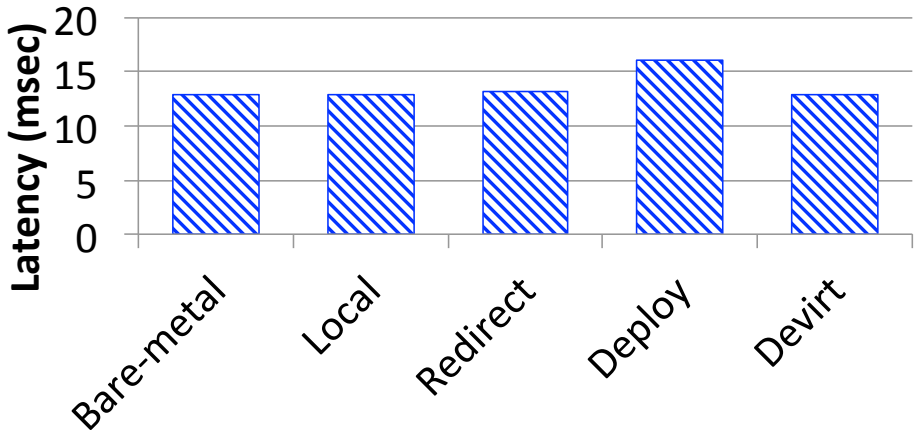


Figure 4.9: Storage latency

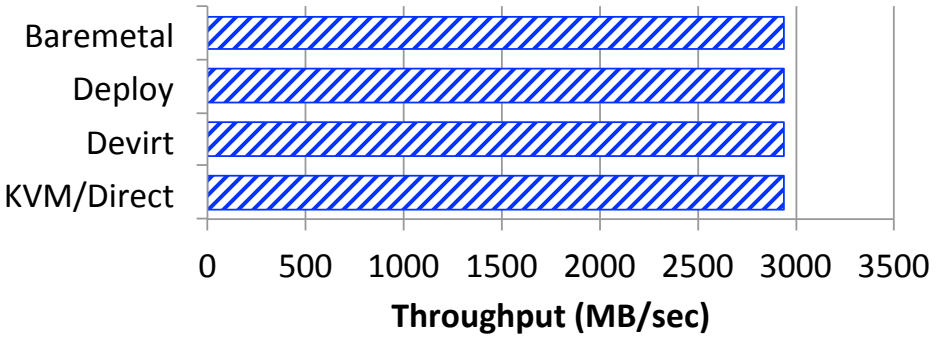


Figure 4.10: Infiniband throughput

Network benchmark

To evaluate the effects on network performance, we measured the raw Infiniband performance. We used the *ib_rdma_bw* and *ib_rdma_lat* commands in the *perftest* package of the Open Fabrics Enterprise Distribution. These commands sent 64-KB packet 1,000 times and measured the throughput and latency of RDMA over Infiniband. We measured the performance on the bare-metal machine (Baremetal), on our VMM while deploying an OS image (Deploy), on our VMM after de-virtualization (Devirt), and on KVM with direct device assignment (KVM/Direct).

Figure 4.10 shows the throughput and Figure 4.11 shows the latency. In our environ-

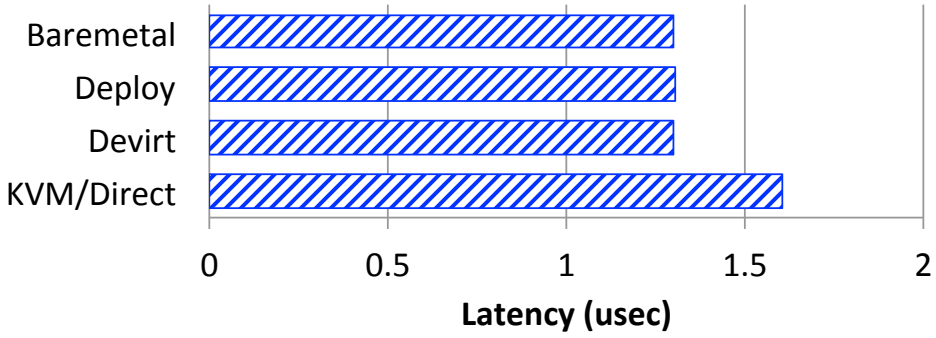


Figure 4.11: Infiniband latency

ment, there was no difference in throughput. At this time, CPU utilization was also very low. This means that network was saturated, and the virtualization overhead was hidden by the command queuing of the RDMA hardware. However, KVM increased the latency by 23.6%. The cause of this would be the overhead of IOMMU, cache pollution pollution, and nested paging. This overhead could become a significant problem in latency-sensitive applications. On the other hand, our VMM incurred only negligible overhead (less than 1%). Therefore, our VMM achieved bare-metal performance in the Infiniband environment.

4.5.6 Moderation of background copy

In the deployment phase, our VMM moderates the speed of background copy by adjusting the interval between each block write. To clarify the relationship between the interval and the storage performance of the guest OS, we measured read and write throughput of the guest OS and the write throughput of the VMM with changing the intervals of the VMM writes. The block size written by the VMM was 1024 KB.

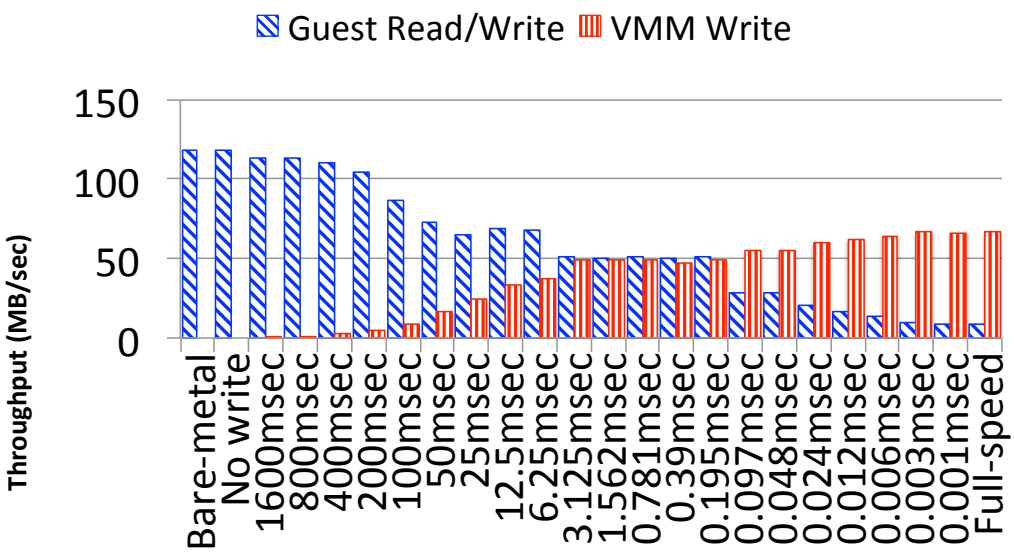
Figure 4.12a shows the relationship between read throughput of the guest OS and write throughput of the VMM. Figure 4.12b shows the relationship between write throughput of the guest OS and that of the VMM. On the both graphs, the left-most bar shows the bare-metal throughput (Bare-metal). We reduced the intervals of VMM writes from 1 sec to 1 μsec, and finally, the VMM issued write requests without intervals (Full-speed).

As the intervals of VMM writes was reduced, the guest throughput gradually decreased, and the VMM write throughput gradually increased. The sum of both throughput did not reach the bare-metal throughput in either case because the VMM used a polling-based storage access, and the guest OS and VMM wrote different regions of the disk that increased seek overhead. Overall, however, the total performance was fairly reasonable and adjusting the write intervals was an effective approach to moderate the speed of background copy.

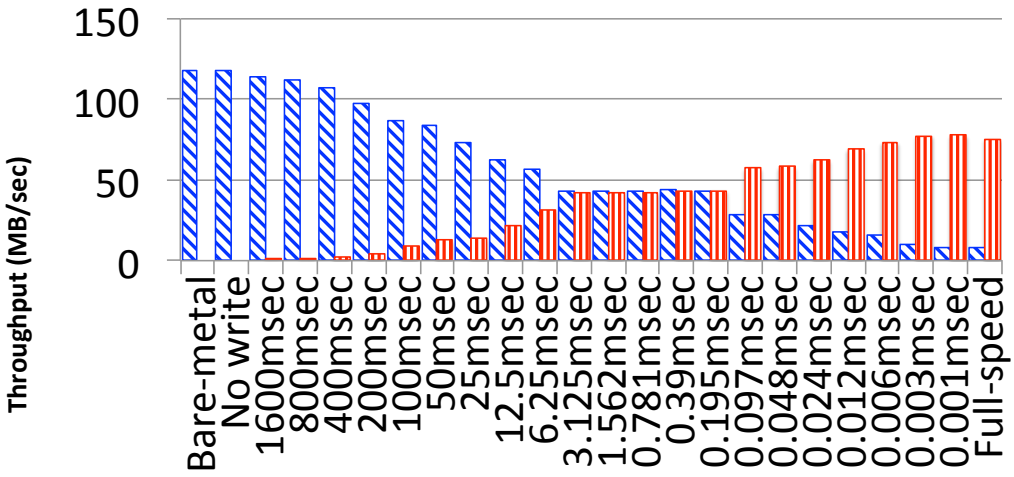
4.6 Summary

In this paper, we presented a lightweight de-virtualizable VMM that supports fast startup of bare-metal instances. The VMM transparently performs OS streaming deployment, and disappears after the completion of the deployment. To achieve seamless and transparent de-virtualization, the VMM directly exposes the physical hardware interfaces to the guest OS while it performs polling-based device-interface-level I/O mediation to allow background copy of disk images.

We implemented a prototype x86 VMM that supports the multiplexing of ATA/AHCI devices, the redirection of ATA commands to the storage server with AoE protocol, and background copy of disk image. We confirmed that the VMM could deploy both Windows and Linux without modifications, and could startup a web service running on a Linux instance in only 48 seconds. Deploying a 50GB OS image to a instance was completed in 30 minutes under a database workload. After the de-virtualization, the VMM incurred zero overhead.



(a) Guest-read and VMM-write I/O throughput



(b) Guest-write and VMM-write I/O throughput

Figure 4.12: Guest and VMM I/O throughput with changing the intervals of the VMM writes

Chapter 5

Application: Background Full-Disk Encryption

5.1 Introduction

To prevent data breaches, many organizations deploy full disk encryption to their computers. Many OS-based approaches to full disk encryption are presented in both industry and academic fields [83, 64, 49, 28, 62, 31, 89], and they are well accepted in practical situations because they allow simple and rapid deployment. On the other hand, several VMM-based approaches are also proposed because they have significant advantages over OS-based ones, such as OS independence and realizing more secure environments [58, 71, 52].

Many such encryption systems can postpone the process of entire disk encryption until after installation by providing *background encryption*, allowing users to continue to use their PCs during encryption [28, 62, 31, 89]. However, despite of their security advantages, existing VMM-based encryption systems cause performance degradation and limited device functionality because of virtualization. To mitigate this problem, we leverage the para pass-through VMM and performs background encryption functions with the VMM. However, the conventional para pass-through VMM does not support rich I/O controllability enough to

perform background encryption; that is, the VMM reads and writes the target local disk while allowing the OS to access the disk. Therefore, we extend the design of the para pass-through VMM and allow I/O multiplexing, properly arbitrating I/Os from both the VMM and the OS.

To support background encryption, moderating encryption speed by watching OS activities is indispensable to avoid significant penalty on the guest OS performance and user's experiences without losing encryption performance. Unfortunately, watching OS activities in VMMs is more difficult than in OSs themselves because VMMs can not directly obtain OS activity information but only observe low-level interactions between OSs and hardware devices. In addition, to avoid costly P2V operations, background encryption must be performed in a compatible way with existing disk images and OS configurations so that encryption processes do not interfere with the operations of existing OSs and users.

In this paper, we present background encryption with the para pass-through VMM that achieves high performance and full device functionality. To achieve efficient background encryption, the VMM moderates the degree of encryption speed by appropriately guessing OS activities from the observations of low-level events such as disk I/Os, keyboard/mouse I/Os, and external interrupts. The VMM transparently performs interposition of disk I/Os to encrypt and decrypt data transferred between the guest OS and disk devices in parallel with background encryption that reads-encrypts-writes unencrypted sectors at appropriate timings based on the guess of guest OS activities.

We implemented our scheme based on BitVisor [78], a thin VMM for enforcing I/O device security. Our experimental results on Windows 7 showed that application benchmark scores of PCMark Vantage were not significantly affected by the background encryption and the overhead on sequential disk access throughput was at most 24%. The throughput of our background encryption was comparable to those of existing OS-based background encryption systems.

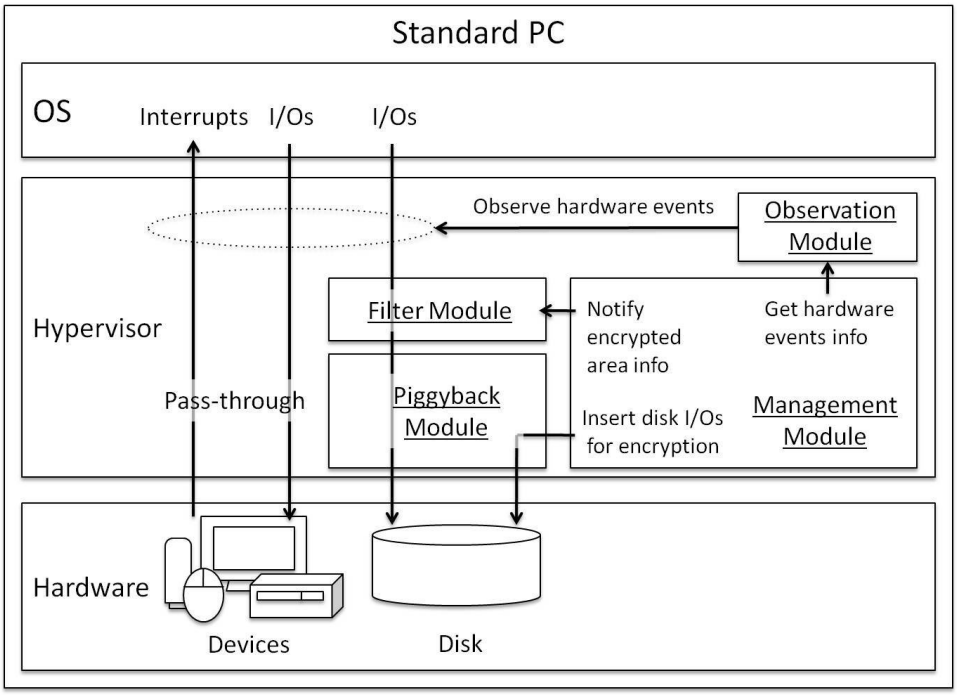


Figure 5.1: Encrypted/unencrypted areas

5.2 Design

Our VMM has four major components: *Observation Module* for guessing guest OS activities, *Filter Module* for encrypting/decrypting data transferred between the guest OS and disks, *Piggyback Module* for accessing unencrypted sectors from the VMM, and *Management Module* for managing the overall background encryption process (see Figure 5.1).

5.2.1 Concurrency control

Filter Module, shown in the middle of the figure, provides full disk encryption in a transparent manner from the guest OS. While background encryption is in progress, there exist two types of area in disks: 1) *encrypted areas* and 2) *unencrypted areas* (see Figure 5.2). Encrypted areas are areas that are already encrypted by background encryption, and unencrypted areas are areas that are not encrypted yet. Read access to encrypted areas is intercepted by the VMM and the data being transferred is decrypted before passed to the

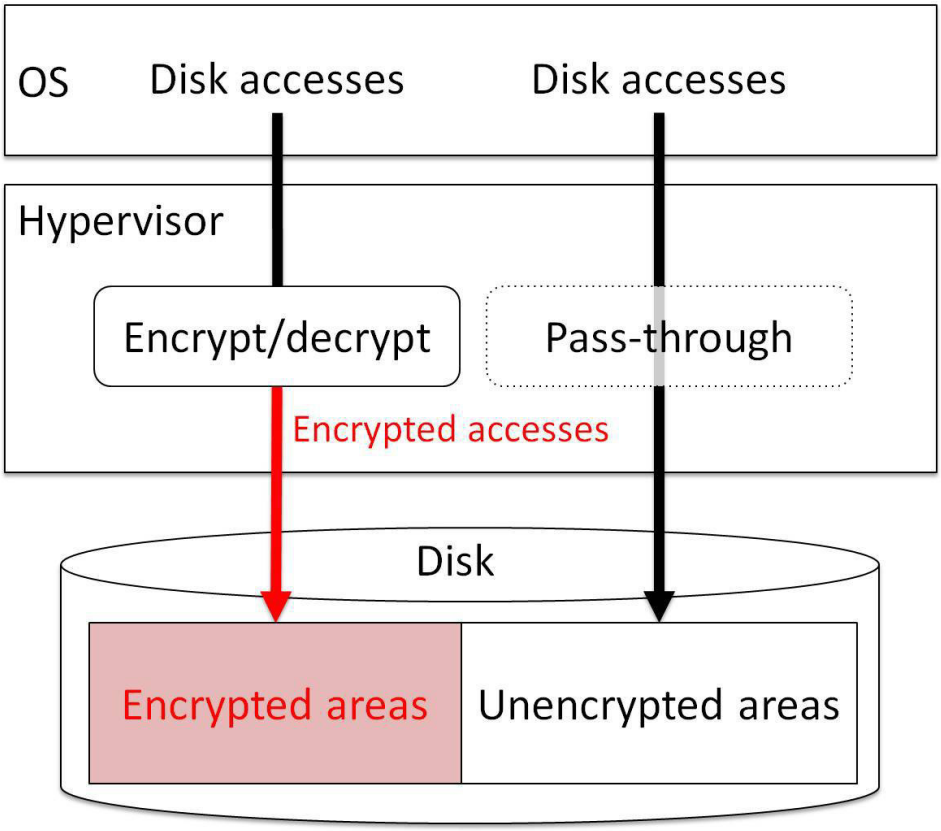


Figure 5.2: Encrypted/unencrypted areas

guest OS. In the same way, write access to encrypted areas is intercepted and the data being written is encrypted before passed to the disk devices. Access to unencrypted areas passes-through the VMM without modification. Filter Module intercepts disk access from the guest OS, determines that the accessed area is encrypted or not, and performs encryption/decryption if necessary.

5.2.2 Storage I/O mediation

To perform background encryption, the VMM needs to access disks in parallel with the guest OS. To access disks without modification of the guest OS, the VMM implements Piggyback Module that allows the VMM to insert its own disk access requests between the requests from the guest OS. With the technique described in Chapter 3, by conforming to

specifications of hardware device interfaces, the guest OS can continue to access disks in the same way as before the VMM is installed while the VMM can also access the same disks.

5.2.3 Background encryption

Management Module manages overall background encryption operations in cooperation with the other modules. It obtains hardware events via Observation Module and guesses the guest OS activities. Based on the guess, it controls the degree of background encryption speed and determines the appropriate timing of inserting I/O requests to perform background encryption. It inserts its own I/O requests for background encryption via Piggyback Module. It also manages the progress of the background encryption and notifies Filter Module of the information of encrypted and unencrypted areas.

The VMM must keep the progress information across system shutdown/reboot and suspend/resume events so that encrypted areas and unencrypted areas can be correctly distinguished. When the system is going to shutdown or suspend, Management Module saves progress information to the disk. When the system is going to boot or resume, Management Module loads this information and resumes background encryption from the appropriate point. The VMM pre-allocates the space for storing this information, as well as the space to store the VMM executable image. These areas are protected by the VMM and cannot be accessible from the guest OS. In addition, the VMM periodically saves this information to disks to tolerate accidental power loss.

If the VMM performs background encryption at full speed without considering the guest OS activities, it consumes too much CPU time and disk bandwidth, incurring significant overhead on the guest OS performance and annoying users. To avoid such a misery, the VMM moderate the degree of the background encryption speed based on the information provided by Observation Module.

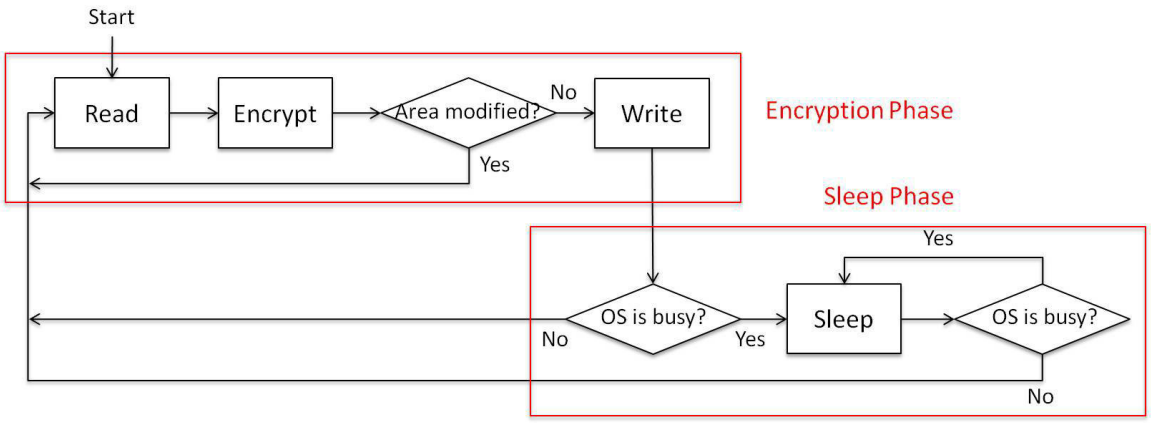


Figure 5.3: A basic unit of background encryption operation

5.2.4 Introspection for encryption speed moderation

Observation Module, shown in the upper-right side of the figure, performs introspection; it observes hardware events to obtain information necessary to guess guest OS activities. It intercepts several I/O instructions, such as keyboard and mouse I/Os, and external interrupts. It then records the information of these events and passes them to the hardware devices or the guest OS without modification. Therefore, these observation does not affect the operations of the guest OS. The frequency of these events are closely related with the guest OS activities and observing these events allows appropriate guess of them. Detailed strategies of guessing guest OS activities are described in the next section.

5.3 Implementation

5.3.1 Background encryption

To moderate the speed of background encryption, the VMM adaptively switches back and forth between two phases: *Encryption Phase* and *Sleep Phase*. Figure 5.3 shows the flow chart of each phase.

In Encryption Phase, the VMM repeats a cycle of three actions: 1) read a single unit of data from the disk, 2) encrypt the data, and 3) write the encrypted data back to the

same place on the disk. The size of a single unit is currently 2MB. At this time, we must be careful to avoid data corruption caused by intermixture of access from the guest OS and background encryption. If the guest OS writes data to the target area being encrypted after the area is read and before the area is written by the VMM (i.e. between the actions 1) and 3) in Encryption Phase), the VMM may overwrite the area with old data. One way to avoid this is to lock the target area and prevent the guest OS from accessing it while the VMM is encrypting. However, using locks may interfere with the guest OS operations. Therefore, we insert checks before writing encrypted data to make sure that the target area is not written by the guest OS after the VMM reads it. If written, the VMM retries the actions from the first.

In Sleep Phase, the VMM does not perform background encryption for a while. If Management Module determines that the guest OS is busy, it suspends the operation of background encryption and goes to Sleep Phase. If it determines that the guest OS is idle, it goes to Encryption Phase and resumes the operation of background encryption. Section 5.3.2 describes the criteria of this decision.

5.3.2 Encryption speed moderation

To moderate the speed of background encryption, Management Module makes two decisions: 1) when to go to Sleep Phase, and 2) how long it will stay in Sleep Phase. These decisions are made based on three parameters obtained from Observation Module: 1) Disk I/O frequency, 2) Keyboard/mouse I/O frequency, 3) External interrupt frequency. We chose threshold values of these parameters by conducting several experiments to measure the relationship between the parameters and guest OS activities. In this section, we first describe how to deal with theses three parameters. And finally, let us briefly mention how we deal with CPU usage of guest OSs for background encryption moderation.

Table 5.1: Relationship between user activities and keyboard/mouse I/O frequencies

User activity	I/O frequency (times/200msec)
Do nothing	0
Keyboard input	2-20
Mouse move	10-50

Disk I/O frequency

The performance of disk I/Os is one of the most important factors that decides the performance of entire systems because disk I/Os are often a bottleneck in today's computers. Therefore, we should avoid additional overhead on disk I/Os caused by background encryption as much as possible. Unfortunately, frequent access to disks from VMMs directly affects the performance of guest disk I/Os because they share the same physical disks whose maximum number of requests that can be handled at the same time is limited.

For the reasons described above, we choose a very strict threshold value: if at least one disk I/O occurs since the last cycle of background encryption, the VMM goes to Sleep Phase, and wait for 200ms. After waiting for 200ms, it checks whether a disk I/O is issued in the period. If issued, it stays in Sleep Phase and wait for another 200ms. If no disk I/O is issued, it goes back to Encryption Phase. The number of disk I/O is maintained in Observation Module by intercepting every disk access.

Keyboard/mouse I/O frequency

Response time of keyboard and mouse operations is directly related to user's experiences. Since encryption consumes much CPU time, frequent background encryption should be avoided if it affects keyboard or mouse operations.

To determine a proper threshold, we conducted experiments to measure the relationship between user activities and keyboard/mouse I/O frequencies. We intercepted access to port 0x60, a port used to get keyboard/mouse inputs, via Observation Module and calculates the number of I/Os issued in every 200msec. Table 5.1 shows the results. As shown in the

Table 5.2: Relationship between user activities and external interrupt frequencies

User activity	Int. freq. (times/200msec) [Ave.]
Do nothing	62-127 [82]
Keyboard/mouse use	78-296 [156]
File download	130-440 [286]
Application loading	73-323 [195]
Movie play	151-469 [215]

table, we can easily distinguish mouse I/Os from keyboard I/Os by watching the frequency of them. Therefore, we do not need to interpret the content of I/Os.

In our experiments, mouse movement easily becomes slower when frequent background encryption is in progress. On the other hand, keyboard inputs are not significantly affected by background encryption because the frequency of keyboard I/O is relatively lower. According to this observation, we choose threshold values of 20 and 10 times/200msec. If the frequency of I/O access to port 0x60 becomes higher than 20 times/200msec, the VMM goes to Sleep Phase and wait for 200ms. After waiting for 200ms, it checks the frequency again and determines whether it goes to Encryption Phase or not. If the frequency is between 20 and 10 times/200msec, the VMM goes to Sleep Phase and wait for 500ms. After that, it goes back to Encryption Phase.

External interrupt frequency

Although the frequencies of disk and keyboard/mouse I/O can be used to guess a part of user activities, they are not enough to guess various kinds of user activities. We use the frequency of external interrupts to guess such activities. For simplicity, we do not distinguish each source of external interrupts but calculate frequency of all external interrupts from all devices including network cards, disk controllers, video devices etc. Table 5.2 shows experimental results of measuring the relationship between several user activities and frequencies of external interrupts. As shown in the table, the frequency of external interrupts increases when user performs some activities. Therefore, at that time, the frequency of background

encryption should be reduced to avoid the performance impact on the user activities.

Based on the experimental results, we choose a threshold value of 100 times per 200msec. If the frequency becomes higher than this, the VMM goes to Sleep Phase and wait for 200ms. After that, it goes back to Encryption Phase. Note that the number of external interrupts occurred in every 200msec is recorded in Observation Module.

CPU usage

For simplicity, our system does not inspect the CPU usage of applications in guest OSs. In our experimental results with some modern PCs (including the environment shown in Table 5.3), the CPU usage of BitVisor with full speed background encryption is around 10-30% (the rest 70-90% are available for the guest OSs). Therefore, heavy CPU load applications in guest OSs without any low-level events will not be significantly affected by CPU time consumption of background encryption process. However, CPU usage of background encryption might become high depending on the machine specification such as CPU computation power, disk performance etc. For safety, our system allows to set maximum CPU usage of background encryption process. While the CPU usage exceeds the limitation, background encryption process stays in Sleep Phase.

5.3.3 Storage and key management

As described in Section 5.2, Management Module maintains progress information on the disks. This information must be stored before shutdown and sleep. We detect shutdown by intercepting INIT signal issued by guest OSs and detect sleep by intercepting ATA commands that make ATA host controller to enter sleep mode.

Our scheme needs a small free space on the disk for storing the VMM image and progress information. Currently the size of them is approximately 16MBytes, which can be stored at free spaces between disk partitions or bootable USB flash memories.

In our current implementation, encryption keys are stored in USB flash memories and

we use them as a physical key to boot the computer. Using TPM or other external tamper resistant devices can increase the security of encryption keys.

5.4 Experimental results

Table 5.3: Experimental environment A

CPU	Intel Core 2 Quad Q9550 2.83GHz
Memory	PC2-6400 4GB
Disk	Seagate Barracuda 7200.12 1TB
OS	Windows 7 Professional 32-bit

Table 5.4: Experimental environment B

CPU	Intel Core 2 Duo E8500 3.16Ghz
Memory	PC2-6400 2GB
Disk	Hitachi HDS721616PLA380 160GB
OS	Windows 7 Professional 32-bit

In this section, we show our experimental results. First, we show application benchmark results, and then show disk access throughput on a guest OS. Next, we show the throughput of background encryption of our system and compare it with those of existing commodity systems. Finally, we discuss initial deployment time of our system and existing VMM-based systems. We mainly used the environment shown in Table 5.3, although the comparison of background encryption throughput with existing commodity systems was performed in the environment shown in Table 5.4.

5.4.1 Application benchmark

To measure the impact of our background encryption on the guest OS performance, we ran an application benchmark on Windows 7 with and without our system, and compare the results. We used PCMark Vantage [39], a standard application benchmark tool using selected real applications. We ran the benchmark with three configurations: “Baremetal”

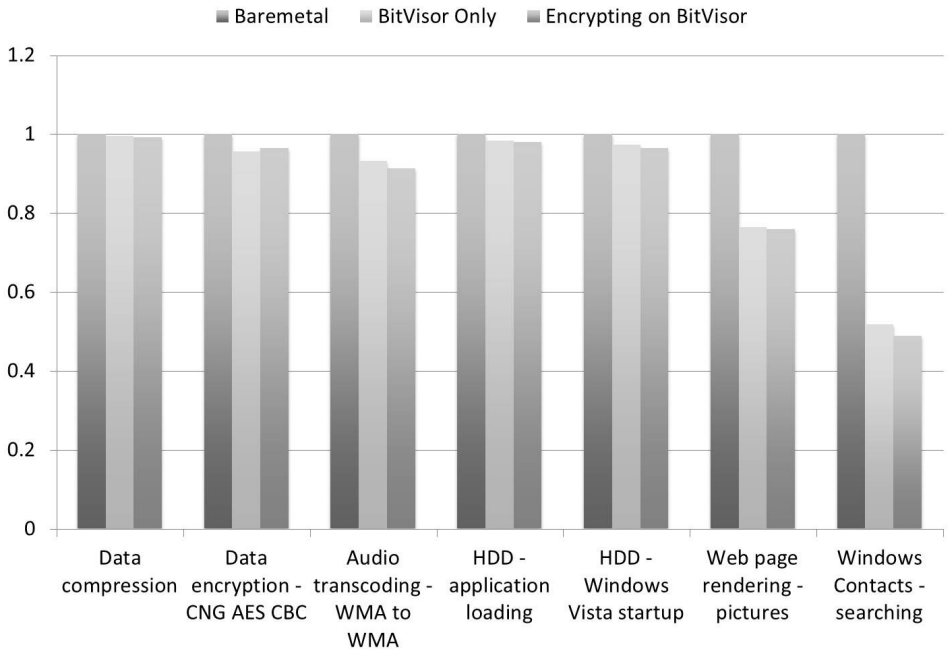


Figure 5.4: Application benchmark results

means the OS is directly running on the hardware, "BitVisor Only" means the OS is running on BitVisor without encryption, and "Encrypting on BitVisor" means the OS is running on our VMM with background encryption enabled.

Figure 5.4 shows the results. The graph shows the ratio to "Baremetal". In the figure, higher values indicate better performance. In "Encrypting on BitVisor", we used unencrypted disks and ran the benchmark while background encryption is in progress to measure the overhead caused by background encryption. The figure shows that our background encryption scheme does not significantly affect the application performance. Note that BitVisor itself incurs significant overhead on "Web page rendering-pictures" and "Windows contacts-searching" results. This is mainly caused by the implementation of shadow paging and will be mitigated by using hardware nested paging mechanisms.

5.4.2 Disk benchmark

To measure the impact on the guest disk performance, we measured disk access throughput with CrystalDiskMark [36]. Figure 5.5 shows the results. We added forth configuration: "Encrypted on BitVisor" means that the OS is running on our VMM but background encryption is completed. In sequential access with 1024KB records, read throughput is 137MB/sec and write is 134MB/sec in "Baremetal". In random access with 4KB records, read throughput is 0.65MB/sec and write is 0.54MB/sec. In most cases, the overhead is very low and negligible. Exception is the throughput of sequential access with 1024KB records: read overhead is 24% and write overhead is 15%. This is caused by cryptographic operations of XTS-AES performed by processors. In random access, this cost is hidden by relatively slow random access performance of disks.

5.4.3 Background encryption performance

We measured background encryption throughput of our system with four configurations: "Do nothing" means the system was left without touching it at all, "Application installation", "Web browsing", "Text editing" means we performed those tasks while measuring throughput. Table 5.5 shows the results. In "Do nothing", the throughput is approximately 1.3GB/min. When doing some tasks, it decreases to 0.3-0.6 GB/min. This means that the moderation of background encryption speed is effectively working.

Table 5.6 shows the throughput of background encryption in our system and existing commodity systems, measured in the environment B. We hide the actual product names for license reasons. All existing systems are OS-based systems, not VMM-based systems. The results show that the performance of our scheme is better than three of four existing systems, meaning that our VMM-based system can achieve comparable performance with OS-based systems.

Table 5.5: Background encryption throughput on our system

User activity	Throughput (GB/min)
Do nothing	1.3
Application installation	0.3
Web browsing	0.3
Text editing	0.6

Table 5.6: Comparison of background encryption throughput with commodity systems

Application	Throughput (GB/min)
Our system	1.02
Existing system A	2.35
Existing system B	0.72
Existing system C	0.93
Existing system D	0.67

5.4.4 Initial deployment cost

We compare our initial deployment cost of our system with that of KVM [51], a traditional VMM-based system, from the viewpoint of system installation time. We used the environment A and the size of the disk partition to be encrypted is 500GBytes.

The deployment time of our system into an existing installed system was less than 10 minutes, except for configuration steps such as selecting the place to install, keys, encryption algorithms, sectors to be encrypted. The installation itself was finished within a few seconds because a dedicated installer automatically stores a small boot loader and an executable core image (about 16MBytes totally) into the place specified at the time of configuration, such as a free space on disk devices, or bootable removable media like USB flash drives or CDs/DVDs. After rebooting once, which took one or two minutes, the user could start using the PC with background encryption enabled. If no OS was installed, about 15 minutes of OS installation (Windows 7) was added to the deployment process, which increased the total required time to 30 minutes.

On the other hand, the initial installation of a traditional VMM-based system of KVM required about at least 2 hours in this environment, except for configuration steps. Installing

a VMM with encryption mechanism took about 30 minutes, which consists of a host OS with KVM and dm-crypt [64] etc. Installing a 500GBytes OS image (Windows 7) onto KVM required 90 minutes, which is six times longer than when installing onto a physical disk without encryption. This is due to the overhead of virtualization and cryptographic computation.

To deploy full disk encryption into existing installed systems, this KVM-based system requires P2V operations and manual encryption of the disk images. According to our measurement, P2V operations required 5 hours, which was actually normal copy of a 500GBytes image here, and manual encryption for the image required 7 hours. Hence, the system required more than 12 hours for deployment before users could start using the encrypted PC environment.

5.5 Related work

5.5.1 OS-based full-disk encryption

In OS-based approaches, applications or kernel modules perform encryption operations. These approaches are well accepted in practical situations because of their simple and rapid deployment [83, 64, 49, 28, 62, 31, 89, 82, 30, 45]. In addition, some of them support background encryption with which users can start using PCs soon after the installation of encryption system without waiting for entire disk encryption to be completed [28, 62, 31, 89].

However, OS-based approaches are less secure than VMM-based approaches because most OS kernels in common use are monolithic and have security vulnerabilities [75, 40]. In addition, they are also strongly dependent on OSs or file systems. For example, BitLocker [28] is built into Windows, and cannot run on Linux/BSD. Some systems provide file-based encryption [82, 30, 45], limiting the choices of file systems.

Network booting systems [59, 74] may allow OS images to be transparently encrypted on server side. For example, a client can boot up an OS over iSCSI, transferring all disk

accesses to servers which encrypt the OS image with OS-based systems. This is also an OS-independent solution that can enforce security by isolating an encryption mechanism from a client OS itself. However, this approach requires initial deployment time to prepare OS images on servers. In addition, this system has additional overhead of network transfer of disk accesses compared to systems on local disks. This system also requires stable network connection and server computation power.

5.5.2 VMM-based full-disk encryption

Some client-side VMM-based systems [33, 58, 71, 51] support full encryption of guest OSs. However, most of these VMMs [33, 58, 51] require installation of large VMM images including host OSs after wiping-out disks. Some VMMs do not require disk wipe-out and can be injected into existing systems on local disks [71]. Unfortunately, to our best knowledge, none of them supports background encryption and require users to wait for the completion of manual encryption of entire OS images or installation with cryptographic computation. Our background encryption scheme could be applied to these types of VMMs. Hosted VMMs [23, 87] could support background encryption on application-based encryption systems with background encryption running on host OSs. However, to encrypt existing systems, it requires complicated deployment such as disk wipe-out, host OS installation, VMM installation as well as installation of encryption applications to host OSs.

Some central management products support encryption of guest OS images on server-side VMMs [85, 65]. However, background encryption is not supported in these systems (or is not mentioned in their data sheets). In these systems, installation cost of VMMs is not a big concern because they are installed to realize central management mechanism. However, when we encrypt guest OS images, background encryption has a possibility to reduce initial deployment cost of each encrypted guest OS by skipping manual encryption steps for existing OS images or avoiding OS installation with overhead of cryptographic computation. To encrypt existing OSs directly on physical disks, these systems still require

additional steps like P2V operations.

There are many approaches to introspect guest OS activities from VMM layer like our approach, sometimes referred as "out-of-the-box" approaches [20, 48, 68, 80]. Some of them introspect guest OSs deeply, watching kernel data structures of guest OSs which are sensitive for OS versions and patches [20, 48, 68]. This may raise initial deployment cost due to compatibility check or modification of guest OS images before installation. Our approach does not require such preparations by intercepting only OS-independent low-level events. Some approaches assume rich VMM functionality for introspection such as multiple guest OS support [80]. Our approach simplify VMMs to make VMM image itself small for easy deployment.

5.5.3 Hardware-based full-disk encryption

Hardware-based approaches can provide secure environment with little overhead [43, 55, 27]. However, additional cost to prepare dedicated hardware devices is required compared to software-based approaches. Hardware-based system is also difficult to be updated once deployed.

5.6 Summary

In this paper, we presented design and implementation of a scheme of background encryption based on the para pass-through VMM. Our scheme allows instant deployment of full disk encryption into existing systems without modification of existing OSs. To allow users to continue to use their PCs, the VMM performs background encryption that does not incur significant impact on the guest OS performance by carefully watching guest OS activities and moderating the degree of encryption speed. We showed a strategy to guess the guest OS activities by watching the frequency of I/O access and external interrupts. Based on the observation, the VMM suspends background encryption operations so that it does not affect the guest OS activities.

We used BitVisor as a base VMM and implemented four components to achieve efficient background encryption. Experimental results on Windows 7 showed that application benchmark scores were not significantly affected by the background encryption and the overhead on sequential disk access throughput was at most 24%. The throughput of our background encryption was comparable to that of existing OS-based background encryption systems and actual deployment time of our system was much shorter than an existing traditional VMM-based system.

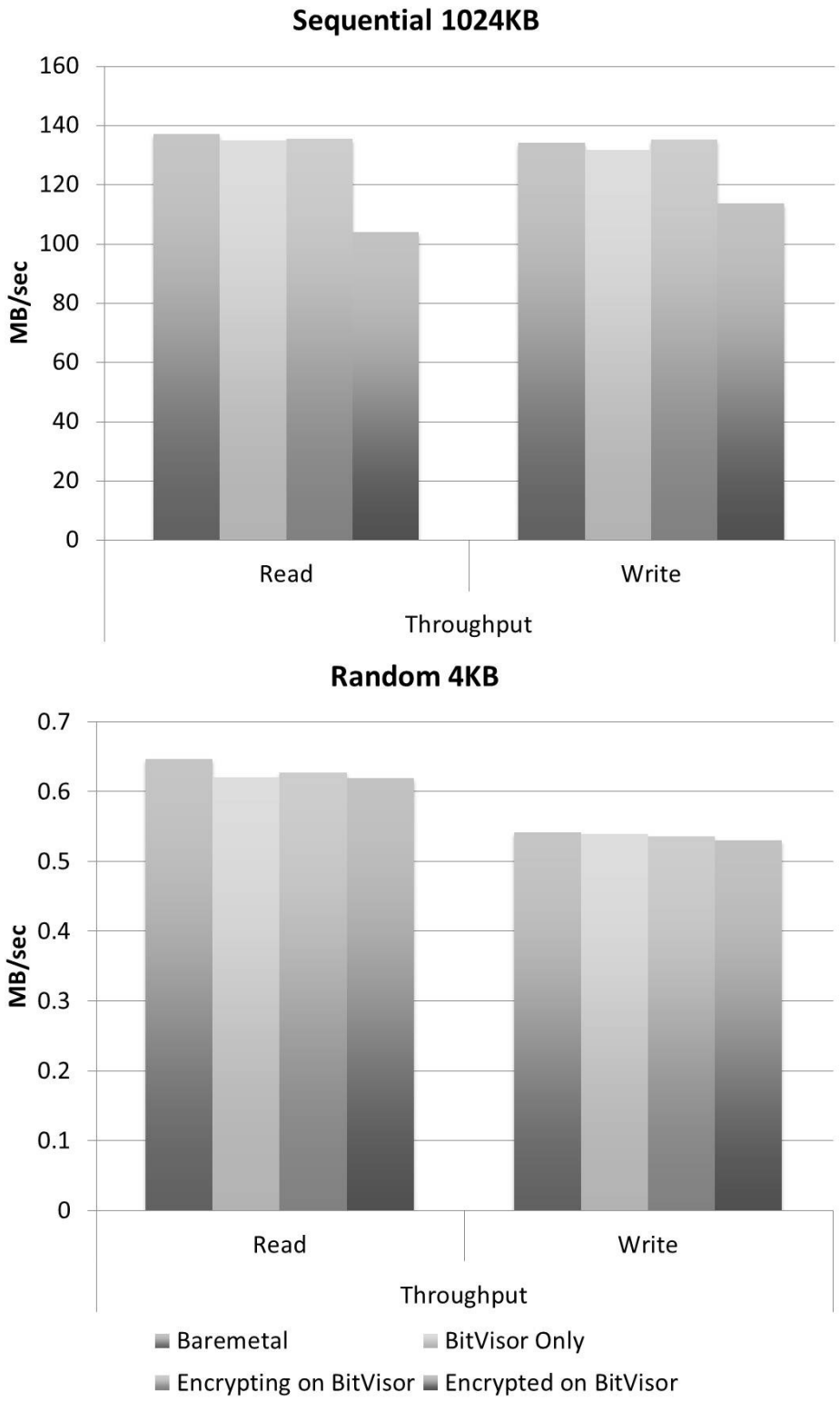


Figure 5.5: Disk access throughput

Chapter 6

Discussion and Future Work

In this chapter, we discuss the constrains of our frameworks and the approach of storage I/O mediation with the para pass-through VMM and future works.

Implementation cost of storage I/O mediators: Like device drivers for commodity OSs, storage I/O mediators also require device-specific knowledge for implementation. However, implementation of device mediators is simpler than that of device drivers because device mediators do not need to handle all I/Os but need to handle only key I/Os which are relevant to I/O redirection and multiplexing. For example, they can basically omit handling operations such as device initialization, power management, revision-specific trivial configuration and workarounds for errata.

Importance of OS-independency: Assuming that there are only limited types of OSs (e.g. Windows and Linux) are used in the systems, implementing device drivers for streaming deployment to them can be less costly.

However, there are always possibilities that users expect various types of OSs that include, but not limited to Windows and Linux. For example, customers may want to use research kernels, such as multikernels (e.g. Barrelfish), microkernels (e.g. L4), or library OSs (e.g. Exokernel), for the performance reason. Even if they use Linux or Windows, in-

stalling and maintaining customized drivers for each type, version and instance are daunting tasks for customers and may cause dependency problems. storage I/O mediators are, once implemented in the provider side, beneficial for all (many) customers by removing such tasks.

Cost of supporting various devices: This approach requires development of storage I/O mediators for each device and could raise cost of supporting various devices. However, we believe this issue is not critical because the variety of server hardware is limited compared to that of client and can be accommodated by software vendors. For example, VMWare ESXi maintains their own set of device drivers for various server hardware. Moreover, storage I/O mediators can be implemented by less development efforts as we mentioned above.

Given formal device specifications, automatic synthesis of storage I/O mediators is theoretically possible and is a future work for covering various devices with minimal development cost. From a high-level viewpoint, storage I/O mediators have similar logic with device drivers in the sense that they perform device-specific operations responding to requests from guest OSs. Therefore, existing techniques for automatic device driver synthesis [72] can be applicable to the development of storage I/O mediators. Furthermore, storage I/O mediators are independent from OSs and are expected to be synthesized with more simplified method.

Chapter 7

Conclusion

VMMs are powerful frameworks that greatly simplify management tasks of computers today, allowing flexible control on OS execution and images (e.g. remapping, caching, deployment, encryption and etc.). Unfortunately, VMMs rely on heavy virtualization for providing these abilities, sacrificing performance and device functionality.

We presented device mediators that allow para pass-through VMMs to control OS images without relying on virtualization. Instead of emulating devices, device mediators perform polling-based device-interface-level I/O mediation by carefully monitoring, intercepting, manipulating and inserting I/O requests in a manner conforming to device specifications. We have achieved bare-metal benefits but still demonstrate flexible management operations of OS images in the VMM layer: network booting, streaming deployment, and background encryption of OS images. Conventionally, the two factors, virtualization and manageability of OS images, were tightly coupled in the world of VMM-based systems management. We have decoupled these two factors, inventing novel I/O controlling technique and paved a new path to VMM-based systems management.

Acknowledgements

I am truly indebted and thankful to Professor, Kazuhiko Kato, for his encouragement and support. His wonderful lecture I heard when I was an undergraduate student led me to this exciting field of computer science. I owe sincere and earnest thankfulness to Associate Professor, Takahiro Shinagawa, for giving me a lot of practical and helpful advices for my research and opportunities to improve my skills of researchs and implementation. This researh and thesis would not have been possible without his advices.

I would also like to thank Hideki Eiraku, Tomohiro Kitamura, and Katsuya Matsubara, for giving me many informative and important advices about the technical details of BitVisor and helping me the implementation and evaluation of our systems, improving the fundamental implementation of BitVisor. Their helps are essential for this work.

I would also like to show my gratitude to Assistant Professor, Hirotake Abe, Akiyoshi Sugiki and Koji Hasebe for giving me some important advices. I would like to thank Atsuko Kakizawa, for her kindness and encouragement. I would also like to thank Takaaki Fukai, Takekoshi Satoru for discussing many topics in computer systems, and thank all other members of Operating Systems and System Software Laboratory.

Last but never the least, I am really grateful to my family and my fiancee, Kumiko Shinokura, for their love, support and encouragement.

Bibliography

[1] ATA over Ethernet Tools. http://aoetools.sourceforge.net/.

[2] AT&T Cloud Architect. http://cloudarchitect.att.com/.

[3] baremetalcloud. http://baremetalcloud.com/.

[4] Containers at scale. https://speakerdeck.com/jbeda/containers-at-scale.

[5] Facebook: Virtualisation does not scale. http://www.zdnet.com/facebook-virtualisation-does-not-scale-4010021998/.

[6] Internap Network Services Corporation. http://www.internap.com/.

[7] ioping. https://code.google.com/p/ioping.

[8] Liquid Web Inc. http://www.liquidweb.com/.

[9] Openstack. http://www.openstack.org/.

[10] OSU Micro-Benchmarks. http://mvapich.cse.ohio-state.edu/benchmarks/.

[11] Peer 1 Hosting. http://www.peer1.com/.

[12] Rackspace, Inc. http://www.rackspace.com/.

[13] SoftLayer Technologies, Inc. http://www.softlayer.com/.

[14] VM streaming and VMware Workstation 6.5. https://blogs.vmware.com/vmtn/2008/09/vm-streaming-an.html.

[15] Webair Internet Development, Inc. http://www.webair.com/.

[16] Five considerations for building online gaming infrastructure. Technical report, November 2003.

[17] Advanced Micro Devices, Inc. *AMD64 Architecture Programmers Manual Revision 3.22*, September 2012.

[18] Mohit Aron and Peter Druschel. Soft Timers: Efficient Microsecond Software Timer Support for Network Processing. *ACM Transactions on Computer Systems (TOCS)*, 18(3):197–228, August 2000.

[19] Jens Axboe. fio – Flexible IO Tester. http://git.kernel.dk/?p=fio.git.

[20] Ahmed M. Azab, Peng Ning, Emre C. Sezer, and Xiaolan Zhang. Hima: A hypervisor-based integrity measurement agent. In *Proceedings of the 2009 Annual Computer Security Applications Conference*, ACSAC '09, pages 461–470, 2009.

[21] 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 nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 164–177, 2003.

[22] Sean Kenneth Barker and Prashant Shenoy. Empirical Evaluation of Latency-sensitive Application Performance in the Cloud. In *Proceedings of the 1st annual ACM SIGMM Conference on Multimedia Systems*, MMSys '10, pages 35–46, 2010.

[23] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, 2005.

[24] 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 Conference on Operating Systems Design and Implementation*, OSDI'10, pages 423–436, 2010.

[25] Muli Ben-Yehuda, Jimi Xenidis, Michal Ostrowski, Karl Rister, Alexis Bruemmer, and Leendert van Doorn. The price of safety: Evaluating iommu performance. In *OLS '07: The 2007 Ottawa Linux Symposium*, pages 9–20, July 2007.

[26] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. Accelerating Two-Dimensional Page Walks for Virtualized Systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 26–35, 2008.

[27] Bill Bose. FDE Performance Comparison Hardware Versus Software Full Drive Encryption, February 2010.

[28] BitLocker Drive Encryption. http://windows.microsoft.com/en-US/windows7/products/features/bitlocker.

[29] Robert Bradford, Evangelos Kotsovinos, Anja Feldmann, and Harald Schiöberg. Live wide-area migration of virtual machines including local persistent state. In *Proceedings of the 3rd international conference on Virtual execution environments*, VEE '07, pages 169–179, 2007.

[30] Giuseppe Cattaneo, Luigi Catuogno, Aniello Del Sorbo, and Pino Persiano. The design and implementation of a transparent cryptographic file system for unix. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 199–212, Berkeley, CA, USA, 2001. USENIX Association.

[31] CE-Infosys. Compusec release 1.1, January 2003.

[32] Ramesh Chandra, Nickolai Zeldovich, Constantine Sapuntzakis, and Monica S. Lam. The Collective: A Cache-Based System Management Architecture. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI)*, pages 259–272, 2005.

[33] Citrix XenClient. http://www.citrix.com/xenclient.

[34] Patrick Colp, Mihir Nanavati, Jun Zhu, William Aiello, George Coker, Tim Deegan, Peter Loscocco, and Andrew Warfield. Breaking up is hard to do: security and functionality in a commodity hypervisor. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 189–202, 2011.

[35] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud Computing*, SoCC '10, pages 143–154, 2010.

[36] Crystal Dew World. http://crystalmark.info/.

[37] Clerc David, Garces-Erice Luis, and Rooney Sean. OS Streaming Deployment. In *Proceedings of the 2010 IEEE 29th International Performance Computing and Communications Conference (IPCCC)*, pages 169 –179, December 2010.

[38] Le Duy, Huang2 Hai, and Wang Haining. Understanding performance implications of nested file systems in a virtualized environment. 2012.

[39] Futuremark Corporation. PCMark Vantage Whitepaper v1.0, November 2007.

[40] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: a virtual machine-based platform for trusted computing. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 193–206, 2003.

[41] Abel Gordon. ELVIS/ELI I/O acceleration code. `http://lists.gnu.org/archive/html/qemu-devel/2013-09/msg04610.html`.

[42] Abel Gordon, Nadav Amit, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir. ELI: Bare-metal Performance for I/O Virtualization. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 411–422, 2012.

[43] Laszlo Hars. Discryption: Internal Hard-Disk Encryption for Secure Storage. *Computer*, 40:103–105, 2007.

[44] Hewlett-Packard Corporation, Intel Corporation, Microsoft Corporation, Phoenix Technologies Ltd. and Toshiba Corporation. *Advanced Configuration and Power Interface Specification Revision 5.0*, December 2011.

[45] James P. Hughes and Christopher J. Feist. Architecture of the secure file system, 2001.

[46] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developers Manual, 253665-045US*, January 2013.

[47] Alexandru Iosup, Simon Ostermann, Nezih Yigitbasi, Radu Prodan, Thomas Fahringer, and Dick Epema. Performance Analysis of Cloud Computing Services for Many-Tasks Scientific Computing. *IEEE Transactions on Parallel and Distributed Systems*, 22(6):931–945, June 2011.

[48] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. Stealthy malware detection and monitoring through vmm-based "gout-of-the-box" semantic view reconstruction. *ACM Trans. Inf. Syst. Secur.*, 13:12:1–12:28, March 2010.

[49] P. H. Kamp. Gbde-geom based disk encryption. *BSDCon 2003*, 2003.

[50] Eric Keller, Jakub Szefer, Jennifer Rexford, and Ruby B. Lee. NoHype: Virtualized Cloud Infrastructure without the Virtualization. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 350–361, 2010.

[51] Avi Kivity. kvm: the Linux Virtual Machine Monitor. In *Proceedings of the 2007 Ottawa Linux Symposium*, pages 225–230, July 2007.

[52] Jinzhu Kong. A practical approach to improve the data privacy of virtual machines. In *Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology*, CIT '10, pages 936–941, 2010.

[53] Thawan Kooburat and Michael Swift. The Best of Both Worlds with On-Demand Virtualization. In *Proceedings of the 13th Workshop on Hot Topics in Operating Systems*, HotOS XIII, 2011.

[54] Alexey Kopytov. Sysbench. `https://launchpad.net/sysbench`.

[55] Cameron Laird. Taking a Hard-Line Approach to Encryption. *Computer*, 40:13–15, 2007.

[56] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, April 2010.

[57] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 2–2, 2004.

[58] Min Liang and Chao wen Chang. Research and design of full disk encryption based on virtual machine. In *Computer Science and Information Technology (ICCSIT), 2010 3rd IEEE International Conference*, pages 642–646, July 2010.

[59] Books LLC. *Network Booting: Preboot Execution Environment, Bootstrap Protocol, Netboot, Gpxe, Remote Initial Program Load.* Books Nippan, 2010.

[60] 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, ASPLOS XI, pages 211–223, 2004.

[61] Ali Mashtizadeh, Emré Celebi, Tal Garfinkel, and Min Cai. The design and evolution of live storage migration in VMware ESX. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, USENIXATC'11, pages 14–14, 2011.

[62] McAfee. Mcafee endpoint encryption data sheet, 2011.

[63] Dutch T. Meyer, Brendan Cully, Jake Wires, Norman C. Hutchinson, and Andrew Warfield. Block mason. In *Proceedings of the First conference on I/O virtualization*, WIOV'08, pages 4–4, 2008.

[64] Mike Petullo. Encrypt your root filesystem. *Linux Journal*, December 2004.

[65] Moka5. http://www.moka5.com/.

[66] Vlad Nae, Radu Prodan, Thomas Fahringer, and Alexandru Iosup. The Impact of Virtualization on the Performance of Massively Multiplayer Online Games. In *Proceedings of the 8th Annual Workshop on Network and Systems Support for Games*, NetGames '09, pages 9:1–9:6, 2009.

[67] Diego Ongaro, Alan L. Cox, and Scott Rixner. Scheduling i/o in virtual machine monitors. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '08, pages 1–10, 2008.

[68] Bryan D. Payne, Martim Carbone, Monirul Sharif, and Wenke Lee. Lares: An architecture for secure active monitoring using virtualization. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, pages 233–247, 2008.

[69] Padala Pradeep, Zhu Xiaoyun, Wang Zhikui, Singhal Sharad, and Kang G. Shin. Performance evaluation of virtualization technologies for server consolidation. Technical report, 2007.

[70] Himanshu Raj and Karsten Schwan. High Performance and Scalable I/O Virtualization via Self-Virtualized Devices. In *Proceedings of the 16th International Symposium on High Performance Distributed Computing*, HPDC '07, pages 179–188, 2007.

[71] M. Rezaei, N.S. Moosavi, H. Nemati, and R. Azmi. Tcvisor: A hypervisor level secure storage. In *Internet Technology and Secured Transactions (ICITST), 2010 International Conference*, pages 1–9, December 2010.

[72] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, Etienne Le Sueur, and Gernot Heiser. Automatic device driver synthesis with termite. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 73–86, New York, NY, USA, 2009. ACM.

[73] B. Coile S. Hopkins. AoE (ATA over Ethernet). February 2009.

[74] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, and E. Zeidner. Internet small computer systems interface (iscsi), 2004.

[75] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, SOSP '07, pages 335–350, 2007.

[76] Jeffrey Shafer. I/O Virtualization Bottlenecks in Cloud Computing Today. In *Proceedings of the Second Workshop on I/O Virtualization*, WIOV'10, 2010.

[77] Takahiro Shinagawa, Hideki Eiraku, Kouichi Tanimoto, Kazumasa Omote, Shoichi Hasegawa, Takashi Horie, Manabu Hirano, Kenichi Kourai, Yoshihiro Oyama, Eiji Kawai, Kenji Kono, Shigeru Chiba, Yasushi Shinjo, and Kazuhiko Kato. BitVisor: A Thin Hypervisor for Enforcing I/O Device Security. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '09, pages 121–130, 2009.

[78] Takahiro Shinagawa, Hideki Eiraku, Kouichi Tanimoto, Kazumasa Omote, Shoichi Hasegawa, Takashi Horie, Manabu Hirano, Kenichi Kourai, Yoshihiro Oyama, Eiji Kawai, Kenji Kono, Shigeru Chiba, Yasushi Shinjo, and Kazuhiko Kato. Bitvisor: a thin hypervisor for enforcing i/o device security. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, pages 121–130, 2009.

[79] Ostermann Simon, Iosup Ru, Yigitbasi Nezih, and Fahringer Thomas. A Performance Analysis of EC2 Cloud Computing Services for Scientific Computing. In *Proceedings of the ICST International Conference on Cloud Computing*, pages 115–131, 2009.

[80] Deepa Srinivasan, Zhi Wang, Xuxian Jiang, and Dongyan Xu. Process out-grafting: an efficient "out-of-vm" approach for fine-grained process execution monitoring. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 363–374, New York, NY, USA, 2011. ACM.

[81] Jakub Szefer, Eric Keller, Ruby B. Lee, and Jennifer Rexford. Eliminating the hypervisor attack surface for a more secure cloud. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 401–412, 2011.

[82] The Encrypting File System. http://technet.microsoft.com/en-us/library/cc700811.aspx.

[83] TrueCrypt. http://www.truecrypt.org/.

[84] Volkmar Uhlig, Joshua LeVasseur, Espen Skoglund, and Uwe Dannowski. Towards scalable multiprocessor virtual machines. In *Proceedings of the 3rd Virtual Machine Research And Technology Symposium*, pages 43 – 56. USENIX Association, May 2004.

[85] VMware ACE. http://www.vmware.com/products/ace/.

[86] Guohui Wang and T. S. Eugene Ng. The Impact of Virtualization on Network Performance of Amazon EC2 Data Center. In *Proceedings of the 29th IEEE International Conference on Computer Communications*, INFOCOM 2010, pages 1163–1171, 2010.

[87] Jon Watson. Virtualbox: bits and bytes masquerading as machines. *Linux J.*, 2008, February 2008.

[88] Paul Willmann, Jeffrey Shafer, David Carr, Aravind Menon, Scott Rixner, Alan L. Cox, and Willy Zwaenepoel. Concurrent Direct Network Access for Virtual Machine Monitors. In *Proceedings of the IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 306–317, 2007.

[89] WinMagic. Securedoctm version 5.2 technical specifications, 2011.

[90] Lei Xia, Sanjay Kumar, Xue Yang, Praveen Gopalakrishnan, York Liu, Sebastian Schoenberg, and Xingang Guo. Virtual wifi: bring virtualization from wired to wireless. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '11, pages 181–192, 2011.

[91] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, pages 203–216, 2011.