Virtualizing Resources for the Cloud

Mohammad Hammoud and Majd F. Sakr

July 13, 2013

Contents

Chapter 1

Virtualizing Resources for the Cloud

Virtualization is at the core of cloud computing. It lies on top of the cloud infrastructure, whereby virtual resources (e.g., virtual CPUs, memories, disks and networks) are constructed from the underlying physical resources and act as proxies to them. As is the case with the idea of cloud computing, which was first introduced in the 1960s [1], virtualization can be traced back to the 1970s [55]. Forty years ago, the mainframe computer systems were extremely large and expensive. To address expanding user needs and costly machine ownerships, the IBM 370 architecture, announced in 1970, offered complete virtual machines (virtual hardware images) to different programs running at the same computer hardware. Over time, computer hardware became less expensive and users started migrating to low-priced desktop machines. This drove the adoption of the virtualization technology to fade for a while. Today, virtualization is enjoying a resurgence in popularity with a number of research projects and commercial systems providing virtualization solutions for commodity PCs, servers, and the cloud.

In this chapter, we present various ingredients of the virtualization technology and the crucial role it plays in enabling the cloud computing paradigm. First, we identify major reasons for why virtualization is becoming important, especially for the cloud. Second, we indicate how multiple software images can run side-by-side on physical resources while attaining security, resource and failure isolations. Prior to delving into more details about virtualization, we present a brief background requisite for understanding how physical resources can be virtualized. In particular, we learn how system complexity can be managed in terms of levels of abstractions and well-defined interfaces. To this end, we formally define virtualization and examine two main virtual machine types, process and system virtual machines.

After introducing and motivating virtualization for the cloud, we describe in detail CPU, memory, and I/O virtualizations. Specifically, we first explore conditions for virtualizing CPUs, identify the difference between full virtualization and paravirtualization, explain emulation as a major technique for CPU virtualization, and examine virtual CPU scheduling in Xen, a popular hypervisor utilized by Amazon at its Amazon Web Services (AWS) cloud computing platform. Second, we outline the difference between conventional Operating System's virtual memory and system memory virtualization, explain the multiple levels of page mapping as dictated by system memory virtualization, define memory over-commitment and illustrate memory ballooning, a reclamation technique to tackle memory over-commitment in VMware ESX, another common hypervisor. Third, we explain how CPU and I/O devices can communicate with and without virtualization, identify the three main interfaces, system call, device driver and operation level interfaces, at which I/O virtualization can be ensued. As a practical example, we discuss Xen's approach to I/O virtualization. Finally, we close with a case study on Amazon Elastic Compute Cloud (Amazon EC2), which applies system virtualization to provide Infrastructure as a Service (IaaS) on the cloud. We investigate the underlying virtualization technology of Amazon EC2 and some of its major characteristics such as elasticity, scalability, performance, flexibility, fault tolerance and security.

1.1 Why Virtualization?

Virtualization is predominantly used by programmers to ease software development and testing, by IT datacenters to consolidate dedicated servers into more cost effective hardware, and by the cloud (e.g., Amazon EC2) to isolate users sharing a single hardware layer and offer elasticity, among others. Next, we discuss seven areas that virtualization enables on the cloud.

1.1.1 Enabling the Cloud Computing System Model

A major use case of virtualization is cloud computing. Cloud computing adopts a model whereby software, computation, and storage are offered as services. These services range from arbitrary applications (called Software as a Service or SaaS) like Google Apps [25], through platforms (denoted as Platform as a Service or PaaS) such as Google App Engine [26], to physical infrastructures (referred to as Infrastructure as a Service or IaaS) such as Amazon EC2. For example, IaaS allows cloud users to provision virtual machines (VMs) for their own use. As shown in Fig. 1.1, provisioning a VM entails obtaining virtual versions of every physical machine component, including CPU, memory, I/O and storage. Virtualization makes this possible via a virtualization intermediary known as the **hypervisor** or the **virtual machine monitor** (VMM). Examples of leading hypervisors are Xen [9, 47] and VMware ESX [59]. Amazon EC2 uses Xen for provisioning user VMs.

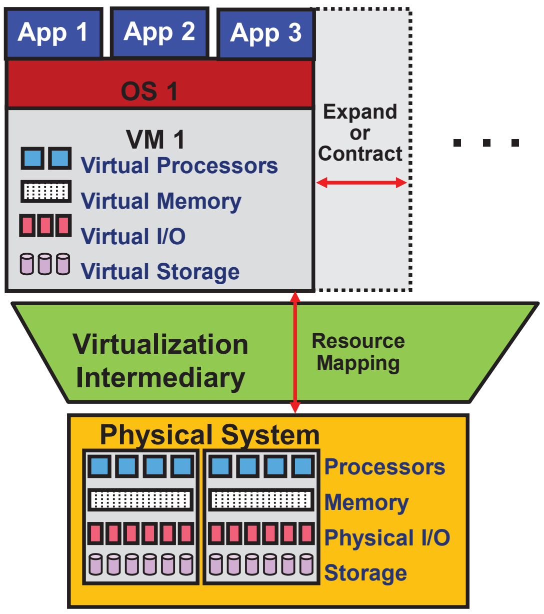


Figure 1.1: Provisioning a Virtual Machine (VM) on a physical system.

1.1.2 Elasticity

A major property of the cloud is *elasticity* or the ability to respond quickly to user demands by including or excluding resources for SaaS, PaaS, and/or IaaS, either manually or automatically. As shown in Fig. 1.1, virtualization enhances elasticity by allowing providers/users to expand or contract services on the cloud. For instance, Google App Engine automatically expands servers during demand spikes, and contracts them during demand lulls. On the other hand, Amazon EC2 allows users to expand and contract their own virtual clusters either manually (by default) or automatically (by using Amazon Auto Scaling [2]). In short, virtualization is a key technology for attaining elasticity on the cloud.

1.1.3 Resource Sandboxing

A system VM provides a sandbox that can isolate one environment from others, ensuring a level of security that may not be applicable with conventional Operating Systems (OSs). First, a user running an application on a private machine might be reluctant to move her/his applications to the cloud; unless guarantees are provided that her/his applications and activities cannot be accessed and monitored by any other user on the cloud. Virtualization can greatly serve in offering a safe environment for every user, through which, it is not possible for one user to observe or alter another's data and/or activity. Second, as the cloud can also execute user applications concurrently, a software failure of one application cannot generally propagate to others, if all are running on different VMs. Such a property is usually referred to as *fault containment*. Clearly, this increases the robustness of the system. In a non-virtualized environment, however, erratic behavior of one application can bring down the whole system.

Sandboxing as provided by virtualization opens up interesting possibilities as well. As illus-

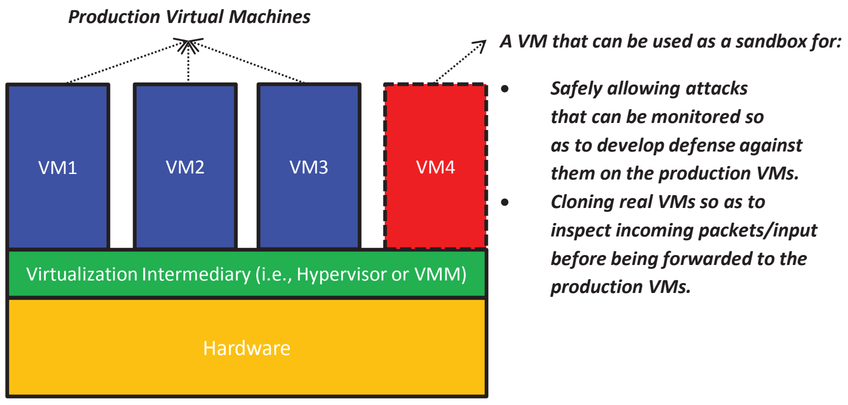

Production Virtual Machines
A VM that can be used as a sandbox for:
• Safely allowing attacks that can be monitored so as to develop defense against them on the production VMs.
• Cloning real VMs so as to inspect incoming packets/input before being forwarded to the production VMs.
VM1
VM2
VM3
VM4
Virtualization Intermediary (i.e., Hypervisor or VMM)
Hardware

Figure 1.2: Using Virtual Sandboxes to develop defenses against attacks and to monitor incoming data.

trated in Fig. 1.2, a specific VM can be used as a sandbox whereby security attacks (e.g., denial-of-service attacks or inserting a malicious packet into a legitimate IP communication stream) can be safely permitted and monitored. This can allow inspecting the effects of such attacks, gathering information on their specific behaviors, and replaying them if necessary so as to design a defense against future attacks (by learning how to detect and quarantine them before they can cause any harm). Furthermore, suspicious network packets or input can be sent to a clone (a specific VM) before it is forwarded to the intended VM so as to preclude any potential ill effect. A VM can be thrown away after it has served its purpose.

1.1.4 Improved System Utilization and Reduced Costs and Energy Consumption

It was observed very early that computer hardware resources are typically underutilized. The concept of resource sharing has been successfully applied in multiprogramming OSs to improve system utilization. Resource sharing in multiprogramming OSs, however, provides only process abstractions (not systems) that can access resources in parallel. Virtualization takes this step forward by creating an illusion of complete systems, whereby multiple VMs can be supported simultaneously, each running its own system image (e.g., OS) and associated applications. For instance, in virtualized datacenters, 7 or more VMs can be provisioned on a single server, providing potentially resource utilization rates of 60~80% [8]. In contrast, only 5~10% resource utilization rates are accomplished in non-virtualized datacenters [8]. By enabling multiple VMs on a single physical server, virtualization allows consolidating physical servers into virtual servers that run on many fewer physical servers (a concept referred to as *server consolidation*). Clearly, this can lead not only to improved system utilization but also to reduced costs.

Server consolidation as provided by virtualization leads not only to improved system utilization and reduced costs but further to optimized energy consumption in cloud datacenters. Datacenters hosting cloud applications consume tremendous amounts of energy, resulting in high operational costs and carbon dioxide emissions [11]. Server consolidation is perceived as an effective way to improve the energy efficiency of datacenters via consolidating applications running on multiple physical servers into fewer virtual servers. Idle physical servers can subsequently be switched off so as to decrease energy consumption. Studies show that server consolidation can save up to 20% of datacenter energy consumption [31, 53]. A large body of research work illustrates the promise of virtualization in reducing energy consumption in cloud datacenters (e.g., [11–13, 31, 33, 57]). Indeed, mitigating the explosive energy consumption of cloud datacenters is currently deemed as one of the key challenges in cloud computing.

1.1.5 Facilitating Big Data Analytics

The rapidly expanding Information and Communications Technology (ICT) that is permeating all aspects of modern life has led to a massive explosion of data over the last few decades. Major advances in connectivity and digitization of information have led to the creation of ever increasing volumes of data on a daily basis. This data is also diverse, ranging from images and videos (e.g., from mobile phones being uploaded to websites such as Facebook and YouTube), to 24/7 digital TV broadcasts and surveillance footages (e.g., from hundreds of thousands of security cameras), to large scientific experiments (e.g., the Large Hadron Collider [35]), which produce many terabytes of data every single day. IDC's latest Digital Universe Study predicts a 300-fold increase in the volume of data globally, from 130 exabytes in 2012 to 30,000 exabytes in 2020 [23].

Organizations are trying to leverage or, in fact, cope with the vast and diverse volumes of data (or *Big Data*) that is seemingly growing ever so fast. For instance, Google, Yahoo! and Facebook have gone from processing gigabytes and terabytes of data to the petabyte range [37]. This puts immense pressure on their computing and storage infrastructures which need to be available 24/7 and scale seamlessly as the amount of data produced rises exponentially. Virtualization provides a reliable and elastic environment, which ensures that computing and storage infrastructures can effectively tolerate faults, achieve higher availability and scale as needed to handle large volumes and varied types of data; especially when the extent of data volumes are not known a-priori. This allows efficient reaction to unanticipated demands of Big Data analytics, better enforcement of reliable Quality of Service (QoS) and satisfactory meeting of Service Level Agreements (SLAs). Besides, virtualization facilitates the manageability and portability of Big Data platforms by abstracting data from its underpinnings and removing the dependency from the underlying physical hardware. Lastly, virtualization provides the foundation that enables many of the cloud services to be used either as Big Data analytics or as data sources in Big Data analytics. For instance, Amazon Web Services added a new service, Amazon Elastic MapReduce (EMR), to enable businesses, researchers, data analysts, and developers to easily and cost effectively process Big Data [4]. EMR exploits virtualization and uses Hadoop [28] as an underlying

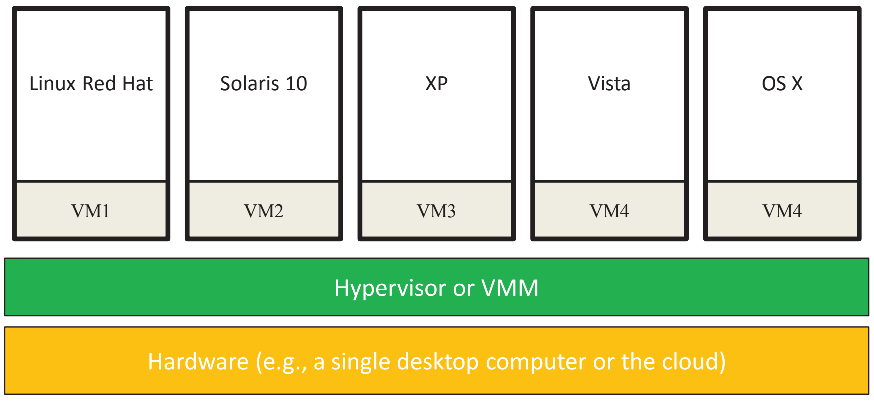


Figure 1.3: Mixed-OS environment offered by system virtualization.

framework hosted on Amazon EC2 and Amazon Simple Storage Service (Amazon S3) [5]. The state-of-the-art Hadoop MapReduce can also be used with Amazon S3, exactly as is the case with Hadoop Distributed File System (HDFS) [24, 28].

1.1.6 Mixed-OS Environment

As shown in Fig. 1.3 and pointed out in Section 1.1.4, a single hardware platform can support multiple OSs simultaneously. This provides great flexibility for users where they can install their own OSs, libraries and applications. For instance, a user can install one OS for office productivity tools and another OS for application development and testing, all on a single desktop computer or on the cloud (e.g., on Amazon EC2).

1.1.7 Facilitating Research

Running an OS on a VM allows the hypervisor to instrument accesses to hardware resources and count specific event types (e.g., page faults) or even log detailed information about events' natures, events' origins, and how operations are satisfied. Moreover, traces of executions and dumps of machine states at points of interests can be taken at the VM level, an action that cannot be performed on native systems. Lastly, system execution can be replayed on VMs from some saved state for analyzing system behavior under various scenarios. Indeed, the complete state of a VM can be saved, cloned, encrypted, moved, and/or restored again, actions that are not so easy to do with physical machines [15]. As such, it has become quite common for OS researchers to conduct most of their experiments using VMs rather than native hardware platforms [55].

1.2 Limitations of General-Purpose Operating Systems

The Operating System (OS) binds all hardware resources to a single entity which is the OS. This limits the flexibility of the system, not only in terms of applications that can run concurrently and share resources, but also in terms of isolation. Isolation is crucial in cloud computing with many users sharing the cloud infrastructure. A system is said to provide full isolation when it supports a combination of *fault isolation*, *resource isolation*, and *security isolation* [56]. Fault isolation reflects the ability to limit a buggy program from affecting another program. Complete fault isolation requires no sharing of code or data. Resource isolation corresponds to the ability of enforcing/controlling resource usages of programs. This requires careful allocation and scheduling of resources. Lastly, security isolation refers to the extent at which accesses to logical objects or information (e.g., files, memory addresses, and port numbers) are limited. Security isolation promotes safety where one application cannot reveal information (e.g., names of files or process IDs) to any other application. General-purpose OSs provide a weak form of isolation (only the process abstraction) and not full isolation.

On the other hand, virtualization relaxes physical constraints and enables optimized system flexibility and isolation. Hypervisors allow running multiple OSs side-by-side, yet provide full isolation (i.e., security, resource and failure isolations). To mention a few, hypervisors can effectively authorize and multiplex accesses to physical resources. Besides, undesired interactions between VMs are sometimes referred to as cross-talks. Hypervisors can incorporate sophisticated resource schedulers and allocators to circumvent cross-talks. Lastly, hypervisors offer no sharing among OS distributions. The only code/data shared among VMs is indeed the hypervisor itself.

Nonetheless, the unique benefits offered by virtualization have some side effects. For instance, the degree of isolation comes at the cost of efficiency. Efficiency can be measured in terms of overall execution time. In general, VMs provide inferior performance as opposed to equivalent physical machines. This is mainly due to: (1) the overhead of context switching between VMs and the hypervisor and (2) the duplication of efforts by the hypervisor and the OSs running in VMs (e.g., all might be running schedulers, managing virtual memories, and interpreting I/O requests), among others.

1.3 Managing System Complexity

Modern computers are among the most advanced human-engineered structures. These structures are typically very complex. Such complexity stems from incorporating various silicon chips embodied as processors, memories, disks, displays, keyboards, mice, network interfaces, and others, upon which programs are operated and services are offered. The key to managing complexity in computer systems is by dividing system components into levels of abstractions separated by well-defined interfaces.

1.3.1 Levels of Abstractions

The first aspect of managing computer complexity is by abstracting computer components. For instance, gates are built on electronic circuits, binary on gates, machine languages on binary,

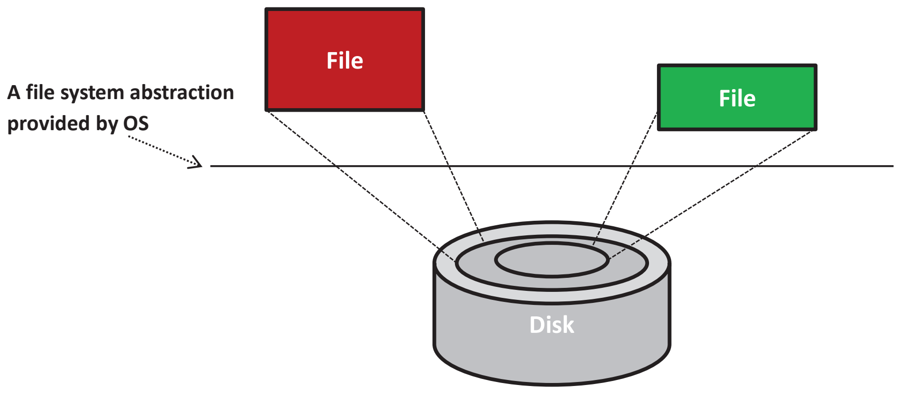


Figure 1.4: Files are abstractions of disks.

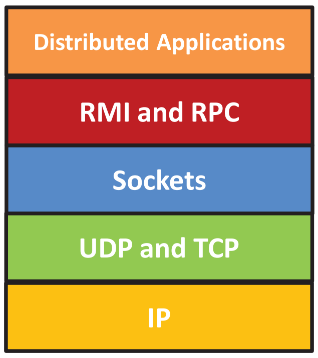


Figure 1.5: Abstraction layers in distributed systems.

programming languages on machine languages, and operating systems along with associated applications on programming languages. Another abstraction that almost every computer user understands and uses is the file. Files are abstractions of disks. As demonstrated in Fig. 1.4, the details of a hard disk are abstracted by the OS so that disk storage appears to applications as a set of variable-size files. Consequently, programmers need not worry about locations and sizes of cylinders, sectors, and tracks or bandwidth allocations at disk controllers. They can simply create, read, and write files without knowledge of the way the hard disk is constructed or organized.

Another common example of abstraction is the process. Processes are abstractions of CPUs and memories. Hence, programmers need not worry whether their processes will monopolize CPUs or consume full memory capacities. The OS creates and manages all users processes without any involvement from users. Abstractions for displays are also delivered through drawing packages, windowing packages, and the like. Mouse clicks are abstracted as invocations to program functions. Key-down events at keyboards are abstracted as inputs of characters. Finally, abstractions for networks include layers/protocols such as IP, UDP and TCP. As shown in Fig. 1.5, distributed systems (e.g., the cloud) build upon network layers and involve extra ones such as sockets, RMIs, and RPCs, among others. In short, the ability of abstracting system components enables simplified design, programming and use of computer systems.

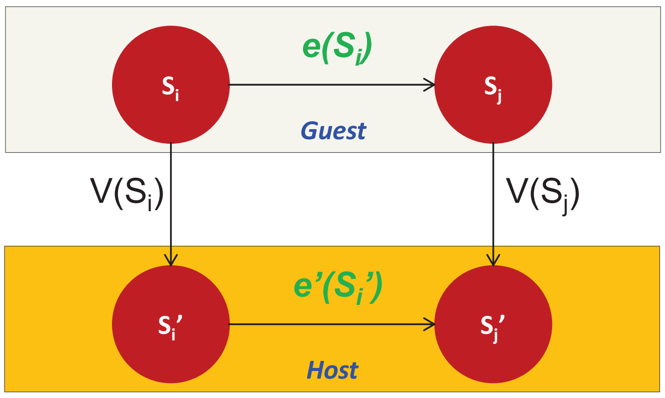


Figure 1.6: Virtualization Isomorphism.

To this end, we note that abstractions could be applied at the hardware or software levels. At the hardware level, components are physical (e.g., CPU and RAM). Conversely, at the software level, components are logical (e.g., RMI and RPC). In this chapter we are most concerned with abstractions at the software or near the hardware/software levels.

1.3.2 Well-Defined Interfaces

A system (or subsystem) interface is defined as a set of function calls that allows leveraging the underlying systems functionalities without needing to know any of its details. The two most popular interfaces in systems are the Application Programming Interface (API) and the Instruction Set Architecture (ISA) interface. Another interface that is less popular, yet very important (especially in virtualization), is the Application Binary Interface (ABI). API is used by high-level language (HLL) programmers to invoke some library or OS features. An API includes data types, data structures, functions, and object classes, to mention a few. An API enables compliant applications to be ported easily (via recompilation) to any system that supports the same API. As the API deals with software source codes, the ABI is a binary interface. The ABI is essentially a compiled version of the API. Hence, it lies at the machine language level. With ABI, system functionalities are accessed through OS system calls. OS system calls provide a specific set of operations that the OS can perform on behalf of user programs. A source code compiled to a specific ABI can run unchanged only on a system with the same OS and ISA. Finally, ISA defines a set of storage resources (e.g., registers and memory) and a set of instructions that allows manipulating data held at storage resources. ISA lies at the boundary between hardware and software. As discussed later in the chapter, ABI and ISA are important in defining virtual machine types.

1.4 What is Virtualization?

Formally, virtualization involves the construction of an isomorphism that maps a virtual guest system to a real host system [48]. Fig. 1.6 illustrates the virtualization process. The function $\boldsymbol{V}$ in the figure maps guest state to host state. For a sequence of operations, $\boldsymbol{e}$, that modifies a guest

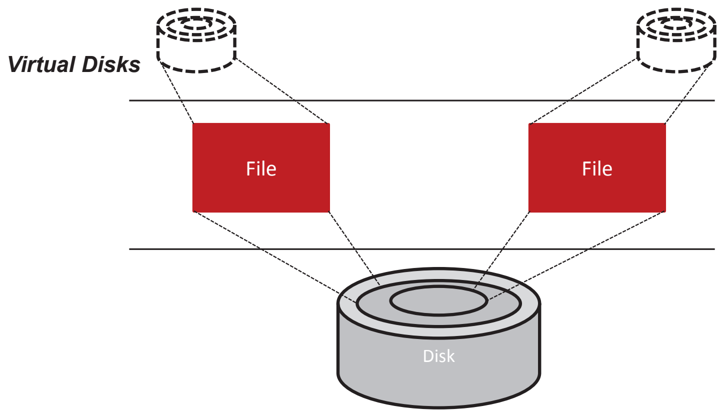


Figure 1.7: Constructing virtual disks by mapping their contents to large files.

state, there is a corresponding sequence of operations, $e\prime$, in the host that performs equivalent modifications. Informally, virtualization creates virtual resources and maps them to physical resources. Virtual resources are formed based on physical resources and act as proxies to them.

The concept of virtualization can be applied to either a system component or to an entire machine. Traditionally, virtualization has been applied to only the memory component in general-purpose OSs, providing what is known as the *virtual memory*. In a revisit to the hard disk example in Fig. 1.4, some applications might desire multiple hard drives. To satisfy such a requirement, the physical hard drive can be partitioned into multiple virtual disks as shown in Fig. 1.7. Each virtual disk will be offered *logical* cylinders, sectors and tracks. This keeps the level of detail analogous to what is offered by general-purpose OSs, yet at a different interface, and actually without being abstracted. The hypervisor can map (the function V in the isomorphism) a virtual disk to a single large file on the physical disk. Afterwards, to carry a read/write operation on a virtual disk (the function e in the isomorphism), the hypervisor interprets the operation as a file read/write followed by an actual disk read/write (the function $e\prime$ in the isomorphism).

As opposed to a single system component, when virtualization is applied to an entire machine, it provides what is known as a *virtual machine* (VM). Specifically, a full set of hardware resources, including processors, memory, and I/O devices will be virtualized to provide the VM. As shown in Fig. 1.8, an underlying hardware machine is usually referred to as *host* and an OS running on a VM is denoted as *guest OS*. A VM can only run at a single host at a time. As compared to a host, a VM can have resources different in quantity and in type. For instance, a VM can obtain more processors than what a host offers and can run an ISA that is different than that of the host. Lastly, every VM can be booted, shut down, and rebooted just like a regular host. Further details on VMs and their different types will be provided in the next section.

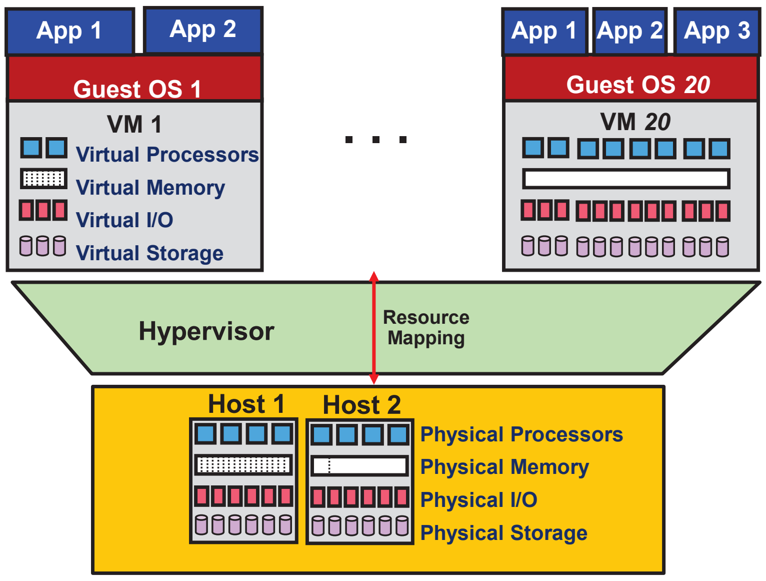


Figure 1.8: Virtualization as applied to an entire physical system. An OS running on a VM is referred to as a ***Guest OS*** and every physical machine is denoted as a ***Host***. Compared to a Host, a VM can have virtual resources different in *Quantity* and *Type*.

1.5 Virtual Machine Types

There are two main implementations of virtual machines (VMs): *process VMs* and *system VMs*. We will first discuss process VMs and then system VMs.

1.5.1 Process Virtual Machines

A process VM is a virtual machine capable of supporting an individual process as long as the process is alive. Fig. 1.9 (a) demonstrates process VMs. A process VM terminates when the hosted process ceases. From a process VM perspective, a machine consists of a virtual memory address space, user-level registers and instructions assigned to a single process so as to execute a user program. Based on this definition, a regular process in a general-purpose OS can also be deemed a machine. However, a process in an OS can only support user program binaries compiled for the ISA of the host machine. In other words, executing binaries compiled for an ISA different than that of the host machine cannot be ensued with regular processes. Conversely, a process VM allows that to happen via what is denoted as *emulation*. As shown in Fig. 1.10, emulation is the process of allowing the interfaces and functionalities of one system (the source) to be employed on a system with different interfaces and functionalities (the target). Emulation will be discussed in detail in Section 1.6.3. The abstraction of the process VM is provided by a piece of a virtualizing software called the *runtime* (see Fig. 1.9 (a)). The runtime is placed at the Application Binary Interface (ABI), on top of the host OS and the underlying hardware. It is this runtime that emulates the VM instructions and/or system calls when guest and host ISAs are different.

Finally, a process VM may not directly correspond to any physical platform but employed mainly to offer cross-platform portability. Such kinds of process VMs are known as High Level

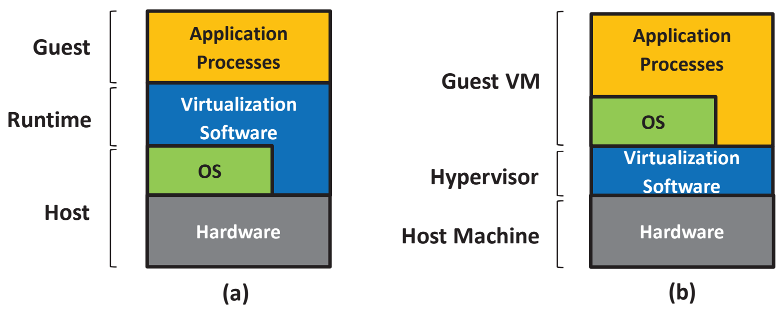


Figure 1.9: Virtual machine types. (a) Process virtual machines, and (b) system virtual machines.

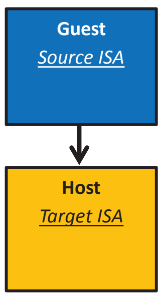


Figure 1.10: The emulation process.

Language virtual machines (HLL VMs). An HLL VM abstracts away details of the underlying hardware resources and the OS, and allows programs to run in the same way on any platform. Java VM (JVM) [36] and Microsoft common language infrastructure (CLI) [42] are examples of HLL VMs. In summary, a process VM is similar to a regular process running on an OS. However, a process VM allows, through emulation, the execution of an application compiled for an ISA different than that of the host machine.

1.5.2 System Virtual Machines

Contrary to process VMs, a system VM is a virtual machine capable of virtualizing a full set of hardware resources including processors, memories, and IO devices, thus providing a complete system environment. A system VM can support an OS along with its associated processes as long as the system environment is alive. Fig. 1.9 (b) illustrates system VMs. As defined previously, the hypervisor (or the virtual machine monitor (VMM)) is a piece of software that provides abstraction for the system VM. It can be placed at the ISA level directly on top of the raw hardware and below system images (e.g., OSs). The hardware resources of the host platform can be shared among multiple guest VMs. The hypervisor manages the allocation of, and access to, the hardware resources to/by the guest VMs. In practice, the hypervisor provides an elegant way to logically isolate multiple guest VMs sharing a single physical infrastructure (e.g., the cloud datacenters). Each guest VM is given the illusion of acquiring the hardware resources of the underlying physical machine.

There are different classes of system VMs. Fig. 1.11 exhibits three of these classes as well

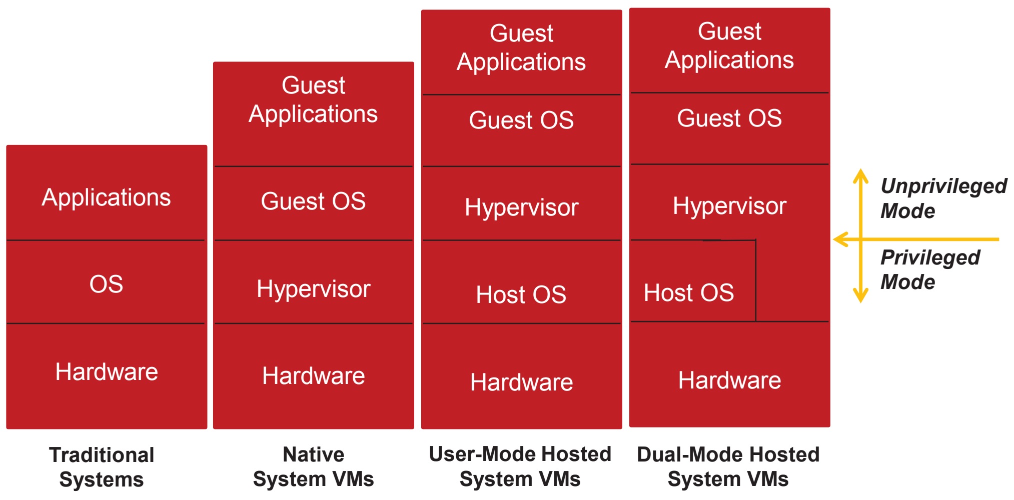


Figure 1.11: Different system VM classes.

as traditional systems. In a conventional time-shared system, the OS runs in privileged mode (system mode) while the applications associated with it run in unprivileged mode (user mode) (more details on execution modes will be discussed in Section 1.6.1). With system virtualization, however, the guest OS(s) will run in unprivileged mode while the hypervisor can operate in privileged mode. Such a system is denoted as *native system VM*. In native system VM, every privileged instruction issued by a user program at any guest OS has to trap to the hypervisor. In addition, the hypervisor needs to specify and implement every function required for managing hardware resources. In contrary, if the hypervisor operates in unprivileged mode on top of a host OS, the guest OS(s) will also operate in unprivileged mode. This system is called *user-mode hosted system VM*. In this case privileged instructions from guest OS(s) still need to trap to the hypervisor. In return, the hypervisor needs also to trap to the host OS. Clearly, this increases the overhead by adding 1 more trap per every privileged instruction. Nonetheless, the hypervisor can utilize the functions already available on the host OS to manage hardware resources. Finally, the hypervisor can operate partly in privileged mode and partly in user mode in a system referred to as *dual-mode hosted system VM*. This way, the hypervisor can make use of the host OS's resource management functions and also preclude the 1 more trap per each privileged instruction incurred in user-mode hosted system VMs.

1.6 CPU Virtualization

Virtualizing a CPU entails two major steps: (1) multiplexing a physical CPU (pCPU) among virtual CPUs (vCPUs) associated with virtual machines (this is usually referred to as vCPU scheduling), and (2) virtualizing the ISA of a pCPU in order to make vCPUs with different ISAs run on this pCPU. First, we present some conditions for virtualizing ISAs. Second, we describe ISA virtualization. Third, we make distinction between two types of ISA virtualization, full virtualization and paravirtualization. Fourth, we discuss Emulation, a major technique for virtualizing

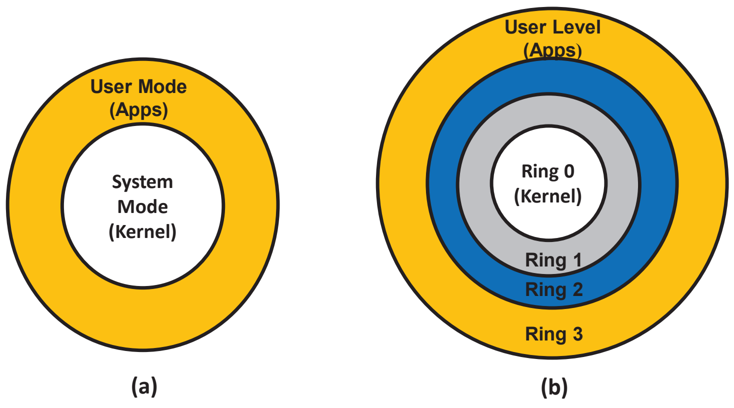


Figure 1.12: System modes of operation (or rings). (a) Simple ISAs have to modes of operation, and (b) Intel's IA-32 allows 4 rings.

CPUs. Fifth, we recognize between two kinds of VMs, Simultaneous Multiprocessing (SMP) and Uniprocessors (UP) VMs. Finally, we close with a discussion on vCPU scheduling. As examples, we present two popular Xen vCPU schedulers.

1.6.1 The Conditions for Virtualizing ISAs

The key to virtualize a CPU lies in the execution of both, privileged and unprivileged instructions issued by guest virtual processors. The set of any processor instructions is documented and provided in the ISA. Besides, special privileges to system resources are permitted by defining modes of operations (or rings) in the ISA. Each CPU ISA usually specifies two modes of operations, system (or supervisor/kernel/privileged) mode and user mode (see Fig. 1.12 (a)). System mode allows a wide accessibility to system components while user mode restricts such accessibility. In an attempt to provide security and resource isolations, OSs in traditional systems are executed in system mode while associated applications are run in user mode. Some ISAs, however, support more than two rings. For instance, the Intel IA-32 ISA supports four rings (see Fig. 1.12 (b)). In traditional systems, when Linux is implemented on an IA-32 ISA, the OS is executed in ring 0 and application processes are executed in ring 3.

A privileged instruction is defined as one that traps in user mode and does not trap in system mode. A trap is a transfer of control to system mode, wherein the hypervisor (as in virtualization) or the OS (as in traditional OSs) performs some action before switching control back to the originating process. Traps occur as side effects of executing instructions. Overall, instructions can be classified into two different categories: ***Sensitive*** and ***Innocuous***. Sensitive instructions can be either ***Control-Sensitive*** or ***Behavior-Sensitive***. Control sensitive instructions are those that attempt to modify the configuration of resources in a system such as changing the mode of operation or CPU timer. An example of control-sensitive instructions is Load Processor Status Word (LPSW) (IBM System/370). LPSW loads the processor status word from a location in memory if the CPU is in system mode and traps otherwise. LPSW contains bits that determine

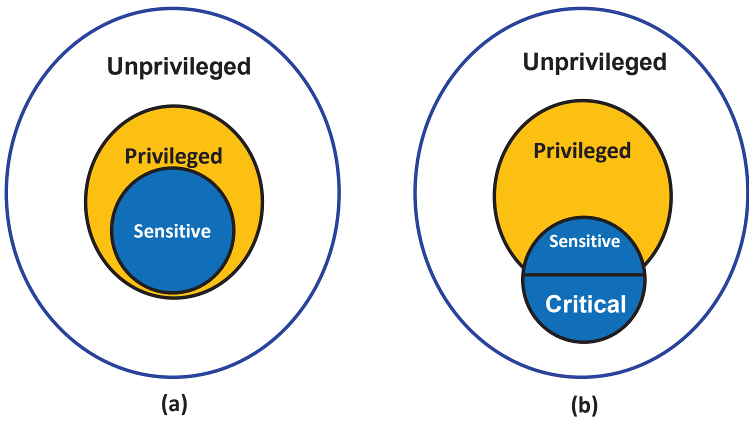


Figure 1.13: Demonstrating Popek and Goldberg's theorem. (a) Satisfies Popek and Goldberg's theorem, and (b) Does not satisfy Popek and Goldberg's theorem.

the state of the CPU. For instance, one of these bits is the *P* bit which specifies whether the CPU is in user mode or in system mode. If executing this instruction is allowed in user mode, a malicious program can easily change the mode of operation to privileged and obtain control over the system. Hence, to protect the system, such an instruction can only be executed in system mode. Behavior-Sensitive instructions are those whose behaviors are determined by the current configuration of resources in a system. An example of behavior-sensitive instructions is Pop Stack into Flags Register (POPF) (Intel IA-32). POPF pops the flag registers from a stack held in memory. One of these flags, known as the interrupt enable flag, can be altered only in system mode. If POPF is executed in user mode by a program that attempts to pop the interrupt enable flag, POPF will act as a *no-op* (i.e., no operation) instruction. Therefore, the behavior of POPF depends on the mode of operation, thus rendering behavior-sensitive. Finally, if the instruction is neither Control-Sensitive nor Behavior-Sensitive, it is Innocuous.

According to Popek and Goldberg [48], a hypervisor can be constructed if it satisfies three properties, *efficiency*, *resource control*, and *equivalence*. Efficiency entails executing all innocuous instructions directly on hardware without any interference from the hypervisor. Resource control suggests that it is not possible for any guest software to change the configuration of resources in a system. Equivalence requires identical behavior of a program running on a VM versus running on a traditional OS. One exception can be a difference in performance. Popek and Goldberg's proposal (or indeed theorem) implies that a hypervisor can only be constructed if the set of sensitive instructions is a subset of the set of privileged instructions. That is to say; instructions that interfere with the correct functioning of the system (i.e., sensitive instructions such as LPSW) should always trap in user mode. Fig. 1.13 (a) illustrates Popek and Goldberg's theorem.

Finally, let us discuss how a trap can be handled in a system. Specifically, we will describe traps in the context of CPU virtualization. Fig. 1.14 demonstrates how a hypervisor can handle

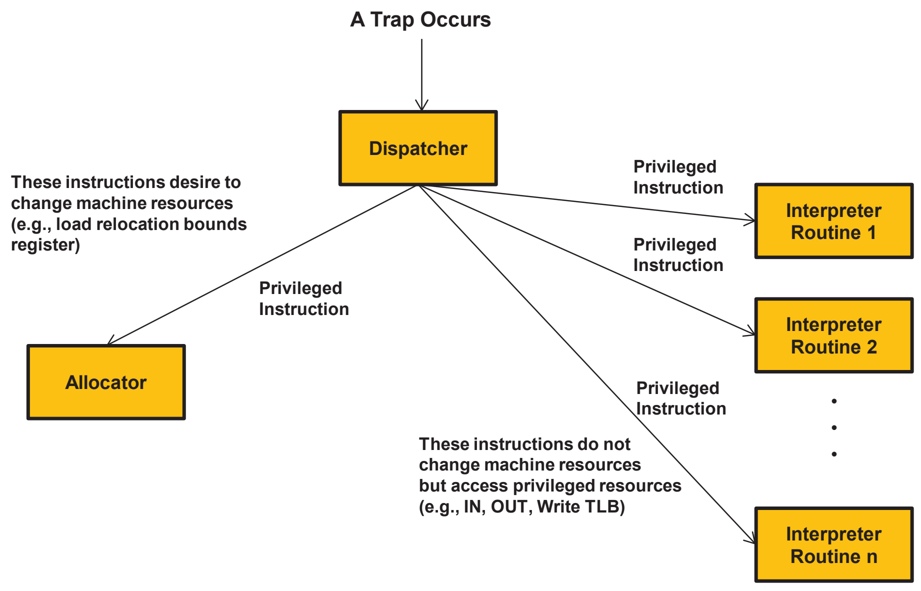

A Trap Occurs
Dispatcher
These instructions desire to change machine resources (e.g., load relocation bounds register)
Privileged Instruction
Allocator
Privileged Instruction
Interpreter Routine 1
Privileged Instruction
Interpreter Routine 2
Privileged Instruction
These instructions do not change machine resources but access privileged resources (e.g., IN, OUT, Write TLB)
Interpreter Routine n

Figure 1.14: Demonstrating a trap to a hypervisor. The hypervisor includes three main components, the dispatcher, the allocator and the interpreter routines.

an instruction trap. The hypervisor's trap handling functions can be divided into three main parts: *dispatcher*, *allocator*, and a set of *interpreter routines*. First, a privileged instruction traps to the hypervisor's dispatcher. If the hypervisor recognizes that the instruction is attempting to alter system resources, it directs it to the allocator; otherwise it sends it to a corresponding interpreter routine. The allocator decides how system resources are to be allocated in a non-conflicting manner, and satisfies the instruction's request accordingly. The interpreter routines emulate (more on emulation shortly) the effects of the instruction when operating on virtual resources. When the instruction is fully handled (i.e., done), control is passed back to the guest software at the instruction that comes immediately after the one that caused the trap.

1.6.2 Full Virtualization and Paravirtualization

A problem arises when an instruction that is sensitive but unprivileged is issued by a process running on a VM in user mode. According to Popek and Goldberg [48], sensitive instructions have to trap to the hypervisor if executed in user mode. However, as explained earlier, sensitive instructions can be privileged (e.g., LPSW) and unprivileged (e.g., POPF). Unprivileged instructions do not trap to the hypervisor. Instructions that are sensitive and unprivileged are called ***critical*** (see Fig. 1.13 (b)). ISAs that contain critical instructions do not satisfy Popek and Goldberg's theorem. The challenge becomes, can a hypervisor be constructed in the presence of critical instructions? The answer is yes. Nonetheless, J. Smith and R. Nair [55] distinguish between a hypervisor that complies with Popek and Goldberg's theorem and a one that does not, by referring to the former as a true or efficient hypervisor and to the latter simply as a hypervisor.

A hypervisor can be typically constructed by using ***full virtualization*** and/or ***paravirtualization***. Full virtualization emulates all instructions in the ISA. Emulation degrades performance as it implies reproducing the behavior of every source instruction, by first translating it to a target instruction and then running it on a target ISA (we describe emulation in the next section). Paravirtualization deals with critical instructions by modifying guest OSs. Specifically, it entails re-writing every critical instruction as a *hypercall* that traps to the hypervisor. As such, paravirtualization improves performance due to totally avoiding emulation, but at the expense of modifying guest OSs. In contrast, full virtualization avoids modifying guest OSs, but at the expense of degrading system performance. As examples, VMWare uses full virtualization [60] while Xen employs paravirtualization [9, 47].

1.6.3 Emulation

Now that we understand the conditions for virtualizing ISAs and the two main classes of CPU virtualization, full virtualization and paravirtualization, we move on to discussing emulation as being a major technique for implementing full virtualization and process VMs. Emulation has been introduced in Section 1.5.1. To recap, emulation is the process of allowing the interfaces and functionalities of one system (the source) to be implemented on a system with different interfaces and functionalities (the target). Emulation is the only CPU virtualization mechanism available when the guest and host ISAs are different. If the guest and host ISAs are identical, direct native execution can be possibly applied.

Emulation is carried out either via ***interpretation*** or ***binary translation***. With interpretation, source instructions are converted to relevant target instructions, one instruction at a time. Interpretation is relatively slow because of emulating instructions one-by-one and not applying any optimization technique (e.g., avoiding the interpretation of an already encountered and interpreted instruction). Binary translation optimizes upon interpretation by converting blocks of source instructions to target instructions and caching generated blocks for repeated use. Typically, a block of instructions is more amenable to optimizations than a single instruction. As compared to interpretation, binary translation is much faster because of applying block caching as well as code optimizations over blocks.

There are three major interpretation schemes, ***decode-and-dispatch***, ***indirect-threaded*** and ***direct-threaded*** [55]. Basically, an interpreter should read through the source code instruction-by-instruction, analyze each instruction, and call relevant routines to generate the target code. This is actually what the decode-and-dispatch interpreter does. Fig. 1.15 exhibits a snippet of code for a decode-and-dispatch interpreter used for interpreting the PowerPC ISA. As shown, the interpreter is structured around a central loop and a switch statement. Each instruction is first decoded (i.e., the extract() function) and subsequently dispatched to a corresponding routine, which in return performs the necessary emulation. Clearly, such a decode-and-dispatch strategy

```
while (!halt && !interrupt) {
        inst = code [PC];
        opcode = extract(inst, 31, 6);
        switch( opcode ) {
        case LoadWordAndZero: LoadWordAndZero_Routine(inst);
        case ALU: ALU_Routine(inst);
        case Branch: Branch_Routine(inst);
        ....
        ....
        }
}

LoadWordAndZero_Routine(inst){
        RT = extract(inst, 25, 5);
        RA = extract(inst, 20, 5);
        displacement = extract(inst, 15, 16);
        if(RA == 0)
                source = 0;
        else
                source = regs[RA];

        address = source + displacement;
        regs[RT] = (data[address] << 32) >> 32;
        PC = PC + 4;
}
.....
.....
```

Figure 1.15: A snippet of code for a decode-and-dispatch interpreter. Interpreter routines are all omitted except one for the brevity of the presentation (example from [55]).

results in a number of direct and indirect branch instructions. Specifically, an indirect branch for the switch statement, a branch to an interpreter routine, and a second indirect branch to return from the interpreter routine will be incurred per each instruction. Furthermore, with decode-and-dispatch, every time the same instruction is encountered, its respective interpreter routine is invoked. This, alongside of excessive branches, tend to greatly degrade performance.

As an optimization over decode-and-dispatch, the indirect-threaded interpreter attempts to escape some of the decode-and-dispatch branches by appending (or *threading*) a portion of the dispatch code to the end of each interpreter routine [20, 32, 34]. This precludes most of the branches incurred in decode-and-dispatch, yet keeps invoking an interpreter routine every time the same instruction is decoded. To address such a drawback, the direct-threaded interpreter capitalizes on the indirect-threaded one and seeks to interpret a repeated operation only once [10]. This is achieved by saving away the extracted information of instructions in an intermediate form for future references [39]. Although the direct-threaded interpreter improves upon the indirect-threaded one, it limits portability because of the dependency of the intermediate form on the exact locations of the interpreter routines (the addresses of the interpreter routines are saved in the intermediate form). Moreover, the size of the intermediate form is proportional to the original source code, thus resulting in vast memory requirements.

Finally, it has been observed that performance can be significantly enhanced by mapping each individual source binary instruction to its own customized target code [55].This process of converting the source binary program into a target binary program is referred to as *binary translation* [54]. As pointed out earlier, binary translation suggests improving upon the direct-threaded interpreter via: (1) translating a block of source binary instructions to a block of target binary instructions, one at a time, and (2) caching the translated blocks for future use. It is possible to binary translate a program in its entirety before execution. Such a scheme is called *static binary translation* [55]. However, a more general approach is to translate the binary instructions during the program execution and interpret new sections of code incrementally as encountered by the program. This mechanism is denoted as *dynamic binary translation* [27,55].

Table 1.1: A Qualitative Comparison of Different Emulation Techniques.

	Memory Requirements	**Start-Up performance**	**Steady-State Performance**	**Code Portability**
Decode-and-Dispatch Interpreter	Low	Fast	Slow	Good
Indirect Threaded Interpreter	Low	Fast	Slow	Good
Direct Threaded Interpreter	High	Slow	Medium	Medium
Binary Translation	High	Very Slow	Fast	Poor

To this end, Table 1.1 qualitatively compares binary translation, decode-and-dispatch, indirect-threaded, and direct-threaded emulation techniques in terms of four metrics, memory requirements, start-up performance, steady-state performance, and code portability (a quantitative performance evaluation can be found in [52]). To exemplify, the decode-and-dispatch interpreter row reads as follows. First, with decode-and-dispatch, memory requirements remain low. This is because of having only one interpreter routine per each instruction type in the target ISA. Alongside, the decode-and-dispatch interpreter averts threading the dispatch code to the end of each routine, thus inherently reduces the pressure on the memory capacity. Second, start-up performance is fast because neither using intermediate forms nor caching translated blocks are adopted. Third, steady-state performance (i.e., the performance after starting up the interpreter) is slow because of: (1) the high number of branches, and (2) the interpretation of every instruction upon every appearance. Finally, code portability is good since saving addresses of interpreter routines (as is the case with direct-threaded interpreters) and caching ISA-dependent translated binary code are totally avoided.

1.6.4 Uniprocessor and Multiprocessor VMs

As described earlier in the chapter, a virtual CPU (vCPU) acts as a proxy to a physical CPU (pCPU). In other words, a vCPU is a representation of a pCPU to a guest OS. A vCPU can be initiated within a VM and mapped to an underlying pCPU by the hypervisor. In principle, a

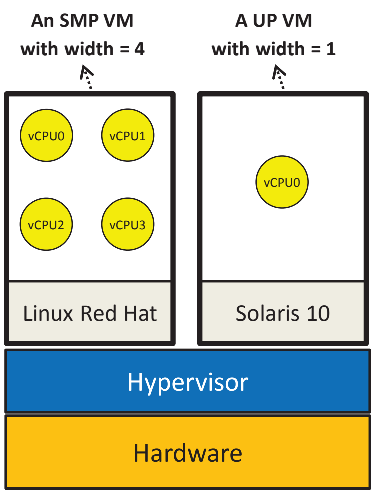


Figure 1.16: SMP and UP VMs. A VM is defined whether it is SMP or UP by its width. Width is the number of vCPUs in a VM. If the width of a VM = 1, the VM is UP, otherwise, it is SMP.

VM can have one or many vCPUs. For instance, a VM in VMWare ESX 4 can have up to 8 vCPUs [61]. This is usually referred to as the *width* of a VM. A VM with a width greater than 1 is denoted as *Symmetric Multiprocessing* (SMP) VM. In contrary, a VM with a width equal to 1 is referred to as *Uniprocessor* (UP) VM. Fig. 1.16 demonstrates an SMP native system VM with a width of 4 and a UP native system VM, both running on the same hardware.

Similar to a process on a general-purpose OS, a vCPU can be in different states such as running, ready and wait states. At a certain point in time, a vCPU can be scheduled by the hypervisor at only a single core (akin to scheduling an OS process at a core). For instance, a UP VM running on a host machine equipped with 2 x Xeon 5405 (i.e., total of 8 pCPUs) will run on only 1 of the 8 available cores at a time. Inherently parallel workloads, such as MapReduce applications, prefer SMP VMs. We next discuss how the hypervisor schedules vCPUs on pCPUs.

1.6.5 Virtual CPU Scheduling and Xen's Schedulers

General-Purpose OSs support two levels of scheduling, process and thread scheduling. With a hypervisor, one extra level of scheduling is added; that is, vCPU scheduling. The hypervisor schedules vCPUs on the underlying pCPU(s), thereby providing each guest VM with a portion of the underlying physical processing capacity.

We *briefly* discuss two popular schedulers from Xen, *Simple Earliest Deadline First* (SEDF) and *Credit Scheduler* (CS) [16, 18]. As its name suggests, SEDF is simple, whereby only two parameters, $\boldsymbol{n}$ (the slice) and $\boldsymbol{m}$ (the period), are involved. A VM (or domain U_i in Xen's parlance) can request $\boldsymbol{n}$ every $\boldsymbol{m}$. SEDF specifies a deadline for each vCPU computed in terms of $\boldsymbol{n}$ and $\boldsymbol{m}$. The deadline is defined as the *latest time* a vCPU should be scheduled within the period $\boldsymbol{m}$. For instance, a domain U_i can request $\boldsymbol{n}$ = 10ms and $\boldsymbol{m}$ = 100ms. Accordingly, a vCPU at this domain can be scheduled by SEDF as late as 90ms into the 100ms period, yet still meet its deadline.

SEDF operates by searching across the set of all runnable vCPUs, held in a queue, and selecting the one with the earliest deadline.

Xen's Credit Scheduler (CS) is more involved than SEDF. First, when creating a VM, each vCPU in the VM is configured with two properties, the *weight* and the *cap*. The weight determines the share of a pCPU's capacity that should be provided to a vCPU. For instance, if two vCPUs, vCPU-1 and vCPU-2 are specified with weights of 256 (the default) and 128, respectively, vCPU-1 will obtain double the shares of vCPU-2. Weights can range from 1 to 65535. The cap determines the total percentage of a pCPU that should be given to a vCPU. The cap can modify the behavior of the weight. Nevertheless, a vCPU can be kept uncapped.

CS converts each vCPU's weight to credits. Credits will be deducted from a vCPU as long as it is running. A vCPU is marked as *over* once it runs out of credits and *under* otherwise. CS maintains a queue per each pCPU (assuming a chip multiprocessors architecture), and stores all *under* vCPUs first, followed by all *over* vCPUs. CS operates by picking the first *under* vCPU in the queue to go next. CS keeps track of each vCPU's credits, and when switching a vCPU out, it places it in the queue at the end of the appropriate category. Finally, CS applies load balancing by allowing a pCPU with no *under* vCPUs to pull *under* vCPUs from queues of other pCPUs.

Xen is an open source hypervisor. Hence, it is possible to devise and add your own scheduler. Chisnall [16] provides an excellent and comprehensive coverage of Xen's internals as well as step-by-step instructions for adding a new scheduler to Xen.

1.7 Memory Virtualization

In essence, a VM can be deemed as a generalization of the classical virtual memory concept. Memory virtualization in a system VM is simply an extension to that concept. Virtual memory in traditional systems distinguishes between the virtual (or logical) view of memory as seen by a user program and the actual (or physical) memory as managed by the OS. We next provide a brief overview of virtual memory then delve into memory virtualization in system VMs.

1.7.1 One-Level Page Mapping

Virtual memory is a well-known virtualization technique supported in most general-purpose OSs. The basic idea of virtual memory is that each process is provided with its own virtual address space, broken up into chunks called virtual pages. A page is a contiguous range of addresses. As shown in Fig. 1.17, virtual memory maps virtual pages to physical pages in what is denoted as a page table. We refer to this as *one-level page mapping* between two types of addresses, the virtual and the physical addresses. Each process in the OS has its own page table. A main

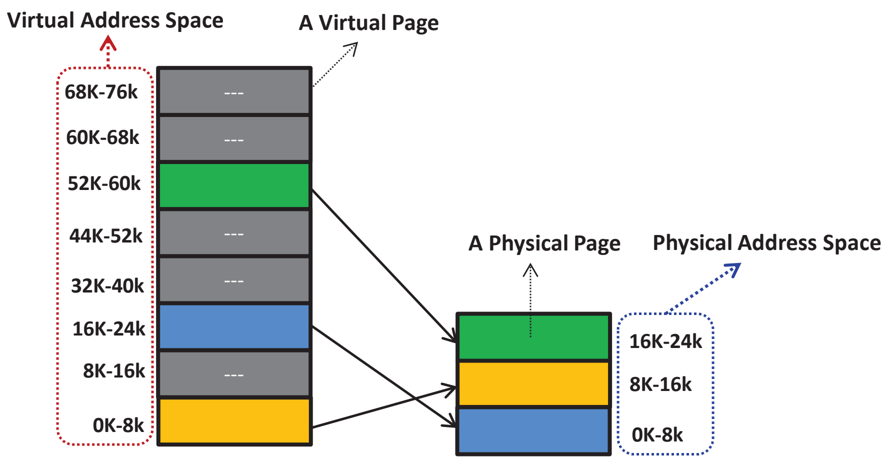


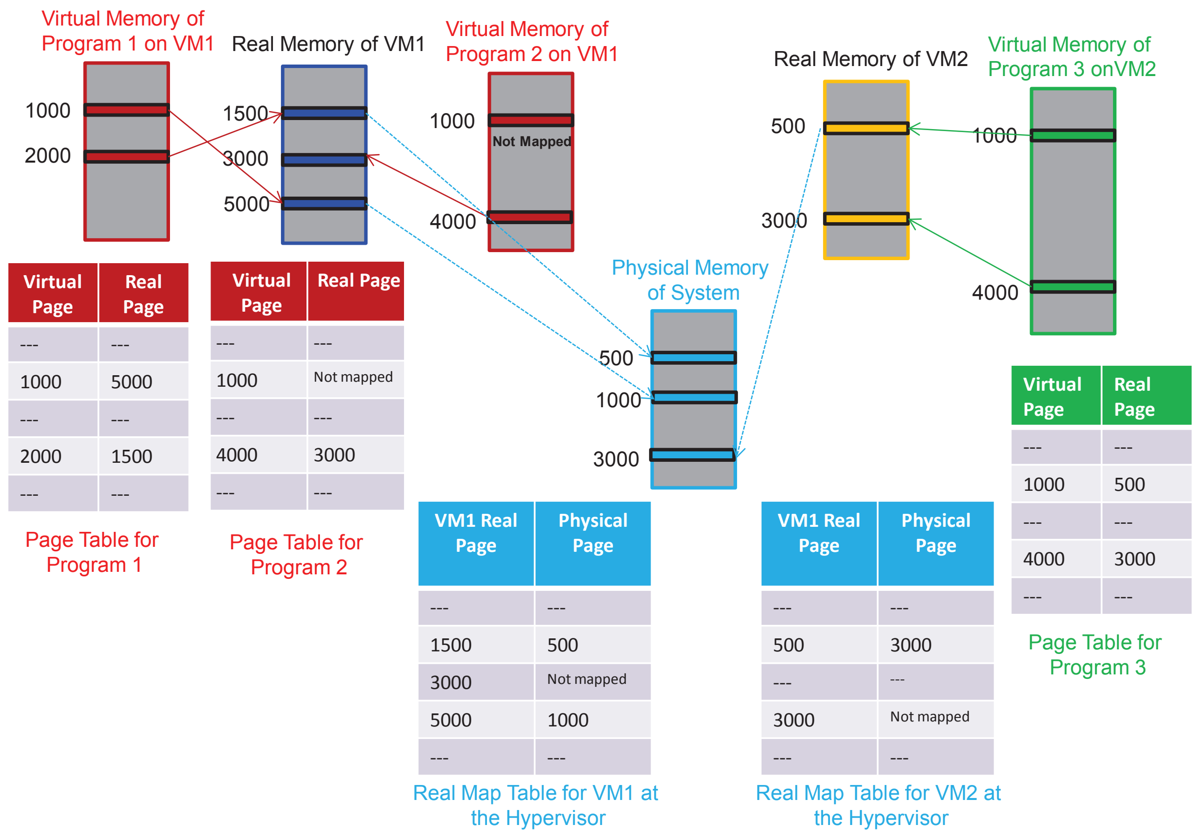
Figure 1.17: Mapping a process virtual address space to physical address space. This is captured in what is referred to as page table. Each process has its own page table.

observation pertaining to page tables is that not all virtual pages of a process need to be mapped to respective physical pages in order for the process to execute. When a process references a page that exists in the physical memory (i.e., there is a mapping between the requested virtual page and the corresponding physical page), a page hit is said to be attained. Upon a page hit, the requested physical page is fetched from the main memory with no further actions. In contrary, when a process references a page that does not exist in the physical memory, a page miss is said to be incurred. Upon a page miss, the OS is alerted to handle the miss. Subsequently, the OS fetches the missed page from disk storage and updates the relevant entry in the page table.

1.7.2 Two-Level Page Mapping

Contrary to OSs in traditional systems, with system virtualization, the hypervisor allocates a contiguous addressable memory space for each created VM (not process). This memory space per a VM is usually referred to as *real memory*. In return, each guest OS running in a VM allocates a contiguous addressable memory space for each process within its real memory. This memory space per a process is denoted as *virtual memory* (same name as in traditional systems). Each guest OS maps the virtual memories of its processes to the real memory of the underlying VM, while the hypervisor maps the real memories of its VMs to the system physical memory. Clearly, in contrast to traditional OSs, this entails two levels of mappings between three types of addresses, virtual, real and physical addresses. In fact, these virtual-to-real and real-to-physical mappings define what is known as system memory virtualization.

Like any general-purpose OS, a guest OS in a system VM would still own its set of page tables. In addition, the hypervisor would own another set of page tables for mapping real to physical addresses. The page tables in the hypervisor are usually referred to as *real map tables*. Fig. 1.18 demonstrates system memory virtualization in a native system VM. It shows page tables maintained by guest VMs and real map tables maintained by the hypervisor. Each entry in a page


Virtual Memory of Program 1 on VM1
Real Memory of VM1
Virtual Memory of Program 2 on VM1
Real Memory of VM2
Virtual Memory of Program 3 onVM2
1000
2000
1500
3000
5000
1000
Not Mapped
4000
500
3000
1000
4000
Physical Memory of System
500
1000
3000

Virtual Page	Real Page
---	---
1000	5000
---	---
2000	1500
---	---

Page Table for Program 1

Virtual Page	Real Page
---	---
1000	Not mapped
---	---
4000	3000
---	---

Page Table for Program 2

VM1 Real Page	Physical Page
---	---
1500	500
3000	Not mapped
5000	1000
---	---

Real Map Table for VM1 at the Hypervisor

VM1 Real Page	Physical Page
---	---
500	3000
---	---
3000	Not mapped
---	---

Real Map Table for VM2 at the Hypervisor

Virtual Page	Real Page
---	---
1000	500
---	---
4000	3000
---	---

Page Table for Program 3

Figure 1.18: Memory Virtualization in a Native System VM (example from [55]).

table maps a virtual page of a program to a real page in the respective VM. Likewise, each entry in a real map table maps a real page in a VM to a physical page in the physical memory. When a guest OS attempts to establish a valid mapping entry in its page table, it traps to the hypervisor. Subsequently, the hypervisor establishes a corresponding mapping in the relevant VM's real map table.

1.7.3 Memory Over-Commitment

In system memory virtualization, the combined total size of real memories can grow beyond the actual size of physical memory. This concept is typically referred to as *memory over-commitment* [62]. Memory over-commitment ensures that physical memory is highly utilized by active real memories (assuming multiple VMs running simultaneously). Indeed, without memory over-commitment, the hypervisor can only run VMs with a total size of real memories that is less than that of the physical memory. For instance, Fig. 1.19 shows a hypervisor with 4GB physical memory and 3 VMs, each with 2GB real memory. Without memory over-commitment, the hypervisor can only run 1 VM, for the fact of not having enough physical memory to assign to 2 VMs at once. Although each VM would require only 2GB of memory and the hypervisor has 4GB of physical memory, this cannot be afforded because the hypervisor generally requires overhead memories (e.g., to maintain various virtualization data structures).

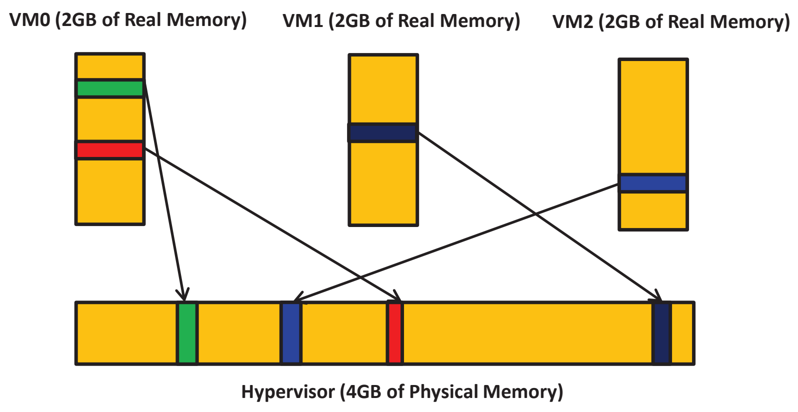


Figure 1.19: A hypervisor with 4GB of physical memory enabling 3 VMs at once with a total of 6GB of real memory.

To this end, in practical situations, some VMs might be lightly loaded while others might be heavily loaded. Lightly loaded VMs can cause some pages to sit idle, while heavily loaded VMs can result in memory page thrashing. To deal with such situations, the hypervisor can take (or steal) the inactive physical memory pages away from idle VMs and provide them to heavily loaded VMs. As a side note, hypervisors usually write zeros to the stolen/reclaimed inactive physical memory pages in order to avert information leaking among VMs.

1.7.4 Reclamation Techniques and VMWare Memory Ballooning

To maintain full isolation, guest OSs are kept unaware that they are running inside VMs. VMs are also kept unaware of the states of other VMs running on the same physical host. Furthermore, with multiple levels of page mapping, VMs remain oblivious of any physical memory shortage. Therefore, when the hypervisor runs multiple VMs at a physical host and the physical memory turns stressed, none of the VMs can automatically help in freeing up memory. The hypervisor deals with the situation by applying a *reclamation technique*. As its name suggests, a reclamation technique attempts to reclaim inactive real memory pages at VMs and make them available for the hypervisor upon experiencing a memory shortage. One popular reclamation techniques is *the ballooning process* incorporated in VMWare ESX [62].

In VMWare ESX, a balloon driver must be installed and enabled in each guest OS as a pseudo-device driver. The balloon driver regularly polls the hypervisor through a private channel to obtain a target balloon size. As illustrated in Fig. 1.20, when the hypervisor experiences memory shortage, it inflates the balloon by setting a proper target balloon size. Fig. 1.20 (a) shows 4 real memory pages mapped to 4 physical pages, out of which only 2 pages are actually active (the red and the yellow ones). Without involving the ballooning process, the hypervisor is unaware of the other 2 inactive pages (the green and the dark-blue ones) since they are still mapped to physical pages. Consequently, the hypervisor will not be able to reclaim inactive pages unless getting informed. With memory ballooning, however, the hypervisor can set the balloon target size to an integer number (say 2 or 3). When recognized by the balloon driver at the guest OS,

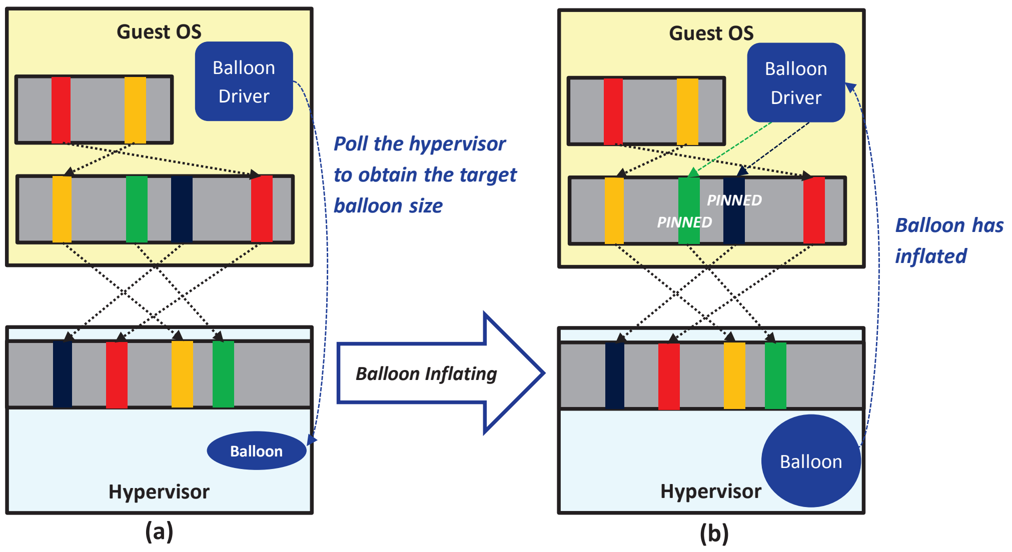


Figure 1.20: The ballooning process in VMWare ESX.

the driver checks out the pages, locates the 2 inactive ones, and pins them (see Fig. 1.20 (b)). The pinning process is carried out by the guest OS via ensuring that the pinned pages cannot be read/written by any process during memory reclamation. After pinning the inactive pages, the balloon driver transmits to the hypervisor the addresses of the pinned pages. Subsequently, the hypervisor proceeds safely with reclaiming the respective physical pages and allocating them to needy VMs. Lastly, to unpin pinned pages, the hypervisor deflates the balloon by setting a smaller target balloon size, and communicates that to the balloon driver. When received by the balloon driver, it unpins the pinned pages so as the guest OS can utilize them.

1.8 I/O Virtualization

The virtualization strategy for a given I/O device type consists of: (1) constructing a virtual version of that device and (2) virtualizing the I/O activity routed to the device. Typical I/O devices include disks, network cards, displays, and keyboards, among others. As discussed previously, the hypervisor might create a virtual display as a window on a physical display. In addition, a virtual disk can be created by assigning to it a fraction of the physical disk's storage capacity. After constructing virtual devices, the hypervisor ensures that each I/O operation is carried out within the bounds of the requested virtual device. For instance, if a virtual disk is allocated 100 cylinders from among 1000 cylinders provided by a physical disk, the hypervisor guarantees that no I/O request intended for that virtual disk can access any cylinder other than the 100 assigned to it. More precisely, the disk location in the issued I/O request will be mapped by the hypervisor to only the area where the virtual disk has been allocated on the physical disk. Next, we will cover some I/O basics and then move on to the details of I/O virtualization.

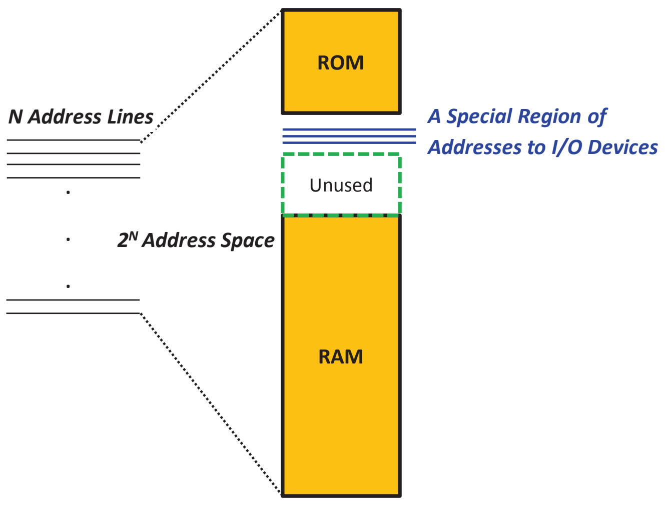


Figure 1.21: Memory mapped I/O with a specific region in the RAM address space for accessing I/O devices.

1.8.1 I/O Basics

To start with, each I/O device has a device controller. A device controller can be typically signaled either by a *privileged I/O instruction* or *memory-mapped I/O*. I/O instructions are provided by ISAs. Intel-32 is an example of a processor that provides I/O instructions within its ISA. Many recent processors, however, allow performing I/O between the CPU and the device controllers through memory-mapped I/O (e.g., RISC processors). As shown in Fig. 1.21, with memory-mapped I/O, a specific region of the physical memory address space is reserved for accessing I/O devices. These addresses are recognized by the memory controller as commands to I/O devices and do not correspond to memory physical locations. Different memory-mapped addresses are used for different I/O devices. Lastly, in order to protect I/O devices, both I/O instructions and memory-mapped addresses are handled in system mode, thus becoming privileged.

Because I/O operations are executed in system mode, user programs can only invoke them through OS system calls (assuming traditional systems). The OS abstracts most of the details of I/O devices and makes them accessible through only well-defined interfaces. Fig. 1.22 shows the three major interfaces that come into play when a user program places an I/O request. These are the *system call interface*, the *device driver interface*, and the *operation level interface*. Starting an I/O operation, a user I/O request causes an OS system call that transfers control to the OS. Next, the OS calls device drivers (a set of software routines) via the device driver interface. A relevant device driver routine converts the I/O request to an operation specific to the requested physical device. The converted operation is subsequently carried through the operation level interface to the corresponding physical device.

1.8.2 Virtualizing I/O Devices

I/O virtualization allows a single physical I/O device to be shared by more than one guest OS. Fig. 1.23 demonstrates multiple guest OSs in native system VMs sharing a single hardware ma-

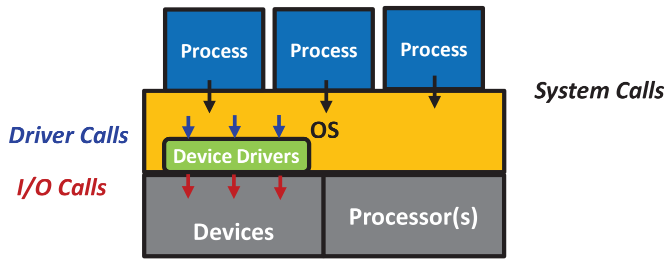


Figure 1.22: The three major interfaces involved in I/O operations, system call, device driver and operation level interfaces.

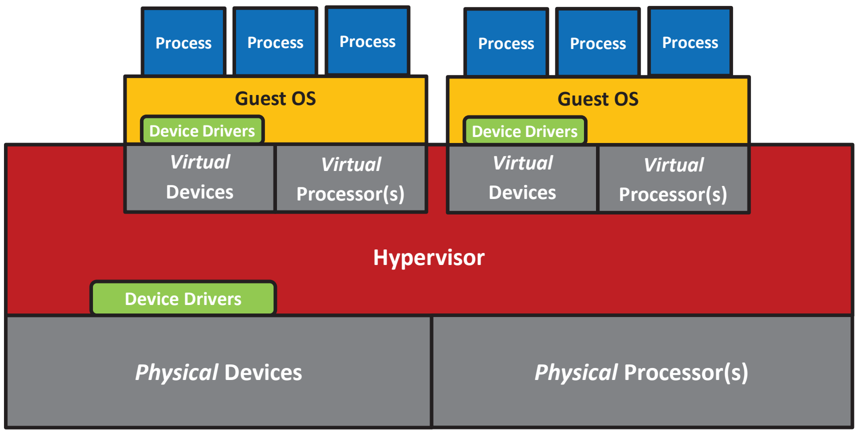


Figure 1.23: Logical locations of device drivers within multiple guest OSs in native system VMs sharing a single hardware machine.

chine. As shown, the hypervisor constructs virtual devices from physical devices. A main observation is that both the guest OSs and the hypervisor must have device drivers encapsulating the interfaces to the devices. This means that with virtualization, two different device drivers must be supported per each device, versus only one without virtualization [21]. In reality, this is a problem because vendors of devices usually supply drivers for only the major OSs, but not for hypervisors (though this could change in the near future). One way to circumvent such a problem is to co-locate the hypervisor with a major OS (e.g., Linux) on the same machine. This way, I/O requests can be handled by the OS which holds all the requisite I/O drivers. This is the approach that Xen adopts, and which we detail in the next section.

Moreover, with I/O virtualization, every I/O request issued by a user program at a guest VM should be intercepted by the hypervisor. This is because I/O requests are all sensitive, thus need to be controlled by the hypervisor. Clearly, this would necessitate a trap to the hypervisor per every I/O request. I/O requests are all privileged, whether issued using I/O instructions or memory-mapped I/O, hence, they are not critical instructions and can naturally trap to the hypervisor. In principle, the hypervisor can intercept I/O requests at any of the three interfaces: the system call interface, the device driver interface, or the operation level interface.

If the hypervisor intercepts an I/O request at the operation level interface, some essential information about the I/O action might be lost. In particular, when an I/O request arrives at the

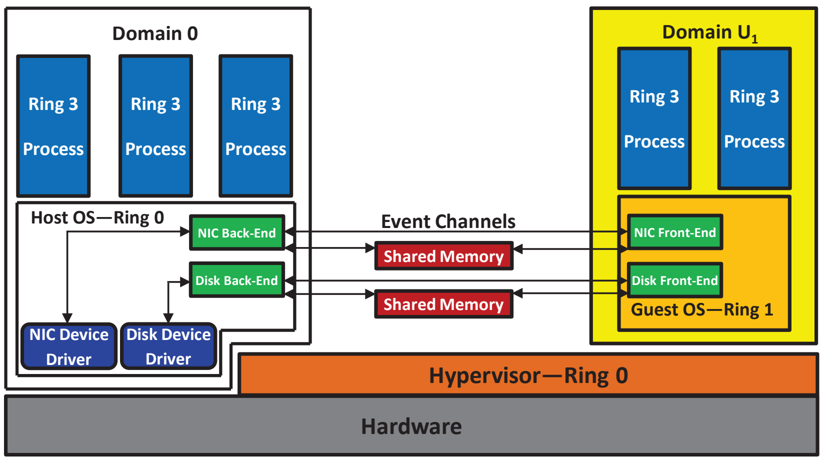


Figure 1.24: Xen's approach to I/O virtualization assuming a system with four rings (e.g., Intel-32). Xen co-locates the hypervisor with a host OS on the physical platform to *borrow* the host OS's device drivers and avoid coding them within the hypervisor. This makes the hypervisor *thinner* and accordingly more reliable. Besides, it makes it easier on the hypervisor developers.

device driver interface, it might get transformed into a sequence of instructions. For instance, a disk write is usually transformed into multiple store instructions, thus when received at the operation level interface, it becomes difficult on the hypervisor to identify the instructions as belonging to a single write request. As such, intercepting I/O requests at the operation level interface is typically avoided. In contrast, intercepting an I/O request at the device driver interface allows the hypervisor to efficiently map the request to the respective physical device and transmit it via the operation level interface. Clearly, this is a natural point to virtualize I/O devices; yet would oblige hypervisor developers to learn about all the device driver interfaces of various guest OSs so as to encode the approach. Finally, intercepting I/O requests at the system call interface (i.e., the ABI), might theoretically make the I/O virtualization process easier, whereby every I/O request can be entirely handled by the hypervisor (the solo controller in this case). To achieve that, however, the hypervisor has to emulate the ABI routines of every guest OS (different OSs have different ABI routines). Consequently, hypervisor developers need again to learn about the internals of every potential guest OS. Alongside, emulating ABI routines can degrade system performance due to the overhead imposed by the emulation process. To this end, intercepting I/O requests at the device driver interface is usually the most efficient approach, thus normally followed.

1.8.3 Xen's Approach to I/O Virtualization

As a concrete example, we discuss Xen's approach to I/O virtualization. As we pointed out earlier, to get around the problem of having device drivers for the hypervisor as well as the guest OSs, Xen co-locates its hypervisor with a traditional general-purpose OS. Fig. 1.24 shows a host OS and the Xen hypervisor executing in full privileges at ring 0. In contrary, guest OSs run unprivileged at ring 1, while all processes at all domains (i.e., virtual machines) run unprivileged at ring 3, assuming a system with four rings (e.g., Intel-32). On systems with only two levels of privileges, the hypervisor and the host OS can execute in system mode, while domains and processes can execute in user mode. As illustrated in the figure, Xen eliminates the device drivers entirely from guest OSs and provides a direct communication between guest OSs at domain U_i and the host OS at domain 0 [21]. More precisely, every domain U_i in Xen will exclude virtual I/O devices and relevant drivers. I/O requests will accordingly be transferred directly to domain 0, which by default hosts all the required device drivers necessary to satisfy any I/O request. For instance, rather than using a device driver to control a virtual Network Card Interface (vNIC), with Xen, network frames/packets are transmitted back and forth via event channels between domain U_i and domain 0, using NIC front-end and back-end interfaces, respectively (see Fig. 1.24). Likewise, no virtual disk is exposed to any guest OS and all disk data blocks incurred by file reads and writes are delegated by the Xen hypervisor to domain 0.

1.8.4 A Taxonomy of Virtualization Suites

To this end, we briefly survey some of the current and common virtualization software **suites**. We distinguish between virtualization suites and hypervisors; many vendors often use these terms interchangeably. As discussed throughout this chapter, a hypervisor is mainly responsible for running multiple virtual machines (VMs) on a single physical host. On the other hand, a virtualization suite comprises of various software components and individual hypervisors that enable the management of many physical hosts and VMs. A management component typically issues commands to the hypervisor to create, destroy, manage and migrate VMs across multiple physical hosts. Table 1.2 shows our taxonomy of four virtualization suites, vSphere 5.1 [59], Hyper-V [43], XenServer 6 [17], and RHEV 3 [50]. We compare the suites in terms of multiple features including the involved hypervisor, the virtualization type, the allowable maximum number of vCPUs per a VM, the allowable maximum memory size per a VM, and whether memory over-commitment, page sharing, and live migration are supported. In addition, we indicate whether the involved hypervisors contain device drivers, and list some of the popular cloud vendors that utilize such hypervisors. To elaborate on some of the features, live migration allows VMs to be seamlessly shifted from one VM to another. It enables many management features like maintenance, power-efficient dynamic server consolidation, and workload balancing, among others. Page Sharing refers to sharing identical memory pages across VMs. This renders effective when VMs use similar OS instances. Lastly, some hypervisors eliminate entirely device drivers at guest OSs and provide direct communications between guest OSs and host OSs co-located with hypervisors (similar to what we discussed in Section 1.8.3 about the Xen hypervisor).

Table 1.2: A Comparison Between Different Virtualization Suites.

Feature	vSphere 5.1	Hyper-V 2012	XenServer 6	RHEV 3
Hypervisor Name	ESXi	Hyper-V	Xen	KVM
CPU Virtualization Support	Full Virtualization	Para-virtualization	Para-virtualization	Full Virtualization
Maximum vCPUs Per VM	160	320	64	160
Maximum Memory Per VM	1TB	1TB	128GB	2TB
Memory Over-Commitment Support	Yes	Yes	Yes	Yes
Page Sharing Support	Yes	No	No	No
Live Migration Support	Yes	Yes	Yes	Yes
Contains Device Drivers	Yes	No	No	Yes
Common Cloud Vendor(s)	vCloud Hybrid Service [58]	Microsoft Azure [41]	Amazon EC2 [3] and Rackspace [49]	IBM SmartCloud [29]

1.9 Case Study: Amazon Elastic Compute Cloud

Amazon Elastic Compute Cloud (Amazon EC2) is a vital part of Amazon's cloud computing platform, Amazon Web Services (AWS). In August 25, 2006 Amazon launched EC2, which together with Amazon Simple Storage Service (Amazon S3), marked the change in the way IT was done. Amazon EC2 is a highly reliable and scalable Infrastructure as a Service (IaaS) with a utility payment model. It allows users to rent virtual machines (VMs) and pay for the resources that they actually consume. Users can set up and configure everything in their VMs, ranging from the operating system up to any application. Specifically, a user can boot an Amazon Machine Image (AMI) to create a VM, referred in Amazon's parlance as an *instance*. AMI is a virtual appliance (or a VM image) that contains user's operating system, applications, libraries, data, and associated configuration settings.

Users can create EC2 instances either via using default AMIs pre-packaged by Amazon or via developing their own AMIs using Amazon's bundling tools. Default AMIs are preconfigured with an ever-growing list of operating systems including Red Hat Enterprise Linux, Windows Server and Ubuntu, among others. A wide selection of free software provided by Amazon can also be directly incorporated within AMIs and executed over EC2 instances. For example, Amazon provides software for databases (e.g., Microsoft SQL [44]), application servers (e.g., Tomcat Java Web Application [7]), content management (e.g., MediaWiki [40]), and business intelligence (e.g., Jasper Reports [30]). Added to the wide assortment of free software, Amazon services (e.g., Amazon Relational Database Service, which supports MySQL [45], Oracle [46] and Microsoft SQL databases) can be further employed in conjunction with EC2 instances. Finally, users can always configure, install and run at any time any compatible software on EC2 instances, exactly as is the case with regular physical machines. We next discuss the virtualization technology that underlies Amazon EC2.

1.9.1 Amazon EC2 Virtualization Technology

Amazon EC2 demonstrates the power of cloud computing. Under the hood, it is a marvel of technology. As of March 2012, it presumably hosts around 7,100 server racks with a total of 454,400 blade servers, assuming 64 blade servers per rack [38]. Above its datacenters, Amazon EC2 presents a true virtualization environment using the Xen hypervisor. Xen is a leading example of system virtualization, initially developed as part of the Xenoserver project at the Computer Laboratory, Cambridge University [18]. Currently, Xen is maintained by an open source community [63]. The goal of Xen is to provide IaaS with full isolation and minimal performance overhead on conventional hardware. As discussed previously in the chapter, Xen is a native hypervisor that runs on bare metal at the most privileged CPU state. Amazon EC2 utilizes a *highly customized version of Xen* [6], supposedly to provision and isolate user instances rapidly, consolidate instances so as to improve system utilization, tolerate software and hardware failures by saving and migrating instances, and apply system load balancing through live and seamless migration of instances, to mention a few.

Amazon EC2 instances can be created, launched, stopped and terminated as needed. Such instances are system VMs composed of virtualized (or paravirtualized) sets of physical resources including CPU, memory and I/O components. To create instances, Xen starts a highly privileged instance (*domain 0*) at a host OS out of which other user instances (*domain* U_i) can be instantiated with guest OSs (see Fig. 1.24). As host OSs, Novell's SUSE Linux Enterprise Server, Solaris and OpenSolaris, NetBSD, Debian, and Ubuntu, among others, can be used. It is not known, however, which among these host OSs Amazon EC2 supports. On the other hand, Linux, Solaris and OpenSolaris, FreeBSD and NetBSD can be employed as guest OSs, to mention a few. Among the guest OSs that Amazon EC2 supports are Linux, OpenSolaris, and Windows Server 2003 [66]. Amazon EC2's guest OSs are run at a lesser privileged ring as opposed to the host OS and the hypervisor. Clearly, this helps isolating the hypervisor from guest OSs and guest OSs from each other, a key requirement on Amazon AWS's cloud platform. Nonetheless, running guest OSs in unprivileged mode violates the usual assumption that OSs must run in system mode. To circumvent consequent ramifications, Xen applies a paravirtualized approach, where guest OSs are modified to run at a downgraded privileged level. As a result, sensitive instructions are enforced to trap to the hypervisor for verification and execution. Linux instances (and most likely OpenSolaris) on Amazon EC2 use Xen's paravirtualized mode, and it is conjectured also that Windows instances do so [14, 66]. Upon provisioning instances, Xen provides each instance with its own vCPU(s) and associated ISA. As discussed in Section 1.6, the complexity of this step depends entirely on the architecture of the underlying pCPU(s). To this end, it is not clear what vCPU scheduler Amazon EC2 applies, but Xen's default scheduler is the Credit Scheduler (see Section 1.6.5). It is suspected though that Amazon EC2 has a modified version of the Credit Scheduler [66].

Memory and I/O resources are virtualized in Xen in a way similar to what is described in Sections 1.7.2 and 1.8.3, respectively. First, Xen uses a two-level page mapping method. Second, the hardware page tables are allocated and managed by guest OSs, with a minimal involvement from the hypervisor [9]. Specifically, guest OSs can read directly from hardware page tables, but writes are intercepted and validated by the Xen hypervisor to ensure safety and isolation. For performance reasons, however, guest OSs can batch write requests to amortize the overhead of passing by the hypervisor per every write request. Finally, with Xen, existing hardware I/O devices are not emulated as is typically done in fully-virtualized environments. In contrast, I/O requests are always transferred from user instances to domain 0 and vice versa, using a shared-memory communication paradigm as demonstrated in Fig. 1.24. At domain 0, device drivers of the host OS are borrowed to handle the I/O requests.

1.9.2 Amazon EC2 Properties

Amazon EC2 is characterized with multiple effective properties, some that are stemmed naturally from its underlying virtualization technology and others that are employed specifically for the AWS cloud computing platform to meet certain performance, scalability, flexibility, security and reliability criteria. We next concisely discuss some of these properties.

1.9.2.1 Elasticity

In leveraging a major benefit offered by virtualization, Amazon EC2 allows users to statically and dynamically scale up and down their EC2 clusters. In particular, users can always provision and de-provision virtual EC2 instances by manually starting and stopping any number of them using the AWS management console, the Amazon command line tools and/or the Amazon EC2 API. Besides, users can employ Amazon's CloudWatch to monitor EC2 instances in real-time and automatically respond to changes in computing requirements. CloudWatch is an Amazon service that allows users to collect statistics about their cluster resource utilization, operational performance, and overall resource demand patterns. Metrics such as CPU utilization, disk operations, and network traffic can be aggregated and fed to the Amazon's Auto Scaling process enabled by CloudWatch. The Auto Scaling process can subsequently add or remove instances so as performance is maintained and costs are saved. In essence, Auto Scaling allows users to closely follow the demand curve of their applications and synergistically alter their EC2 clusters according to conditions they define (e.g., add 3 more instances to the cluster when the average CPU utilization exceeds 80%).

1.9.2.2 Scalability and Performance

Table 1.3: Amazon EC2 Instance Types as of March 4, 2013.

Instance Type	Instance Name	CPU Capacity	Memory Size	Storage Size and Type	Platform
Standard	M1 Small	1 vCPU with 1 ECU	1.7GB	160GB local storage	32-bit or 64-bit
	M1 Medium	1 vCPU with 2 ECUs	3.75GB	410GB local storage	32-bit or 64-bit
	M1 Large	2 vCPUs, each with 2 ECUs	7.5GB	850GB local storage	64-bit
	M1 Extra Large	4 vCPUs, each with 2 ECUs	15GB	1690 GB local storage	64-bit
	M3 Extra Large	4 vCPUs, each with 3.25 ECUs	15GB	EBS storage only	64-bit
	M3 Double Extra Large	8 vCPUs, each with 3.25 ECUs	30GB	EBS storage only	64-bit
Micro	Micro	Up to 2 ECUs	613MB	EBS storage only	32-bit or 64-bit
High-Memory	Extra Large	2 vCPUs, each with 3.25 ECUs	17.1GB	420GB local storage	64-bit
	Double Extra Large	4 vCPUs, each with 3.25 ECUs	34.2GB	850GB local storage	64-bit
	Quadruple Extra Large	8 vCPUs, each with 3.25 ECUs	68.4GB	1690GB local storage	64-bit
High-CPU	Medium	2 vCPUs, each with 2.5 ECUs	1.7GB	350GB local storage	32-bit or 64-bit
	Extra Large	8 vCPUs, each with 2.5 ECUs	7GB	1690GB local storage	64-bit
Cluster Compute	Eight Extra Large	88 ECUs	60.5GB	3370GB local storage	64-bit & 10 Gigabit Ethernet
High Memory Cluster	Eight Extra Large	88 ECUs	244GB	240GB local storage	64-bit & 10 Gigabit Ethernet
Cluster GPU	Quadruple Extra Large	33.5 ECUs 2 $\times$ NVIDIA Tesla Fermi M2050 GPUs	22GB	1690GB local storage	64-bit & 10 Gigabit Ethernet
High I/O	Quadruple Extra Large	35 ECUs	60.5GB	2 $\times$ 1024GB SSD-based local storage	64-bit & 10 Gigabit Ethernet
High Storage	Eight Extra Large	35 ECUs	117GB	24 $\times$ 2TB hard disk drive local storage	64-bit & 10 Gigabit Ethernet

Amazon EC2 instances can scale to more than 255 pCPUs per host [65], 128 vCPUs per guest, 1TB of RAM per host, up to 1TB of RAM per unmodified guest and 512GB of RAM per paravirtualized guest [64]. In addition, Amazon EC2 reduces the time needed to boot a fresh instance to seconds, thus expediting scalability as the needs for computing varies. To optimize performance, Amazon EC2 instances are provided in various resource capacities that can suit different application types, including CPU-intensive, memory-intensive and I/O-intensive applications (see Table 1.3). The vCPU capacity of an instance is expressed in terms of Elastic Compute Units (ECU). Amazon EC2 uses ECU as an abstraction of vCPU capacity whereby one ECU provides the equivalence of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor [3]. Different instances with different ECUs provide different application runtimes. The performances of instances with identical type and ECUs may also vary as a result of what is denoted in cloud computing as performance variation [22, 51].

1.9.2.3 Flexibility

Amazon EC2 users are provided with complete control over EC2 instances, with a root access to each instance. They can create AMIs with software of their choice and apply many of Amazon services, including Amazon Simple Storage Service (Amazon S3), Amazon Relational Database Service (Amazon RDS), Amazon SimpleDB and Amazon Simple Queue Service (Amazon SQS), among others. These services and the various available Amazon EC2 instance types can jointly deliver effective solutions for computing, query processing and storage across a wide range of applications. For example, users running I/O-intensive applications like data warehousing and Hadoop MapReduce [19, 28] can exploit High Storage instances. On the other hand, for tightly

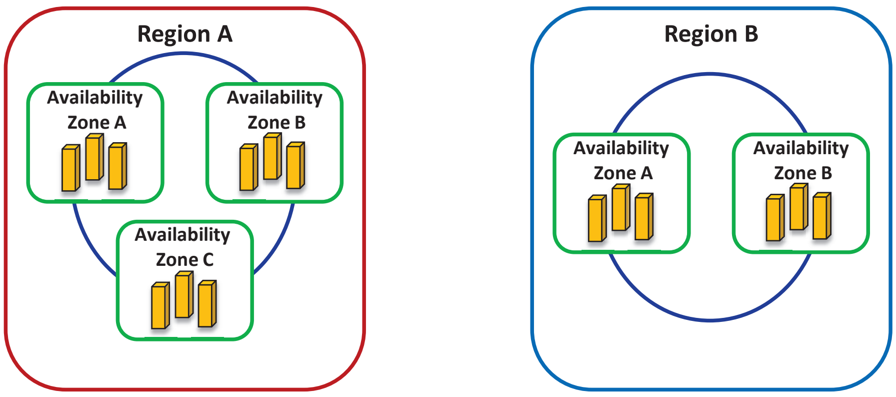


Figure 1.25: Regions and Availability Zones in AWS cloud platform. Regions are geographically dispersed to avoid disasters. Availability Zones are engineered as autonomous failure zones within Regions.

coupled, network-intensive applications users can utilize High Performance Computing (HPC) clusters.

In addition, users have the flexibility to choose among multiple storage types that can be associated with their EC2 instances. First, users can rent EC2 instances with *local instance-store disks* as root devices. Instance-store volumes are volatile storages and cannot survive stops and terminations. Second, EC2 instances can be attached to Elastic Block Storages (EBS), which provide raw block devices. The block devices can then be formatted and mounted with any file system at EC2 instances. EBS volumes are persistent storages and can survive stops and terminations. EBS volumes of sizes from 1GB to 1TB can be defined and RAID arrays can be created by combining two or more volumes. EBS volumes can even be attached or detached from instances while they are running. They can also be moved from one instance to another, thus rendering independent of any instance. Finally, applications running on EC2 instances can access Amazon S3 through a defined API. Amazon S3 is a storage that makes web-scale computing easier for developers, whereby any amount of data can be stored and retrieved at any time, and from anywhere on the web [5].

To this end, Amazon EC2 users do not only have the flexibility of choosing among many instance and storage types, but further have the capability of mapping elastic IP addresses to EC2 instances, without a network administrator's help or the need to wait for DNS to propagate new bindings. Elastic IP addresses are static IP addresses, but tailored for the dynamicity of the cloud. For example, unlike a traditional static IP address, an Elastic IP address enables tolerating an instance failure by programmatically remapping the address to any other healthy instance under the same user account. Thus, Elastic IP addresses are associated with user accounts and not EC2 instances. Elastic IP addresses exist until explicitly removed, and persist even while accounts have no current running instances.

1.9.2.4 Fault Tolerance

Amazon EC2 users are capable of placing instances and storing data at multiple locations represented as *Regions* and *Availability Zones*. As shown in Fig. 1.25, a Region can consist of one or many Availability Zones and an Availability Zone can consist of typically many blade servers. Regions are independent collections of AWS resources that are geographically dispersed to avoid catastrophic disasters. An Availability Zone is a distinct location in a Region designed to act as an autonomous failure zone. Specifically, an Availability Zone does not share a physical infrastructure with other Availability Zones, thus limiting failures from transcending its own boundaries. Furthermore, when a failure occurs, automated AWS processes start moving customer application traffic away from the affected zone [6]. Consequently, applications that run in more than one Availability Zone across Regions can inherently achieve higher availability and minimize downtime. Amazon EC2 guarantees 99.95% availability per each Region [3].

Lastly, EC2 instances that are attached to Amazon EBS volumes can attain improved durability over EC2 instances with local stores (or the so-called ephemeral storage). Amazon EBS volumes are automatically replicated in the backend of a single Availability Zone. Moreover, with Amazon EBS, point-in-time consistent snapshots of EBS volumes can be created and reserved in Amazon S3. Amazon S3 storage is automatically replicated across multiple Availability Zones and not only in a single Availability Zone. Amazon S3 helps maintain the durability of users' data by quickly detecting and repairing losses. Amazon S3 is designed to provide 99.999999999% durability and 99.99% availability of data over a given year [6]. A snapshot of an EBS volume can also serve as the starting point for a new EBS volume in case the current one fails. Therefore, with the availability of Regions and Availability Zones, the virtualized environment provided by Xen, and the Amazon's EBS and S3 services, Amazon EC2 users can achieve long term protection, failure isolation and reliability.

1.9.2.5 Security

Security within Amazon EC2 is provided at multiple levels. First, as pointed out earlier, EC2 instances are completely controlled by users. Users have full root access or administrative control over their instances, accounts, services and applications. AWS does not have any access rights to user instances and cannot log into their guest OSs [6]. Second, Amazon EC2 provides a complete firewall solution, whereby the default state is to deny all incoming traffic to any user instance. Users must explicitly open ports for specific inbound traffic. Third, API calls to start/stop/terminate instances, alter firewall configurations and perform other related functions are all signed by the user's Amazon Secret Access Key. Without the Amazon Secret Access Key, API calls on Amazon EC2 instances cannot be made. Fourth, the virtualized environment provided by Xen provides a clear security separation between EC2 instances and the hypervisor as they run at different privileged modes. Fifth, the AWS firewall is placed within the hypervisor, between the physical network interface and the virtual interfaces of instances. Hence, as packet requests are all privileged, they must trap to the hypervisor and accordingly pass through the AWS firewall. Consequently, any two communicating instances will be treated as separate virtual machines on the Internet, even if they are placed on the same physical machine. Finally, as Amazon EBS volumes can be associated with EC2 instances, their accesses are restricted to the AWS accounts that created the volumes. This indirectly denies all other AWS accounts (and corresponding users) from viewing and accessing the volumes. We note, however, that this does not impact the flexibility of sharing data on the AWS cloud platform. In particular, users can still create Amazon S3 snapshots of their Amazon EBS volumes and share them with other AWS accounts/users. Nevertheless, only the users who own the volumes will be allowed to delete or alter EBS snapshots.

1.10 Summary

In this chapter, we first motivated the case of why virtualization is important in cloud computing. We discussed various key properties required by the cloud and offered by virtualization such as elasticity, resource sandboxing, enhanced system utilization, reduced costs, optimized energy consumption, facilitated Big Data analytics, and mixed-OS environment. Next we discussed the limitations of general-purpose OSs for acting as sole enablers for cloud computing. Specifically, we described OS limitations pertaining to flexibility, fault isolation, resource isolation and security isolation. Before delving into further details about virtualization, we introduced two major strategies for managing complexity in computer systems (which are necessary for virtualizing cloud resources), abstracting the system stack and specifying well-defined interfaces. Afterwards we formally defined virtualization and presented two main implementations of virtual machines (VMs), process VMs and system VMs.

To this end, we started describing how various physical components can be virtualized and encompassed within VMs. We began with the CPU component whereby we discussed the conditions for virtualizing CPUs and investigated different techniques for virtualizing CPUs such as paravirtualization and emulation. We closed the discussion on CPU virtualization with a real example from Xen, a popular virtualization hypervisor employed by Amazon's cloud computing platform, Amazon Web Services (AWS). Subsequently, we introduced system memory virtualization and detailed how virtual memory in the contexts of OSs and hypervisors can be constructed. In particular, we examined the one-level and the two-level page mapping techniques for creating virtual memories in OSs and hypervisors, respectively. As a side effect of virtual memory, we identified the memory over-commitment problem and presented a solution from VMWare

ESX, the memory ballooning reclamation technique. Next we explored I/O virtualization. We commenced with a brief introduction on some I/O basics and debated on the pros and cons of virtualizing I/O operations at the system call, device driver, and operation level interfaces. As a practical example, Xen's approach to I/O virtualization was described. Finally, we wrapped up our discussion on virtualizing resources for the cloud with a case study, Amazon Elastic Compute Cloud (Amazon EC2), a vital part of Amazon's AWS. Specifically, we discussed the underlying virtualization technology of Amazon EC2 and some of its major characteristics including elasticity, scalability, performance, flexibility, fault tolerance and security.

References

[1] ”A History of Cloud Computing, http://www.cloudtweaks.com/2011/02/a-history-of-cloud-computing/.”

[2] ”Amazon Auto Scaling, http://aws.amazon.com/autoscaling/.”

[3] ”Amazon Elastic Compute Cloud, http://aws.amazon.com/ec2/.”

[4] ”Amazon Elastic MapReduce, http://aws.amazon.com/elasticmapreduce/.”

[5] ”Amazon Simple Storage Service, http://aws.amazon.com/s3/.”

[6] Amazon, ”Amazon Web Services: Overview of Security Processes,” *Amazon Whitepaper,* May 2011.

[7] ”Apache Tomcat, http://tomcat.apache.org/.”

[8] M. Bailey, ”The Economics of Virtualization: Moving Toward an Application-Based Cost Model,” *VMware Sponsored Whitepaper,* 2009.

[9] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt and A. Warfield, ”Xen and the Art of Virtualization,” *SOSP,* Oct. 2003.

[10] J. R. Bell, ”Threaded Code,” *Communications of the ACM,* 1973.

[11] A. Beloglazov and R. Buyya, ”Energy Efficient Allocation of Virtual Machines in Cloud Data Centers,” *CCGrid,* May 2010.

[12] A. Beloglazov, J. Abawajy and R. Buyya, ”Energy-Aware Resource Allocation Heuristics for Efficient Management of Data Centers for Cloud Computing,” *Future Generation Computer Systems,* 2012.

[13] A. Berl, E. Gelenbe, M. Di Girolamo, G. Giuliani, H. De Meer, M. Q. Dang and K. Pentikousis, ”Energy-Efficient Cloud Computing,” *The Computer Journal,* 2010.

[14] C. Boulton, ”Novell, Microsoft Outline Virtual Collaboration,” *Serverwatch,* 2007.

[15] P. M. Chen and B. D. Nobel, ”When Virtual Is Better Than Real,” *HOTOS,* May 2001.

[16] D. Chisnall, ”The Definitive Guide to the Xen Hypervisor,” *Prentice Hall, 1st Edition* Nov. 2007.

[17] ”Citrix XenServer 6 Platinum Edition, http://support.citrix.com/product/xens/v6.0/.”

[18] G. Coulouris, J. Dollimore, T. Kindberg and G. Blair, ”Distributed Systems: Concepts and Design,” *Addison-Wesley, 5 Edition* May 2011.

[19] J. Dean and S. Ghemawat, ”MapReduce: Simplified Data Processing On Large Clusters,” *OSDI,* Dec. 2004.

[20] R. B. K. Dewar, ”Indirect Threaded Code,” *Communications of the ACM,* June 1975.

[21] T. W. Doeppner, ”Operating Systems In Depth: Design and Programming,” *Wiley, 1st Edition* Nov. 2010.

[22] B. Farley, V. Varadarajan, K. Bowers, A. Juels, T. Ristenpart and M. Swift ”More for Your Money: Exploiting Performance Heterogeneity in Public Clouds,” *SOCC,* 2012.

[23] J. Gantz and D. Reinsel, ”The Digital Universe in 2020,” *IDC Whitepaper,* 2012.

[24] S. Ghemawat, H. Gobioff, and S. T. Leung, ”The Google File System,” *SOSP,* Oct. 2003.

[25] ”Google Apps. https://developers.google.com/google-apps/.”

[26] ”Google App Engine. https://developers.google.com/appengine/.”

[27] M. Gschwind, E. R. Altman, S. Sathaye, P. Ledak and D. Appenzeller, ”Dynamic and Transparent Binary Translation,” *IEEE Computer,* 2000.

[28] ”Hadoop, http://hadoop.apache.org/.”

[29] ”IBM SmartCloud, http://www.ibm.com/cloud-computing/us/en/.”

[30] ”Jasper Reports, http://community.jaspersoft.com/project/jasperreports-library.”

[31] Y. Jin, Y. Wen and Q. Chen, ”Energy Efficiency and Server Virtualization in Data Centers: An Empirical Investigation,” *Computer Communications Workshops (INFOCOM WKSHPS),* March 2012.

[32] P. Klint, ”Interpretation Techniques,” *Software Practice and Experience,* 1981.

[33] D. Kusic, J. O. Kephart, J. E. Hanson, N. Kandasamy, and G. Jiang, ”Power and Performance Management of Virtualized Computing Environments via Lookahead Control,” *Cluster Computing,* 2009.

[34] P. M. Kogge, ”An Architecture Trail to Threaded-Code Systems,” *IEEE Computer,* March 1982.

[35] ”Large Hadron Collide, http://www.lhc.ac.uk/.”

[36] ”Learn About Java Technology, http://www.java.com/en/about/.”

[37] P. Li, ”Cloud Computing: Big Data is the Future of IT,” *http://assets.accel.com/5174affa160bd_cloud_computing_big_data.pdf* 2009.

[38] H. Liu, ”Amazon Data Center Size,” *http://huanliu.wordpress.com/2012/03/13/amazon-data-center-size/* March 2012.

[39] P. Magnusson and D. Samuelsson, ”A Compact Intermediate Format for SIMICS,” *Swedish Institute of Computer Science, Tech. Repoart R94:17* Sep. 1994.

[40] ”MediaWiki, http://www.mediawiki.org/wiki/MediaWiki.”

[41] ”Microsoft Azure, http://www.windowsazure.com/en-us/.”

[42] Microsoft Corporation, ”ECMA C# and Common Language Infrastructure Standards,” Oct. 2009.

[43] ”Microsoft Hyper-V Server 2012, http://www.microsoft.com/en-us/server-cloud/hyper-v-server/default.aspx.”

[44] ”Microsoft SQL Server, http://www.microsoft.com/en-us/sqlserver/default.aspx.”

[45] ”MySQL, http://www.mysql.com/.”

[46] ”Oracle SQL, http://www.oracle.com/technetwork/developer-tools/sql-developer/overview/index.html.”

[47] I. Pratt, K. Fraser, S. Hand, Limpach, A. Warfield, Magenheimer, Nakajima and Mallick, ”Xen 3.0 and the Art of Virtualization,” *Proceedings of the Linux Symposium (Volume Two),* July 2005.

[48] G. J. Popek and R. P. Goldberg, ”Formal Requirements for Virtualizable Third Generation Architectures,” *Communications of the ACM,* July 1974.

[49] ”Rackspace, http://www.rackspace.com/.”

[50] ”Red Hat Enterprise Virtualization (RHEV) 3 for Servers, http://www.redhat.com/promo/rhev3/.”

[51] M. S. Rehman and M. F. Sakr, ”Initial Findings for Provisioning Variation in Cloud Computing,” *CloudCom,* Nov. 2010.

[52] T. H. Romer, D. Lee, G. M. Voelker, A. Wolman, W. A. Wong, J.-L. Baer, B. N. Bershad, and H. M. Levy, ”The Structure and Performance of Interpreters,” *ASPLOS,* 1996.

[53] Silicon Valley Leadership Group, ”Accenture, Data Centre Energy Forecast Report,” *Final Report,* July, 2008.

[54] R. L. Sites, A. Chernoff, M. B. Kirk, M. P. Marks, and S. G. Robinson, ”Binary Translation,” *Communications of the ACM,* Feb. 1993.

[55] J. E. Smith and R. Nair, ”Virtual Machines: Versatile Platforms for Systems and Processes,” *Morgan Kaufmann,* 2005.

[56] S. Soltesz, H. Potz, M. E. Fiuczynski, A. Bavier and L. Peterson, ”Container-Based Operating System Virtualization: a Scalable, High-Performance Alternative to Hypervisors,” *EuroSys,* March 2007.

[57] S. Srikantaiah, A. Kansal, and F. Zhao, ”Energy Aware Consolidation for Cloud Computing,” *Cluster Computing,* 2009.

[58] ”vCloud Hybrid Service, http://www.vmware.com/products/datacenter-virtualization/vcloud-hybrid-service/overview.html.”

[59] ”VMWare, http://www.vmware.com.”

[60] VMWare, ”Understanding Full Virtualization, Paravirtualization, and Hardware Assist,” *VMware Whitepaper,* Nov. 2007.

[61] VMWare, ”VMware vSphere: The CPU Scheduler in VMware ESX 4.1,” *VMware Whitepaper,* 2010.

[62] VMWare, ”Understanding Memory Resource Management in VMware ESX Server,” *VMware Whitepaper,* 2009.

[63] ”Xen Open Source Community, http://www.xen.org.”

[64] ”Xen 4.0 Release Notes, http://wiki.xen.org/wiki/Xen_4.0_Release_Notes.”

[65] ”Xen 4.1 Release Notes, http://wiki.xen.org/wiki/Xen_4.1_Release_Notes.”

[66] F. Zhou, M. Goel, P. Desnoyers and R. Sundaram, ”Scheduler Vulnerabilities and Attacks in Cloud Computing,” *arXiv:1103.0759v1 [cs.DC],* March 2011.