CHAPTER

Cloud resource virtualization

5

Virtualization abstracts the underlying physical resources of computer and communication systems and simplifies their use, isolates users from one another, and supports replication, which, in turn, increases system elasticity and reliability. Resource virtualization, the technique analyzed in this chapter, is a ubiquitous alternative to traditional data center operation. Virtualization, a basic tenet of cloud computing, simplifies some of the resource management tasks. For example, a *virtual machine* (VM) running under a hypervisor can be migrated to another server to balance the load and save energy. At the same time, virtualization allows users to operate in environments they are familiar with, rather than forcing them to work in idiosyncratic environments.

Interpreters, memory, and communication channels are fundamental abstractions necessary to describe the operation of a computing system [430]. Their respective physical realizations are: processors to transform information; primary and secondary memory to store information; and communication systems enabling all functional units to interact with one another. Processors, memory, and communication systems have different bandwidths, latencies, reliabilities, and other physical characteristics.

System software transforms physical implementations of the three abstractions into computer systems able to run diverse applications and manages all system resources. The traditional modus operandi of a data center is to install an operating system (OS) on individual computers and rely on conventional OS techniques to ensure resource sharing, application protection, and performance isolation.

Cloud service providers (CSPs), as well as cloud users, face multiple challenges in such a setup. CSPs are under pressure to optimize accounting, security, and system resource management. Users are under pressure to develop and optimize their application performance for one system and have to start over again when the application is moved to another data center with different hardware, system software, and libraries. Virtualization alleviates these problems for CSPs and cloud users, but at a price.

Resource sharing in a VM environment requires not only ample hardware support and, in particular, powerful processors and fast interconnects but also architectural support for multilevel control because resource sharing occurs at multiple levels. Resources, such as CPU cycles, memory, secondary storage, and I/O and communication bandwidth, are shared among several VMs. VM resources are shared among processes or threads of an application.

This chapter starts with a discussion of virtualization principles and the motivation for virtualization in Section 5.1. Section 5.2 is focused on performance and security isolation. Alternatives for the implementation of virtualization are analyzed in Section 5.3. Two distinct approaches to processor virtualization, *full virtualization* and *paravirtualization,* are presented in Section 5.4. Full virtualization is feasible when the hardware abstraction provided by the hypervisor is an exact replica of the physical hardware. In this case any operating system running on the hardware will run without modifications under the hypervisor. Paravirtualization requires modifications of the guest operating systems because the hardware abstraction provided by the hypervisor does not support all hardware operations.

Cloud Computing. https://doi.org/10.1016/B978-0-32-385277-7.00012-9

Traditional processor architectures were conceived for one level of control as they support two execution modes, the kernel and the user mode. In a virtualized environment, all resources are under the control of a hypervisor, and a second level of control is exercised by the guest OS. While a two-level scheduling for sharing CPU cycles can be easily implemented, sharing of resources such as cache, memory, and I/O bandwidth is more intricate. In 2005 and 2006, the x86 processor architecture was extended to provide hardware support for virtualization, as discussed in Section 5.5. Nested virtualization allows hypervisors to run inside a VM, complicating even further the virtualization landscape.

Section 5.6 covers an open-source emulator and virtualizer. Several hypervisors are used nowadays. Xen and an optimization of its network performance are presented in Sections 5.8 and 5.9. KVM, a virtualization infrastructure of the Linux kernel, is discussed in Section 5.7. Nested virtualization is analyzed in Section 5.10, followed by the presentation of a trusted kernel virtualization in Section 5.11.

High-performance processors have multiple functional units, but do not provide explicit support for virtualization, as discussed in Section 5.12 which covers Itanium paravirtualization. System functions critical for the performance of a VM environment are cache and memory management, handling of privileged instructions, and I/O handling.

Cache misses are an important source of performance degradation in a VM environment as we shall see in Section 5.13. An overview of open-source software platforms for virtualization is presented in Section 5.14. The potential risks of virtualization are the subject of Section 5.15, and virtualization software is discussed in Section 5.16.

5.1 Resource virtualization

Virtualization has been used successfully since the late 1950s; a virtual memory based on paging was first implemented on the Atlas computer at the University of Manchester in the United Kingdom in 1959.

Virtualization simulates the interface with a physical object by any one of four means [430]:

1. *Multiplexing:* create multiple virtual objects from one instance of a physical object. For example, a processor is multiplexed among a number of processes or threads.
2. *Aggregation:* Create one virtual object from multiple physical objects. For example, a number of physical disks are aggregated into a RAID disk.
3. *Emulation:* Construct a virtual object from a different type of a physical object. For example, a physical disk emulates a Random Access Memory.
4. *Multiplexing and emulation.* Examples are: virtual memory with paging multiplexes main memory and secondary storage; a virtual address emulates a real address; the TCP protocol emulates a reliable bit pipe and multiplexes a physical communication channel and a processor.

Virtualization is a critical aspect of cloud computing, equally important for the providers and the consumers of cloud services, and plays an important role for: (i) system security because it enables isolation of services running on the same hardware; (ii) performance and reliability because it enables applications to migrate from one platform to another; (iii) the development and management of services offered by a provider; and (iv) performance isolation.

In a cloud computing environment, a hypervisor or virtual-machine monitor runs on the physical hardware and exports hardware-level abstractions to one or more guest operating systems. A guest OS

interacts with the virtual hardware in the same manner it would interact with the physical hardware, but under the watchful eye of the hypervisor that traps all privileged operations and mediates the interactions of the guest OS with the hardware. For example, a hypervisor would control I/O operations for two virtual disks implemented as two different sets of tracks on a physical disk. New services can be added without the need to modify an operating system.

User convenience is a necessary condition for the success of the utility computing paradigm; one of the multiple facets of user convenience is the ability to run remotely using the system software and libraries required by the application. User convenience is a major advantage of a VM architecture versus a traditional operating system. For example, an AWS user could submit an Amazon Machine Image (AMI) containing the applications, libraries, data, and the associated configuration settings; a user could choose the operating system for the application, then start, terminate, and monitor as many instances of AMI as needed, using web service APIs and performance monitoring and management tools provided by AWS.

There are side effects of virtualization, notably the *performance penalty* and the *hardware costs*. All privileged operations of a VM must be trapped and validated by the hypervisor, which ultimately controls the system behavior. The increased overhead introduced by the hypervisor has a negative impact on the performance.

The cost of a system running multiple VMs is higher than the cost of a system running a traditional OS. In the former case, the physical hardware is shared among a set of guest operating systems, and it is typically configured with faster and/or multicore processors, more memory, larger disks, and additional network interfaces as compared to a system running a traditional operating system.

5.2 Performance and security isolation in computer clouds

To exhibit predictable performance an application must be isolated from applications it shares resources with. *Performance isolation* is a critical condition for QoS guarantees in cloud computing where all system resources are shared. If the run-time behavior of an application is affected by other applications running concurrently and competing for CPU cycles, cache, main memory, disk, and network access, it is rather difficult to predict application completion time and to optimize its performance. Performance unpredictability is a "deadly sin" for real-time operation and for embedded systems.

Several operating systems including Linux/RK [369], QLinux [466], and SILK [54] support some performance isolation. In spite of the efforts to support performance isolation, interactions among applications sharing the same physical system, often described as *QoS crosstalk*, still exist [473]. Accounting for consumed resources and the overhead of system activities, including context switching and paging to individual users, is challenging.

Processor virtualization presents multiple copies of the same physical processor or core to applications. Processor virtualization is different from *processor emulation* when the machine instructions of guest system are "emulated" in software running on the host system. Emulation is much slower than virtualization. For example, Microsoft's VirtualPC was designed to run on x86 processor architecture. VirtualPC was running on emulated x86 hardware until Apple adopted Intel chips.

Traditional operating systems multiplex multiple processes or threads, while virtualization supported by a hypervisor multiplexes full operating systems. A hypervisor executes directly on the hardware a subset of frequently used machine instructions generated by the application and emulates

privileged instructions including device I/O requests. The subset of instructions executed directly by the hardware includes arithmetic instructions, memory access, and branching instructions. The overhead of context switching carried out by the hypervisor is larger and leads to a significant performance.

Operating systems use the process abstraction not only for resource sharing but also to support isolation. Unfortunately, this is not sufficient from a security perspective because once a process is compromised, it is rather easy for an attacker to penetrate the entire system.

Applications running under a VM can only access virtual devices emulated by the software. This layer of software has the potential to provide a level of isolation nearly equivalent to the isolation presented by two different physical systems. Therefore, virtualization can be used to improve security in a cloud computing environment.

A hypervisor is a much simpler and better specified system than a traditional OS. For example, the Xen hypervisor discussed in Section 5.8 has approximately 60 000 lines of code, while the *Denali* hypervisor [514] has only about half, i.e., 30 000 lines of code.

The security vulnerability of hypervisors is considerably reduced because the systems expose a much smaller number of privileged functions. For example, Xen can be accessed through 28 hypercalls, while a standard Linux allows hundreds, e.g., Linux 2.6.11 allows 289 system calls. A traditional OS supports special devices, e.g., */dev/kmem*, and many privileged third-party programs, e.g., *sendmail* and *sshd*, in addition to a plethora of system calls.

5.3 Virtual machines

A VM is an isolated environment with access to a subset of the physical resources of a computer system. Each VM appears to be running on the bare hardware, giving the appearance of multiple instances of the same computer, though all are supported by a single physical system. VM history can be traced back to the 1960s.[1] In the early 1970s, IBM released its widely used VM 370 system, followed in 1974 by the MVS (Multiple Virtual Storage) system.

There are two types of VMs, process and system. A *process VM* is a virtual platform created for an individual process and destroyed once the process terminates. Virtually all operating systems provide a process VM for each one of the applications running, but the more interesting process VMs are those that support binaries compiled on a different instruction set. A *system VM* supports an OS together with many user processes. When the VM runs under the control of a normal OS and provides a platform-independent host for a single application, we have an *application VM*, e.g., Java Virtual Machine (JVM).

A *system VM* provides a complete system: Each VM can run its own OS, which in turn can run multiple applications. Systems such as Linux Vserver http://linux-vserver.org, OpenVZ (Open Virtual-iZation) [376], FreeBSD Jails [415], and Solaris Zones [403] based on Linux, FreeBSD, and Solaris, respectively, implement *OS-level virtualization technologies*.

The OS-level virtualization enables a physical server to run multiple isolated OS instances subject to several constraints; the instances are known as containers, Virtual Private Servers, or Virtual Envi-

[1] In 1963, MIT announced Project MAC (Multiple Access Computer) and chose the GE-645 as the mainframe for its Multics project. IBM was GE's competitor, realized that there is a demand for such systems, and designed the CP-40 mainframe, followed by the CP-67, also called CP/CMS mainframes. CP was a program running on the mainframe used to create VMs running a single-user OS called CMS.

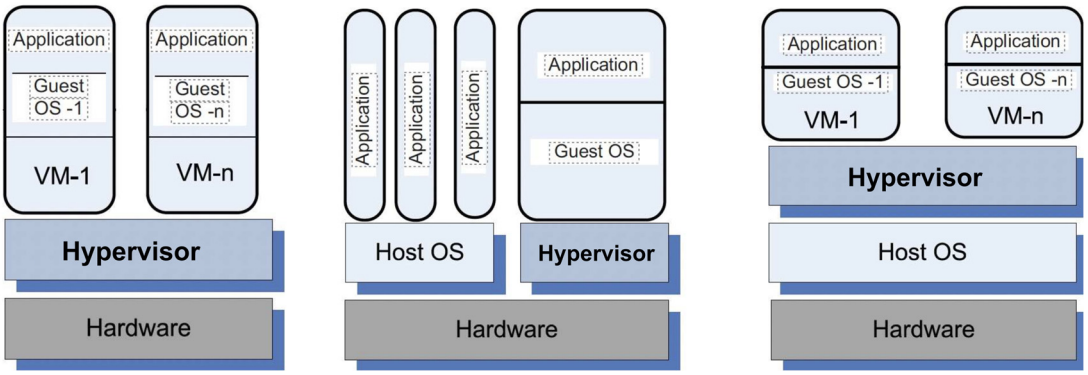

FIGURE 5.1

Hypervisor's role in traditional, hybrid, and hosted VM architectures. (Left) Traditional VMs; hypervisor supports multiple VMs and runs directly on the hardware. (Center) Hybrid VM; hypervisor shares the hardware with a host OS and supports multiple VMs. (Right) Hosted VM; hypervisor runs under a host OS.

ronments. For example, OpenVZ requires both the host and the guest OS to be Linux distributions. These systems claim performance advantages over the systems based on a hypervisor, such as Xen or VMware. According to [376], there is only a 1% to 3% performance penalty for OpenVZ as compared to a standalone Linux server. OpenVZ is licensed under the GPL version 2.

A hypervisor enables several VMs to share a physical system. Several organizations of the software stack supporting virtualization are possible:

- *Traditional—VM also called a "bare metal" hypervisor*—a thin software layer that runs directly on the host machine hardware; its main advantage is performance, Fig. 5.1(b). Examples: VMware ESX, ESXi Servers, Xen, OS370, and Denali.
- *Hybrid*—the hypervisor shares the hardware with an existing OS, Fig. 5.1(c). Example: VMWare Workstation.
- *Hosted*—the VM runs on top of an existing OS, Fig. 5.1(d); the main advantage of this approach is that the VM is easier to build and install. Another advantage is that the hypervisor could use several components of the host OS, such as the scheduler, the pager, and the I/O drivers, rather than providing its own. A price to pay for this simplicity is the increased overhead and the associated performance penalty; indeed, the I/O operations, page faults, and scheduling requests from a guest OS are not handled directly by the hypervisor, instead they are passed to the host OS. Performance, as well as the challenges to support complete isolation of VMs, make this solution less attractive for servers in a cloud computing environment. User-mode Linux is an example of a hosted VM.

Processor virtualization is not without its own challenges. As pointed out in [100] services provided by a VM "operate below the abstractions provided by the guest OS... it is difficult to provide a service that checks file system integrity without the knowledge of on-disk structure." The hypervisors discussed in Section 4.4 manage resource sharing among the VMs running on a physical system.

5.4 Full virtualization and paravirtualization

In 1974, Popek and Goldberg gave a set of sufficient conditions for a computer architecture to support virtualization and allow a hypervisor to operate efficiently. Their crisp description of these conditions in [402] is a major contribution to the field:

```
1. A program running under the hypervisor should exhibit
    a behavior essentially identical to that demonstrated
    when running on an equivalent machine directly.

2. The hypervisor should be in complete control of the
    virtualized resources.

3.  A statistically significant fraction of machine instructions
     must be executed without the intervention of the hypervisor.
```

One way to identify an architecture suitable for a VM is to distinguish *sensitive machine instructions* that require special precautions at execution time from the ones that can be executed without special precautions. In turn, sensitive instructions are: (i) *control sensitive*, i.e., instructions that attempt to change either the memory allocation, or operate in kernel mode; and (ii) *mode sensitive*, i.e., instructions whose behavior is different in kernel mode.

An equivalent formulation of the conditions for efficient virtualization can be based on this classification of machine instructions: *A hypervisor for a third or later generation computer can be constructed if the set of sensitive instructions is a subset of the privileged instructions of that machine.* To handle nonvirtualizable instructions, one could resort to two strategies:

Binary translation. The hypervisor monitors the execution of guest operating systems; nonvirtualizable instructions executed by a guest OS are replaced with other instructions.
Paravirtualization. The guest OS is modified to use only instructions that can be virtualized.

There are two basic approaches to processor virtualization, see Fig. 5.2: *full virtualization* when each VM runs on an exact copy of the actual hardware; and *paravirtualization* when each VM runs on a slightly modified copy of the actual hardware. Paravirtualization is often adopted for several reasons: (1) some aspects of the hardware cannot be virtualized; (2) has better performance; and (3) presents a simpler interface.

Full virtualization requires a virtualizable architecture. The hardware is fully exposed to the guest OS which runs unchanged, and it is necessary to ensure that this execution mode is efficient. Systems such as the VMware EX Server support full virtualization on x86 architecture and have to address several problems, including the virtualization of the MMU. Privileged x86 instructions executed by a guest OS fail silently, thus, traps must be inserted whenever privileged instructions are issued by a guest OS. The system must also maintain shadow copies of system control structures, such as page tables, and trap every event affecting the state of these control structures. Therefore, the overhead of many operations is substantial.

Computer architectures such as x86 are not easily virtualizable as we shall see in Section 5.5. Paravirtualization is the alternative, though it has its own problems. Paravirtualization requires modifi-

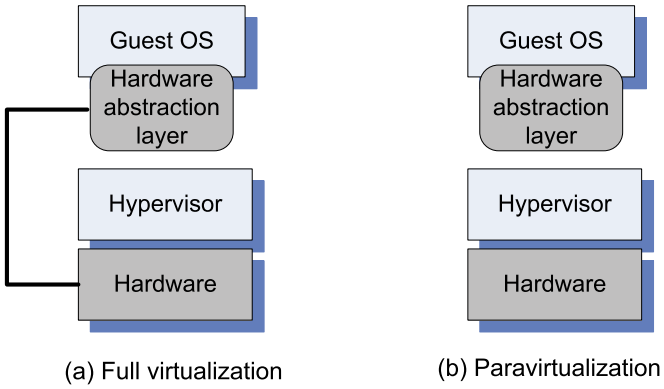

FIGURE 5.2

(a) Full virtualization requires the hardware abstraction layer of the guest OS to have some knowledge about the processor architecture. The guest OS runs unchanged, thus, this virtualization mode is more efficient than paravirtualization. (b) Paravirtualization is used when the processor architectures is not easily virtualizable. In this case the hardware abstraction layer of the guest OS does not have knowledge about the hardware. The guest OS is modified to run under the hypervisor and must be ported to individual hardware platforms.

cations to a guest OS. Moreover, the code of the guest OS has to be ported to each individual hardware platform. Xen [48] and Denali [514] are based on paravirtualization.

Generally, the virtualization overhead negatively affects the performance of applications running under a VM. Sometimes, an application running under a VM can perform better than one running under a classical OS. This is the case of *cache isolation* when the cache is divided among VMs. In this case it is beneficial to run workloads competing for cache in two different VMs [449]. Often, the cache is not equally partitioned among processes running under a classical OS, and one process may use the cache space better than the other. For example, in the case of two processes, one write-intensive and the other read-intensive, the cache may be aggressively filled by the first.

The I/O performance of applications running under a VM depends on factors, such as: the disk partition used by the VM, the CPU utilization, the I/O performance of the competing VMs, and the I/O block size. The discrepancies between the optimal choice and the default ones on a Xen platform are between 8% and 35% [449].

5.5 Hardware support for virtualization

In early 2000, it became obvious that hardware support for virtualization was necessary, and Intel and AMD started working on the first generation virtualization extensions of the x86[2] architecture. In 2005,

[2] x86-32, i386, x86, and IA-32 refer to the Intel CISC-based instruction architecture, now supplanted by x86-64, which supports vastly larger physical and virtual address spaces. The x86-64 specification is distinct from Itanium, initially known as IA-64 architecture.

Intel released two Pentium 4 models supporting *VT-x,* and in 2006, AMD announced Pacifica and then several Athlon 64 models.

The Virtual Machine Extension (VMX) was introduced by Intel in 2006, and AMD responded with the Secure Virtual Machine (SVM) instruction-set extension. The *Virtual Machine Control Structure* (VMCS) of VMX tracks the host state, and the guest VMs as control is transferred between them. Three types of data are stored in VMCS:

- *Guest state.* Holds virtualized CPU registers (e.g., control registers or segment registers) automatically loaded by the CPU when switching from kernel mode to guest mode on *VMEntry*.
- *Host state.* Data used by the CPU to restore register values when switching back from guest mode to kernel mode on *VMExit*.
- *Control data.* Data used by the hypervisor to inject events, such as exceptions or interrupts into VMs and to specify which events should cause a *VMExit*; it is also used by the CPU to specify the *VMExit* reason.

VMCS is shadowed in hardware to overcome the performance penalties of nested hypervisors discussed in Section 5.10. This enables the guest hypervisor to access VMCS directly, without disrupting the root hypervisor in the case of nested virtualization. VMCS shadow access is almost as fast as a nonnested hypervisor environment. VMX includes several instructions [253]:

VMXON—enter vmx operation;
VMXOFF—leave vmx operation;
VMREAD—read from the VMCS;
VMWRITE—write to the VMCS;
VMCLEAR—clear VMCS;
VMPTRLD—load VMCS pointer;
VMPTRST—store VMCS pointer;
VMLAUNCH/VMRESUME—launch or resume a VM; and
VMCALL—call to the hypervisor.

A 2006 paper [361] analyzes the challenges to virtualizing Intel architectures and presents VT-x and VT-i, virtualization architectures for x86 and Itanium architectures, respectively. Software solutions at that time addressed some of the challenges, but hardware solutions could not only improve performance but also security and, at the same time, simplify the software systems. The problems faced by virtualization of x86 architecture are:

1. *Ring deprivileging.* x86 architecture provides four protection rings, 0–3. A hypervisor forces a guest VM, including an OS, to run at a privilege level greater than 0. Two solutions are then possible:
   1.1. The (0/1/3) mode when the hypervisor, the guest OS, and the application run at privilege levels 0, 1 and 3, respectively; this mode is not feasible in 64-bit mode, as we shall see shortly.
   1.2. The (0/3/3) mode when the hypervisor, a guest OS, and applications run at privilege levels 0, 3, and 3, respectively.

2. *Ring aliasing.* Such problems are created where a guest OS is forced to run at a privilege level other than that it was originally designed for. For example, when the CS register[3] is *PUSH*ed, the current privilege level in the CR is stored on the stack [361].
3. *Address space compression.* A hypervisor uses parts of the guest address space to store several system data structures such as the interrupt-descriptor table and the global-descriptor table. Such data structures must be protected, but the guest software must have access to them.
4. *Nonfaulting access to privileged state.* LGDT, SIDT, SLDT, and LTR instructions load registers GDTR, IDTR, LDTR, and TR, respectively, and can only be executed by software running at privileged level 0 because they point to data structures that control CPU operation. When executed at a privilege level other than 0, the four instructions *fail silently,* and a guest OS does not realize that any of these instructions has failed.
5. *Guest system calls.* Two instructions, SYSENTER and SYSEXIT, support low-latency system calls. The first causes a transition to privilege level 0, while the second causes a transition from privilege level 0 and fails if executed at a level higher than 0. The hypervisor must then emulate every guest execution of either of these instructions and that has a negative impact on performance.
6. *Interrupt virtualization.* In response to a physical interrupt, the hypervisor generates a "virtual interrupt" and delivers it later to the target guest OS. But every OS has the ability to mask interrupts,[4] thus the virtual interrupt could only be delivered to the guest OS when the interrupt is not masked. Keeping track of all guest OS attempts to mask interrupts greatly complicates the hypervisor and increases the overhead.
7. *Access to hidden states.* Elements of the system state, e.g., descriptor caches for segment registers, are hidden; there is no mechanism for saving and restoring the hidden components when there is a context switch from one VM to another.
8. *Ring compression.* Paging and segmentation are the two mechanisms to protect hypervisor code from being overwritten by guest OS and applications. Systems running in 64-bit mode can only use paging, but paging does not distinguish between privilege levels 0, 1, and 2, thus the guest OS must run at privilege level 3, the so-called (0/3/3) mode. Privilege levels 1 and 2 cannot be used, thus the name ring compression.
9. *Frequent access to privileged resources increases hypervisor overhead.* The task-priority register (TPR) is frequently used by a guest OS; the hypervisor must protect the access to this register and trap all attempts to access it. That can cause a significant performance degradation.

A major architectural enhancement provided by the VT-x is the support for two modes of operation and a new data structure, Virtual Machine Control Structure (VMCS), including *host-state* and *guest-state* areas, see Fig. 5.3: *VMX root:* intended for hypervisor operations, and very close to the x86 without *VT-x* and *VMX nonroot:* intended to support a VM.

When executing a *VMEntry* operation, the processor state is loaded from the *guest-state* of the VM scheduled to run; then, the control is transferred from the hypervisor to the VM. A *VMExit* saves the processor state in the *guest-state* area of the running VM; it loads the processor state from the *host-state* area and finally transfers control to the hypervisor.

[3] The x86 architecture supports memory segmentation with a segment size of 64 K. The CR (code-segment register) points to the code segment. *MOV, POP*, and *PUSH* instructions serve to load and store segment registers, including CR.

[4] The interrupt flag, IF in the EFLAGS register is used to control interrupt masking.

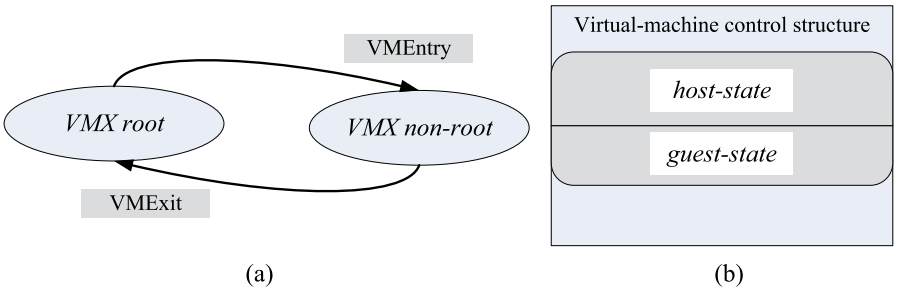

FIGURE 5.3

(a) Two VT-x operation modes and the two transitions from one to the other; (b) VMCS includes *host-state* and *guest-state* areas controlling *VM entry* and *VM exit* transitions.

All *VMExit* operations use a common entry point to the hypervisor. Each *VMExit* operation saves the reason for the exit and eventually some qualifications in VMCS. Some of this information is stored as bitmaps. For example, the *exception bitmap* specifies which one of 32 possible exceptions caused the exit. The *I/O bitmap* contains one entry for each port in a 16-bit I/O space.

The VMCS area is referenced with a physical address, and its layout is not fixed by the architecture, but can be optimized by a particular implementation. VMCS includes control bits that facilitate virtual interrupts implementation. For example, *external-interrupt exiting* when set causes the execution of a *VM exit* operation; moreover, the guest is not allowed to mask these interrupts. When the *interrupt window exiting* is set, a *VM exit* operation is triggered if the guest is ready to receive interrupts.

Processors based on two new virtualization architectures, VT-d[5] and VT-c have been developed. The first supports the I/O Memory Management Unit (I/O MMU) virtualization and the second supports the network virtualization. Also known as *PCI pass-through,* the I/O MMU virtualization gives VMs direct access to peripheral devices. VT-d supports:

- DMA address remapping: address translation for device DMA transfers.
- Interrupt remapping: isolation of device interrupts and VM routing.
- I/O device assignment: devices can be assigned by an administrator to a VM in any configuration.
- Reliability features: it reports and records DMA and interrupt errors that may otherwise corrupt memory and impact VM isolation.

5.6 QEMU

QEMU (Quick EMUlator) is a open-source machine emulator and virtualizer that emulates the host processor architecture through dynamic binary translation for several architectures, including x86-64, PowerPC, RISC-V, ARMv7, and ARMv8. It supports a set of hardware and device models for the host, enabling it to run a variety of guest operating systems. QEMU has four operating modes:

[5] The corresponding AMD architecture is called AMD-Vi.

a. *User-mode emulation.* Used primarily for fast cross-compilation and cross-debugging. Runs Linux or macOS code compiled with different instruction sets. Calls a subroutine to deal with endianness[6] and 32/64 bit addressing mismatch. This process called *thunk* is used to inject an additional calculation into another subroutine; a thunk is primarily used to delay a calculation until its result is needed, or to insert operations at the beginning or the end of the other subroutine.

b. *System emulation.* Supports virtual hosting of several VM on a physical system and emulates a full system including I/O devices. QEMU can boot several guest operating systems, including Linux, Solaris, Microsoft Windows, DOS, and BSD; it supports emulating the ISAs mentioned above.

c. *KVM hosting.* Supports setting up and migration of KVM images. It emulates the hardware, but the execution of the guest is done by KVM at the request of QEMU.

d. *Xen hosting.* Emulates the hardware, and the execution of the guest is done within Xen, totally hidden from QEMU.

QEMU can simulate multiple CPUs running as a symmetric multiprocessor. The VM can interface with different host hardware devices. Virtual disk images can be stored in a special format that only takes up as much disk space as the guest OS allows. The disk QCOW2 format allows creation of overlay images and allows reverting the emulated disk contents to an earlier state. For example, a stable image could hold a fresh install of an operating system known to work and be reverted to it when one of the overlay images become unusable due to a virus attack or some other cause.

QEMU can emulate network cards and share host connectivity using network address translation and can also connect to network cards of other instances of QEMU. QEMU does not depend on the presence of graphical output methods on the host system. Instead, it can allow one to access the screen of the guest OS via an integrated VNC server. A VNC (Virtual Network Server) transmits keyboard and mouse input from one computer to another over a network, relaying graphical-screen updates using the Remote Frame Buffer protocol (RFB).

QEMU does not require administrative rights to run, unless additional modules require it. For example, KQEMU, a Linux kernel module which speeds up emulation of x86 or x86-64 guests on platforms with the same CPU architecture, requires administrative privileges. There are several OEMS that support hot pluggable NUMA hardware.

QEMU emulates ARMv7 instruction set. Xilinx Cortex A9-based Zynq SoC includes models for: the ARM Cortex-A9 CPU, ARM Cortex-A9 MPCore, DDR Memory Controller, Static Memory Controller (NAND/NOR Flash), DMA Controller, Gigabit Ethernet Controller, USB Controller, and other systems. QEMU version 6.0.0, released in 2021, includes support for ARMv8.1-M “Helium” architecture and Cortex-M55 CPU and for ARMv8.4 TTST, SEL2, and DIT extensions.

5.7 Kernel-based Virtual Machine

Kernel-based Virtual Machine (KVM) [288] is a virtualization infrastructure of the Linux kernel.[7] KVM was released as part of the 2.6.20 Linux kernel in 2007. KVM technology is implemented as two

[6] Endianness determines if the least significant byte of a word to be stored in memory will go to lowest or the highest address of the assigned memory space.

[7] Up-to-date information about KVM can be found at http://www.linux-kvm.org.

components: (i) the KVM-loadable module installed in the Linux kernel, which provides management of the virtualization hardware, exposing its capabilities through the */proc* file system; and (ii) platform emulation, provided by a modified version of QEMU, which executes as a user-space process, coordinating with the kernel for guest operating system requests.

KVM was originally designed for x86-32 and has since been ported to x86-64, ARM, and several other architectures. Running multiple guest operating systems on x86 architecture was quite difficult before the introduction of VMX and SVM extensions of Intel architecture. These extensions enable the hypervisor to run within the privileged ring -1 and enable KVM to provide VMs with an execution environment nearly identical to the physical hardware. KVM executes the guest VM's instructions directly on the host. Each guest OS is isolated and runs in a different instance of the execution environment.

KVM provides hardware-assisted virtualization for several guest operating systems, including Linux, BSD, Solaris, Windows, and macOS. KVM provided paravirtualization support for Linux, OpenBSD, FreeBSD, NetBSD, and Windows guests using the *VirtIO* API. *Virtio* is the main platform for I/O virtualization in KVM. The idea behind *virtIO* is to have a common framework for I/O virtualization for different hypervisors.

KVM supports hot plug vCPUs, dynamic memory management, and live migration. KVM inherits Linux kernel support for *cpu-hot plugging*; this means that one can enable or disable a CPU or a core without the need to reboot. This mechanism can be used to replace defective components and can be exploited for dynamic partitioning when running multiple Linux partitions. As the workloads change, it is useful to move CPUs from one partition to the next without rebooting or interrupting the workloads.

Live migration is the process of moving a running VM or an application between different physical machines without disconnecting the client or application. With live migration, the memory, storage, and network connectivity of the VM are transferred from the original guest machine to the destination.

KVM converts Linux into a bare-metal hypervisor and inherits from the Linux kernel operating system-level components, such as a memory manager, process scheduler, input/output (I/O) stack, device drivers, security manager, and the network stack. Every VM is implemented as a regular Linux process, scheduled by the standard Linux scheduler, with dedicated virtual hardware like a network card, graphics adapter, CPU(s), memory, and disks, Fig. 5.4(b).

KVM does not run as a normal program inside Linux, but it relies on the Linux kernel infrastructure to run. Its organization is shown in Fig. 5.4.(a) As opposed to Xen which runs on the physical hardware, KVM runs inside Linux as a driver handling the new virtualization instructions exposed by hardware. KVM has several components:

1. A generic host kernel module exposing the architecture-independent functionality.
2. An architecture-specific kernel module for the host system.
3. A user-space emulation of the VM hardware that the guest OS runs on.
4. A guest OS performance-optimization additions.

Instead of emulating several hardware systems, KVM uses higher-level client applications such as QEMU, *crosvm*, or *Firecracker* for device emulation. *kvm-userspace* is a fork of the QEMU project; it short-circuits the emulation code to allow only x86-on-x86 and use the KVM API for running the guest OS on the host CPU. When the guest OS performs a privileged operation, the CPU exits and KVM takes over. If KVM itself can service the request, it runs it and then gives control back to the guest.

Firecracker is an open-source virtualization technology built for creating and managing secure, multitenant container and function-based services, such as AWS Lambda and AWS Fargate, a serverless

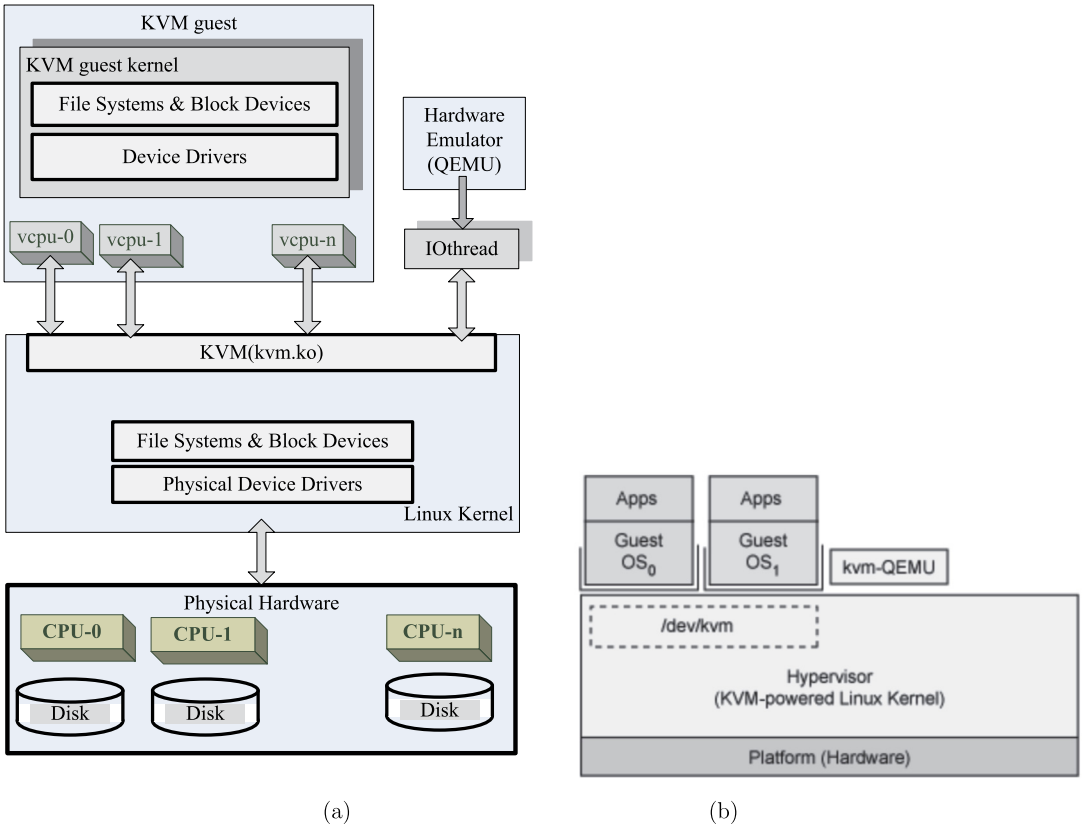

FIGURE 5.4

(a) KVM organization; KVM runs inside Linux as a driver handling the new virtualization instructions exposed by hardware; the *IOthread* generates requests on the guest's behalf to the host; it also handles events. (b) Multiple VMs running under KVM.

compute engine for containers working with Amazon Elastic Container Service (ECS) and Amazon Elastic Kubernetes Service (EKS). Firecracker uses KVM to manage microVMs.

KVM exposes the */dev/kvm* interface enabling a user-space host to: (a) setup the guest VM address space; (b) feed the guest simulated I/O; and (c) map the video display of the host. The host supplies a firmware image used by the guest to bootstrap into the host OS.

Several characteristics of KVM and other virtualization environments are displayed in Table 5.1. KVM has a number of important advantages over other virtualization environments:

1. Lower total cost of ownership and no vendor lock-in.
2. Excellent performance: apps run faster on KVM compared with other hypervisors.
3. The open-source advantage, the access source code, and the flexibility to integrate with anything.

Table 5.1 Virtualization software. Only a subset of architectures and operating systems are shown. Operating systems abbreviated as F, L, M, U, and W for Free-BSD, Linux, MacOS, Unix-like, and Windows, respectively. Software licenses are described at https://en.wikipedia.org/wiki/GNU_General_Public_License.

Name	Creator/Licences	Host CPU	Guest CPU	Host OS	Guest OS
KVM	Red Hat GPLV2	x86-64, ARM, Alpha	Same as host	L, F L, F, M, W	L, F W
QEMU	– GPL/LGPL /	x86-64, ARM Alpha	x86-64, ARM Alpha	L, F, M, W	Changes regularly
VMware ESX server	VMware / Proprietary	x86-64	x86-64	no host	L, F, W
VMware Fusion	VMware / Proprietary	x86-64	x86-64	M	L, F, W
VMWare Server	VMWare / Proprietary	x86-64	x86-64	L, W	L, F, W
Xen	Citrix / GNU, GPLV2	x86-64, ARM	Same as host	L, U	L, F, W
Xen Server	Citrix / GNU, GPLV2	x86-64, ARM	Same as host	No host	L, F, W

4. Cross-platform interoperability: users benefit from existing infrastructure investments.
5. Simplicity to create, start, stop, pause, migrate, and template a large number of VMs on a range of hardware and software.

KVM virtualization includes sVirt and SELinux (Security-Enhanced Linux) technologies developed to detect and prevent complex security threats. The stability, consistency, and compatibility of Linux is shared with the virtual layer since KVM has been part of the Linux kernel for a long time.

In conclusion, a hypervisor using Linux as the core has tangible benefits. An obvious one is the steady progression of Linux, a platform that continues to advance, from typical optimizations and bug fixes, scheduling, and memory-management innovations to support for multiple processor architectures. Another benefit is that one can take advantage of the Linux platform as an operating system in addition to a hypervisor. In addition to running multiple guest operating systems on the Linux hypervisor, one can run other traditional applications at that level.

5.8 Xen—a hypervisor based on paravirtualization

Xen is a hypervisor developed by the Computing Laboratory at the University of Cambridge, United Kingdom, in 2003. The goal of the Cambridge group was to design a hypervisor capable of scaling to about 100 VMs running standard applications and services without any modifications to the Application Binary Interface. Fully aware that x86 architecture does not support efficiently full virtualization, the designers of Xen opted for paravirtualization.

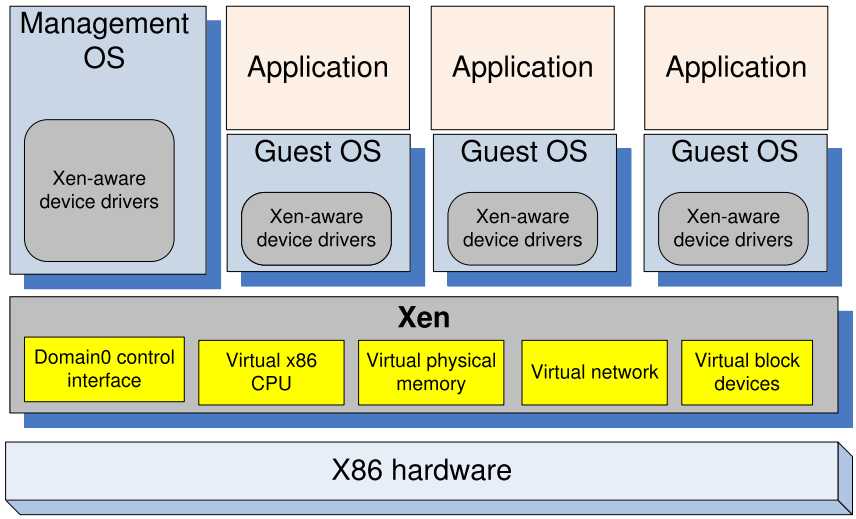

FIGURE 5.5

Xen for x86 architecture. The management OS dedicated to the execution of Xen control functions and privileged instructions resides in *Dom0*; guest operating systems and applications reside in *DomU*.

Since 2010, Xen has been a free software, developed by the community of users and licensed under the GNU General Public License (GPLv2). Several operating systems including Linux, Minix, NetBSD, FreeBSD, NetWare, and OZONE can operate as paravirtualized Xen guest operating systems running on x86, x86-64, Itanium, and ARM architectures.

We analyze next the original implementation of Xen for the x86 architecture discussed in [48]. The creators of Xen used the concept of *domain* (Dom) to refer to the ensemble of address spaces hosting a guest OS and address spaces for applications running under this guest OS. Each domain runs on a virtual x86 CPU. *Dom0* is dedicated to the execution of Xen control functions and privileged instructions, and *DomU* is a user domain, Fig. 5.5.

The most important aspects of Xen paravirtualization for virtual memory management, CPU multiplexing, and I/O device management are summarized in Table 5.2 [48]. Efficient management of TLB (Translation Look-aside Buffer), a cache for page table entries, requires either the ability to identify the OS and the address space of every entry or to allow software management of the TLB. Unfortunately, x86 architecture does not support either the tagging of TLB entries or the software management of the TLB. As a result, the address space switching when the hypervisor activates a different OS, requires a complete TLB flush. Flushing the TLB has a negative impact on performance.

The solution adopted was to load Xen in a 64-MB segment at the top of each address space and to delegate the management of hardware page tables to the guest OS with minimal intervention from Xen. The 64-MB region occupied by Xen at the top of every address space is not accessible or not remappable by the guest OS.

When a new address space is created, the guest OS allocates and initializes a page from its own memory, registers it with Xen, relinquishing control of the write operations to the hypervisor. Thus, a

Table 5.2 Paravirtualization strategies for virtual memory management, CPU multiplexing, and I/O devices for the original x86 Xen implementation.

Function	Strategy
Paging	A domain may be allocated discontinuous pages. A guest OS has direct access to page tables and handles pages faults directly for efficiency; page table updates are batched for performance and validated by Xen for safety.
Memory	Memory is statically partitioned between domains to provide strong isolation. *XenoLinux* implements a *balloon driver* to adjust domain memory.
Protection	A guest OS runs at a lower priority level, in ring 1, while Xen runs in ring 0.
Exceptions	A guest OS must register with Xen a description table with the addresses of exception handlers previously validated.
System	To increase efficiency, a guest OS must install a "fast" handler
Interrupts	A lightweight event system replaces hardware interrupts; synchronous system calls from a domain to Xen use *hypercalls* and notifications are delivered using the asynchronous event system.
Multiplexing	A guest OS may run multiple applications.
I/O devices	Data is transferred using asynchronous I/O rings.
Disk access	Only *Dom0* has direct access to IDE and SCSI disks; all other domains access persistent storage through the Virtual Block Device (VBD) abstraction.

guest OS could only map pages it owns. On the other hand, a guest OS has the ability to batch multiple page-update requests to improve performance. A similar strategy is used for segmentation.

x86 Intel architecture supports four protection rings or privilege levels; virtually all OS kernels run at level 0, the most privileged one, and applications at level 3. In Xen the hypervisor runs at level 0, the guest OS at level 1, and applications at level 3.

Applications make system calls using the so-called *hypercalls* processed by Xen; privileged instructions issued by a guest OS are *paravirtualized* and must be validated by Xen. When a guest OS attempts to execute a privileged instruction directly, the instruction fails silently.

Memory is statically partitioned between domains to provide strong isolation. To adjust domain memory, XenoLinux implements a *balloon driver* that passes pages between Xen and its own page allocator. Page faults are handled directly by the guest OS for efficiency.

Xen schedules individual domains using Borrowed Virtual Time (BVT), a work-conserving[8] and low-latency wake-up scheduling algorithm discussed in Section 9.7. BVT uses a virtual-time warping mechanism to support low-latency dispatch to ensure timely execution whenever needed, for example, for timely delivery of TCP acknowledgments.

A guest OS[9] must register with Xen a *description table* with the addresses of exception handlers for validation. Exception handlers are identical to the native x86 handlers; the only one that does not follow this rule is the page fault handler, which uses an extended stack frame to retrieve the faulty address because the privileged register *CR2*, where this address is found, is not available to a guest OS.

Each guest OS can validate and then register a "fast" exception handler executed directly by the processor without the interference of Xen. A lightweight event system replaces hardware interrupts;

[8] A work-conserving scheduling algorithm does not allow the processor to be idle when there is work to be done.

[9] In the original Xen implementation [48], a guest OS could be either XenoLinix, XenoBSD, or XenoXP.

notifications are delivered using this asynchronous event system. Each guest OS has a timer interface and is aware of "real" and "virtual" time.

XenStore is a *Dom0* process supporting a system-wide registry and naming service. It is implemented as a hierarchical key-value stor. A *watch* function of the process informs listeners of changes of the key in the storage they have subscribed to. XenStore communicates with guest VMs via shared memory using *Dom0* privileges, rather than grant tables.

Toolstack is another *Dom0* component responsible for creating, destroying, and managing the resources and privileges of VMs. To create a new VM, a user provides a configuration file describing memory and CPU allocations, as well as device configuration. Then, *Toolstack* parses this file and writes this information in *XenStore*. *Toolstack* takes advantage of *Dom0* privileges to map guest memory, to load a kernel and virtual BIOS, and to set up initial communication channels with *XenStore* and with virtual console when a new VM is created.

Xen defines abstractions for networking and I/O devices. *Split drivers* have a frontend in the DomU and the backend in *Dom0*; the two communicate via a ring in shared memory. Xen enforces access control for the shared memory and passes synchronization signals. Access Control Lists (ACLs) are stored in the form of *grant tables*, with permissions set by the owner of the memory.

Data for I/O and network operations moves vertically through the system, very efficiently, using a set of I/O rings, see Fig. 5.6. A *ring* is a circular queue of descriptors allocated by a domain and accessible within Xen. Descriptors do not contain data, the data buffers are allocated off-band by the guest OS. Memory committed for I/O and network operations is supplied in a manner designed to avoid "crosstalk," and the I/O buffers holding the data are protected by preventing page faults of the corresponding page frames.

Each domain has one or more Virtual Network Interfaces (VNIs), which support the functionality of a network interface card. A VNI is attached to a Virtual Firewall-Router (VFR). Two rings of buffer descriptors, one for packet sending and one for packet receiving, are supported. To transmit a packet, a guest OS enqueues a buffer descriptor to the send ring, then Xen copies the descriptor and checks safety, and finally copies only the packet header, not the payload, and executes the matching rules.

Rules of the form $(< pattern >, < action >)$ require the *action* to be executed if the *pattern* is matched by the information in the packet header. The rules can be added or removed by *Dom0*; they ensure the demultiplexing of packets based on the destination IP address and port and, at the same time, prevent spoofing of the source IP address. Dom0 is the only one allowed to access directly the physical IDE or SCSI disks. All domains other than *Dom0* access persistent storage through a Virtual Block Device (VBD) abstraction created and managed under the control of *Dom0*.

Xen includes a device emulator, QEMU, to support unmodified commodity operating systems. QEMU is a machine emulator; it runs unmodified OS images and emulates those architecture's instructions for the host architecture it runs on. The project had several devices already emulated for the x86 architecture, including the chipset, network cards, and display adapters. QEMU emulates a DMA[10]

[10] Direct Memory Access (DMA) is specialized hardware that allows the I/O subsystems to access the main memory without CPU intervention. It can also be used for memory-to-memory copying and can offload expensive memory operations, e.g., scatter-gather operations, from a CPU to dedicated a DMA engine.

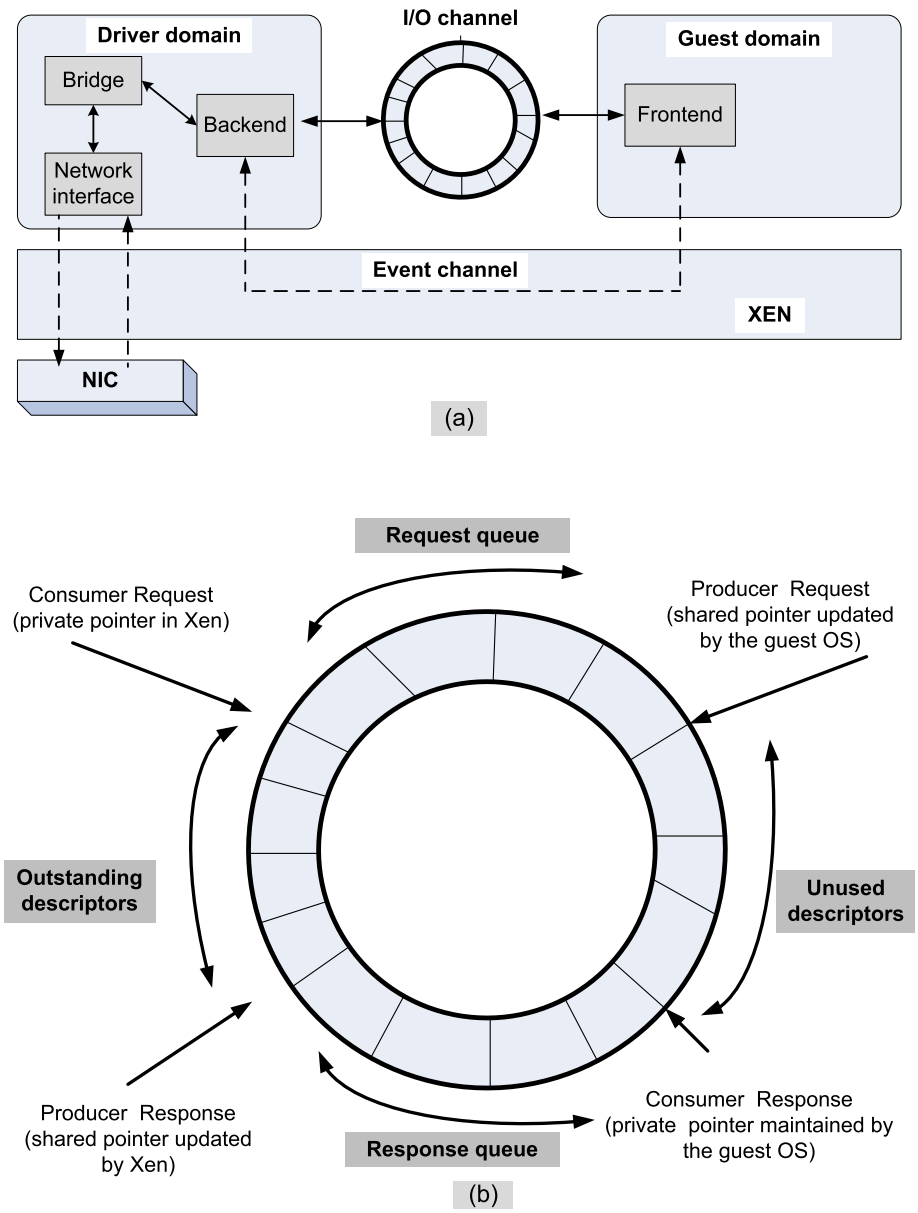

FIGURE 5.6

Xen zero-copy semantics for data transfer using I/O rings. (a) The communication between a guest domain and the driver domain over an I/O and an event channel; NIC is the Network Interface Controller. (b) The circular ring of buffers.

and can map any page of the memory in a DomU. Each VM has its own instance of QEMU, which can run either as a *Dom0* process, or as a process of the VM.

Xen, initially released in 2003, has undergone significant changes in 2005, when Intel released the *VT-x* processors. In 2006, Xen was adopted by Amazon for its EC2 service, and in 2008, Xen, running on Intel's *VT-d,* passed the ACPI S3[11] test. Xen support for *Dom0* and *DomU* was added to the Linux kernel in 2011.

In 2008, the PCI pass-through was incorporated for Xen running on *VT-d* architectures. The PCI[12] pass-through allows a PCI device, be it a disk controller, Network Interface Card (NIC),[13] graphic card, or Universal Serial Bus (USB), to be assigned to a VM. This avoids the overhead of copying and allows setting up of a *Driver Domain* to increase security and system reliability. A guest OS can exploit this facility to access the 3D acceleration capability of a graphics card. The BDF[14] of a device must be known for pass-through.

An analysis of VM performance for I/O-bound applications under Xen is reported in [405]. Two web servers, each running under a different VM, share the same server running Xen; the workload generator sends requests for files of fixed size ranging from 1 KB to 100 KB. When the file size increases from 1 KB to 10 KB and to 100 KB, the CPU utilization, the throughput, data rate, and the response times are (97.5%, 70.44%, 44.4%), (1 900, 1 104, 1 112) requests/sec, (2 018, 11 048, 11 208) KBps, and (1.52, 2.36, 2.08) msec, respectively. From the first group of results, we see that for files 10 KB or larger, the system is I/O bound. The second set of results shows that the throughput measured in requests/second decreases by less than 50% when the system becomes I/O bound, but the data rate increases by a factor of five over the same range. The response time increases only about 10% when the file size increases by two orders of magnitude.

The paravirtualization strategy in Xen is different from the one adopted by the group at the University of Washington, the creators of the Denali system [514]. Denali was designed to support a number of VMs running network services one or more orders of magnitude larger than Xen. The design of the Denali system did not target existing application binary interfaces; it does not support some features of potential guest operating systems; for example, it does not support segmentation. Denali does not support application multiplexing, running multiple applications under a guest OS, while Xen does.

Finally, a few words regarding the complexity of porting commodity operating systems to Xen. It is reported that a total of about 3 000 lines of Linux code, or 1.36%, had to be modified; for Windows XP, this figure is 4 620, or about 0.04%, [48].

---

[11] Advanced Configuration and Power Interface (ACPI) specification is an open standard for device configuration and power management by the OS. It defines four Global "Gx" states and six Sleep "Sx" states. "S3" is referred to as Standby, Sleep, or Suspend to RAM.

[12] PCI stands for Peripheral Component Interconnect and describes a computer bus for attaching hardware devices to a computer. The PCI bus supports the functions found on a processor bus, but in a standardized format independent of any particular processor. The OS queries all PCI buses at startup time to identify the devices connected to the system and the memory space, I/O space, interrupt lines, and so on needed by each device present.

[13] A Network Interface Controller is the hardware component connecting a computer to a LAN.

[14] BDF stands for Bus.Device.Function, and it is used to describe PCI devices.

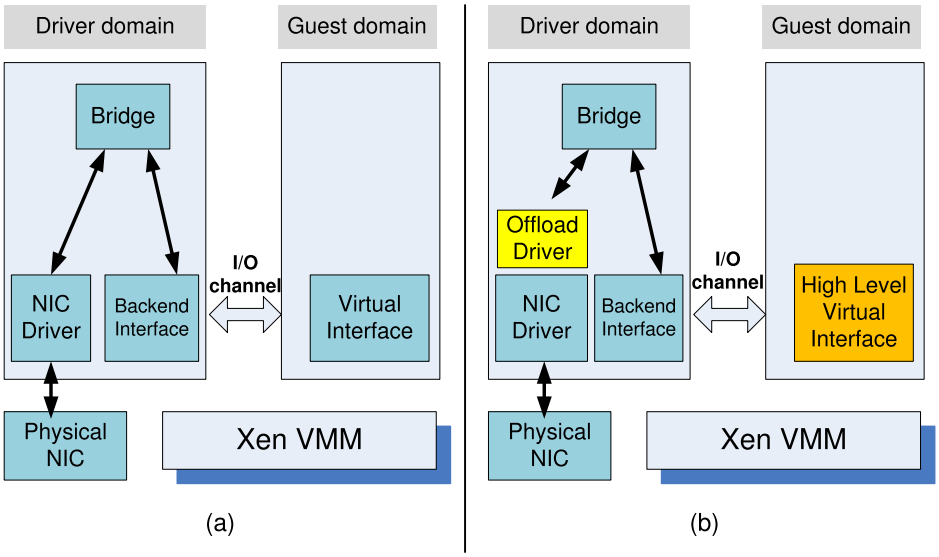

FIGURE 5.7

Xen network architecture: (a) Original architecture; (b) Optimized architecture.

5.9 Optimization of network virtualization in Xen 2.0

A hypervisor introduces a significant network communication overhead. For example, it is reported that the CPU utilization of a VMware Workstation 2.0 system running Linux 2.2.17 was 5 to 6 times higher than that of the native system (Linux 2.2.17) while saturating a 100 Mbps network [462]. In other words, the hypervisor executes a much larger number of instructions, 5 to 6 times larger, to saturate the network, while handling the same amount of traffic as the native system.

Similar overheads are reported for other hypervisors and, in particular, for Xen 2.0 [345,346]. To understand the sources of network overhead, we examine the basic network architecture of Xen, see Fig. 5.7(a). Privileged operations, including I/O, are executed by *Dom0* on behalf of a guest OS. In this context, we shall refer to *Dom0* as the *driver domain*. The *driver domain* is called in to execute networking operations on behalf of the *guest domain*. It uses the native Linux driver for the NIC (Network Interface Controller), which in turn, communicates with the physical NIC, also called the network adapter. The *guest domain* communicates with the *driver domain* through an I/O channel, see Section 5.8. More precisely, the guest OS in the guest domain uses a virtual interface to send and receive data to/from the backend interface in the driver domain.

A *bridge* uses broadcast communication to identify the MAC address[15] of a destination system. Once identified, this address is added to a table. A bridge uses the link layer protocol to send a packet

[15] A Media Access Control (MAC) address is a unique identifier permanently assigned to a network interface by the manufacturer.

Table 5.3 A comparison of send and receive data rates for a native Linux system, the Xen driver domain, an original Xen guest domain, and an optimized Xen guest domain.

System	Receive data rate (Mbps)	Send data rate (Mbps)
Linux	2 508	3 760
Xen driver	1 728	3 760
Xen guest	820	750
optimized Xen guest	970	3 310

to the proper MAC address, rather than broadcast it when the next packet for the same destination arrives.

The bridge in the driver domain performs a multiplexing/demultiplexing function. Packets received from the NIC are demultiplexed and sent to the VMs running under the hypervisor. Similarly, packets arriving from multiple VMs have to be multiplexed into a single stream before being transmitted to the network adaptor. In addition to bridging, Xen supports IP routing based on network address translation.

Table 5.3 shows the ultimate effect of this longer processing chain for the Xen hypervisor, as well as the effect of optimizations [346]; the receiving and sending rates from a guest domain are roughly 30% and 20%, respectively, of the corresponding rates of a native Linux application. Packet multiplexing/demultiplexing accounts for about 40% and 30% of the communication overhead for the incoming traffic and for the outgoing traffic, respectively.

The Xen network optimization discussed in [346] covers optimization of: (i) the virtual interface; (ii) the I/O channel; and (iii) the virtual memory. The effects of these optimizations are significant for the send-data rate from the optimized Xen guest domain, an increase from 750 to 3 310 Mbps and rather modest for the receive-data rate, 970 versus 820 Mbps.

We next examine each optimization area and start with the virtual interface. There is a tradeoff between generality and the flexibility, on one hand, and the performance on the other hand. The original virtual network interface provides the guest domain with a simple low-level network interface abstraction supporting sending and receiving primitives.

This design supports a wide range of physical devices attached to the driver domain but does not take advantage of the capabilities of some physical NICs, such as checksum offload, e.g., TSO,[16] and scatter/gather DMA support. These features are supported by the High Level Virtual Interface of the optimized system, Fig. 5.7(b).

The next target of the optimization effort is the communication between the guest domain and the driver domain. Rather than copying a data buffer holding a packet, each packet is allocated space in a new page, and then the physical page containing the packet is remapped onto the target domain; for example, when a packet is received, the physical page is remapped to the guest domain. The optimization is based on the observation that there is no need to remap the entire packet.

For example, when sending a packet, the network bridge needs only to know the MAC header of the packet. As a result of this, the optimized implementation is based on an "out-of-band" channel used

[16] TSO stands for TCP segmentation offload. This option enables the network adapter to compute the TCP checksum on *transmit* and *receive*, and to save the host CPU the overhead for computing the checksum; large packets have larger savings.

by the guest domain to provide the bridge with the packet MAC header. This strategy contributed to a better than four times increase in the send data rate compared with the nonoptimized version.

The third optimization covers virtual memory. The virtual memory in Xen 2.0 takes advantage of the *superpage* and *global page mapping* hardware features available on Pentium and Pentium Pro processors. A superpage increases the granularity of the dynamic address translation; a superpage entry covers 1 024 pages of physical memory, and the address translation mechanism maps a set of contiguous pages to a set of contiguous physical pages. This helps reduce the number of TLB misses.

All pages of a superpage belong to the same guest OS. When new processes are created, the guest OS must allocate read-only pages for the page tables of the address spaces running under the guest OS. This forces the system to use traditional page mapping, rather than the superpage mapping. The optimized version on network virtualization uses a special memory allocator to avoid this problem.

5.10 Nested virtualization

Nested virtualization describes a system organization when a *guest hypervisor* runs inside a VM, which is itself running under a *host hypervisor*. Fig. 5.8(a) illustrates an instance of nested virtualization where a KVM is the host hypervisor supporting resource sharing among three VMs. Two of the three VMs run guest hypervisors, Xen, and VMware's ESXi, and the third VM runs Windows. There are two VM running under guest hypervisors, one runs Linux under Xen and another runs Windows under ESXi.

Nested virtualization is useful for experimenting with server setup or testing configurations. Nested virtualization enables IaaS users to run their own hypervisor as a VM. Nested virtualization can be also used for live migration of hypervisors together with their guest VMs for load balancing, for hypervisor-level protection, and for supporting other security mechanisms. Another use of nested virtualization is to experiment with cloud interoperability alternatives.

The x86 virtualization is based on the *trap and emulate* model. This model requires every sensitive instruction executed by either a guest hypervisor or OS to be handled by the most privileged hypervisor. Nested virtualization incurs a substantial performance price, unless the switching between the levels of the virtualization stack is optimized. It is thus not surprising that nested virtualization is not supported by many hypervisors, and not all operating systems can nest successfully with all hypervisors.

Nested virtualization is supported differently by Intel and AMD processors. Consider for example the Intel version discussed in [127] and illustrated in Fig. 5.8(b). In this example, KVM runs at level L_0 and controls the allocation of all resources. Xen runs at level L_1, and KVM uses $VMCS_{01}$ for the VM running Xen.

When Xen executes a *vmlaunch* operation to start a new VM, (see Section 5.5), a new VM control structure, $VMCS_{12}$ is created. Then, *vmlaunch* traps to L_0 and L_0 merges $VMCS_{01}$ with $VMCS_{12}$ and creates $VMCS_{02}$ to run Linux at level L_2. When an application running under Linux at level L_2 makes a system call, or when Linux itself executes a privileged instruction, L_2 traps and KVM decides whether to handle the trap itself or to forward it to Xen at level L_1 and, eventually, Xen resumes execution.

Nested virtualization is limited by the hardware support. When the hardware supports *multilevel nested virtualization*, each hypervisor handles all traps caused by sensitive instructions of guest hypervisors running directly above of it. Multilevel nested virtualization is supported by the IBM System Z architecture [381]. Intel and AMD processors support only *single-level nested virtualization*. This

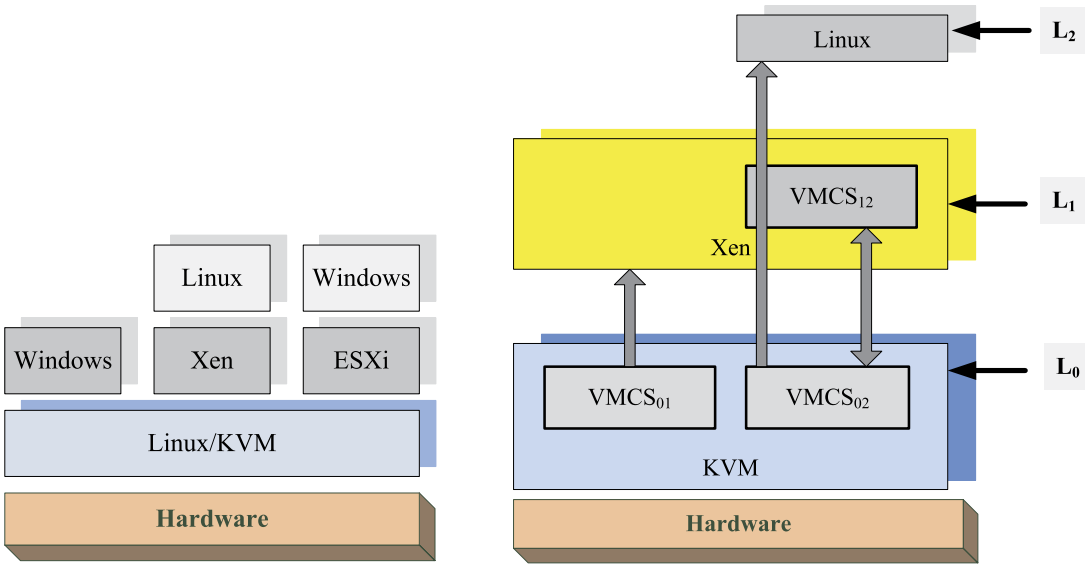

FIGURE 5.8

Nested virtualization [127]. (a) KVM allows three VM to run concurrently. Two VMs run hypervisors Xen and ESXi, and the third runs Windows. A VM runs Linux under Xen, and another VM runs Windows under ESXi. (b) Intel-supported nested virtualization. KVM runs at level L_0, Xen runs at level L_1, and KVM uses $VMCS_{01}$ for the VM running Xen.

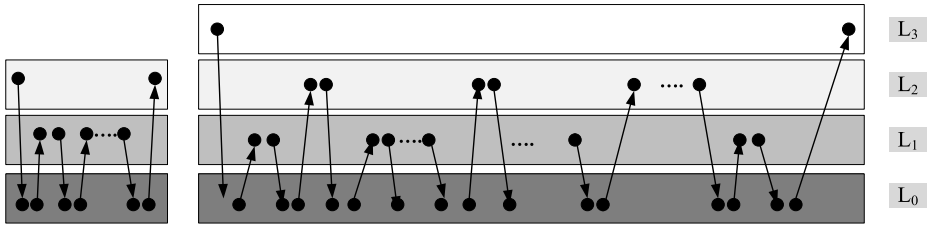

FIGURE 5.9

Nested virtualization with single-level hardware virtualization support. A trap is handled by the L_0 trap handler regardless of the hypervisor where a trap occurs. Nested traps for: (Left) Two-level, L_0, L_1, and L_2 nested hypervisor(s); and (Right) Three-level, L_0, L_1, L_2, and L_3 nested hypervisor(s).

implies that the host hypervisor, the one running directly above the hardware and managing all system resources, handles all trapped instructions as shown in Fig. 5.9.

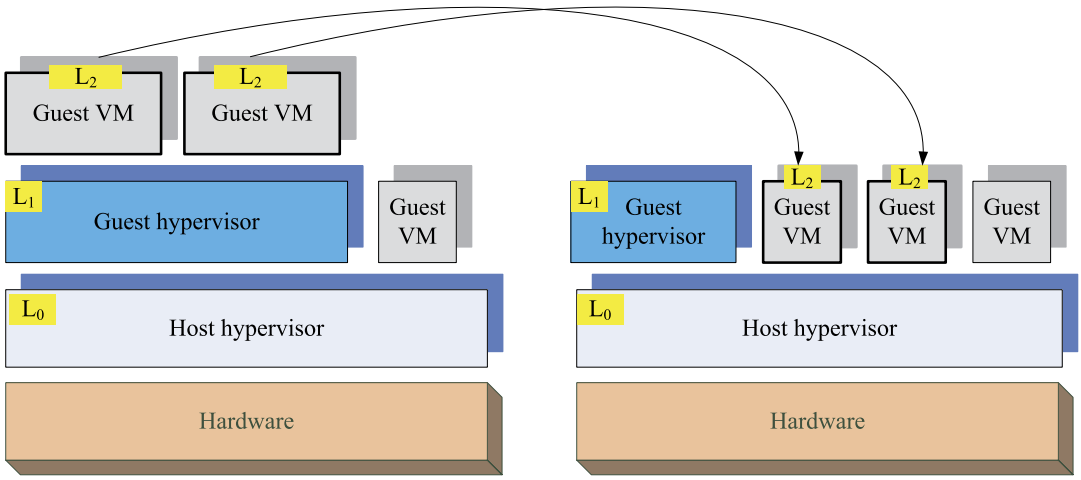

FIGURE 5.10

Multiple virtualization levels on the left are multiplexed into the singe hardware virtualization level on the right, as described in [57]. A VMX instruction used by a guest hypervisor running in guest mode at level L_i is trapped and translated by the host hypervisor at level L_0 running in kernel mode into one that can be used to a VM at level L_{i+1}.

An in-depth discussion of the intricacies of nested virtualization on x86 architecture can be found in a paper describing the Turtle project from IBM Israel [57]. A guest hypervisor cannot use the hardware virtualization support because the x86 provides only a single-level hardware virtualization support. The aim of the project was to show that a 6–8% overhead is feasible for "unmodified binary-only hypervisors executing nontrivial workloads."

VMX instructions can only be successfully executed in kernel mode. A guest hypervisor at level L_i operates in guest mode, and, whenever it executes a VMX instruction to launch a level L_{i+1} guest, the instruction is trapped and handled at level L_0. Trapping execution exceptions enables the host hypervisor at level L_0 running in kernel mode to emulate VMX instruction executed by guest hypervisors at level L_i. This mechanism supports a critical technique for increasing efficiency of nested virtualization, the multiplexing multiple hypervisors, as shown in Fig. 5.10.

As long as the host hypervisor at level L_0 emulates faithfully the VMX instruction set, a guest hypervisor at level L_1 cannot distinguish if it is running directly on the hardware or not. It follows that the guest hypervisor at level L_1 can launch VMs using the standard mechanisms. The guest hypervisor at level L_1 does not run at the highest privileged level, and the action of starting a VM is trapped and handled by the trap handler at level L_0. The specification of the new VM is then translated by the host hypervisor at level L_0 running in kernel mode into one that can be used to run L_2 directly on the hardware. This translation includes converting L_1 physical addresses to the physical address space of L_0.

The guest hypervisor can use the same technique to give another guest hypervisor at level L_2 the same illusion, namely, that it is running directly on the hardware. The process can be extended, a hypervisor at level L_i giving the illusion that the one at level L_{i+1} is running directly on the hardware.

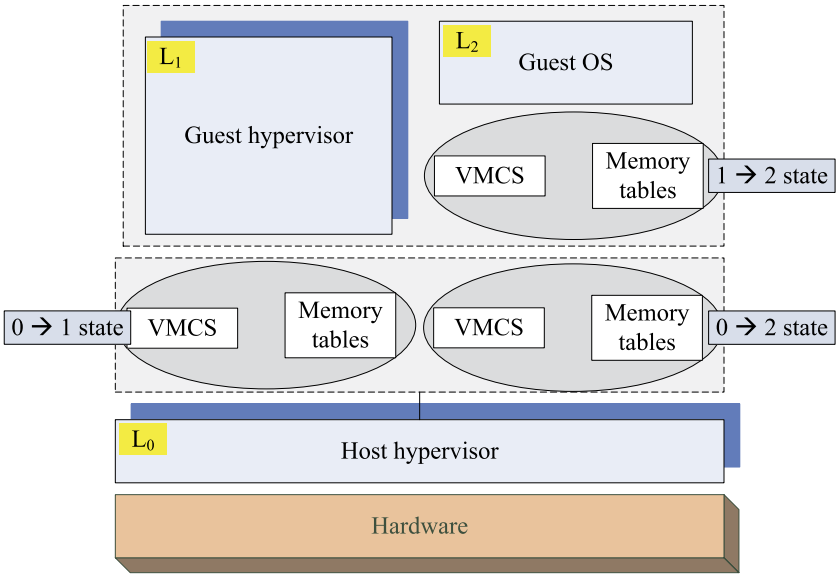

FIGURE 5.11

VMX extension for nested virtualization as described in [57].

L_0, the host supervisor, manages resources allocated to L_1, a guest hypervisor and L_2, a guest OS, using $VMCS_{0\to1}$ and $VMCS_{0\to2}$ environment specification, respectively. L_1 creates $VMCS_{1\to2}$ within its own virtualized environment, and the processor uses it to emulate VMX for L_1, as illustrated by Fig. 5.11. Switching from one level to another is emulated. For example, when an *VMExit* occurs while L_2 is running there are two possible paths:

- When an external interrupt, a nonmaskable interrupt, or any trappable event specified in $VMCS_{0\to2}$ that was not specified in $VMCS_{1\to2}$ occurs, L_0 handles the event and then L_2 execution is resumed.
- Trappable events specified in $VMCS_{1\to2}$ are handled by L_1. The host hypervisor L_0 forwards the event to L_1 by copying $VMCS_{0\to2}$ fields updated by the processor to $VMCS_{1\to2}$ and then resuming L_1. This makes the L_1 hypervisor believe there was a VMExit directly from L_2 to L_1, handle the event and then resume L_2 by executing *VMLAUNCH* or *VMRESUME*, emulated by L_0.

Another complication of nested virtualization is that the MMU must be virtualized to allow a guest hypervisor to translate guest virtual addresses to guest physical addresses. A multidimensional paging for multiplexing the three needed translation tables onto the two available in hardware is described in [57].

5.11 A trusted kernel-based virtual machine for ARMv8

Advanced RISC Machine (ARM) processors are widely used in mobile devices such as smartphones, tables, and laptops. ARM processors are also used in embedded systems connected to the IoT. Such

systems require an increased level of security, therefore, it is not surprising that the latest generation of the ubiquitous ARM processors support the Trusted Execution Environment (TEE).

TEE functions are summarized at http://www.globalplatform.org/ as: "TEE's ability to offer isolated safe execution of authorized security software, known as trusted applications, enables it to provide end-to-end security by enforcing protected execution of authenticated code, confidentiality, authenticity, privacy, system integrity and data access rights." Trusted applications, running in TEE, their assets, and data are isolated from the Rich Execution Environment where standard operating systems such as Linux run. TEE consists of several components:

1. A common abstraction layer, the Trusted Core Framework, providing OS functions, such as memory management, entry points for trusted applications, panic and cancelation handling, and trusted application properties access.
2. Inter-process communication used by rich execution environment applications to request services from TEE.
3. API for accessing services such as Trusted Storage for Data and Keys, TEE Cryptographic Operations, Time, and TEE Arithmetica.

AArch64, the 64-bit ARM architecture is compatible with AArch32. The members of the AArch64 family including ARMv8 Cortex-Axx (xx = {35, 53, 57, 72, 73}) processors share a number of features:

- Support a new instruction set, A64, with the same instruction semantics as AArch32, but with fewer conditional instructions. A64 includes major functional enhancements:
  1. 32 128-bit wide registers.
  2. Advanced SIMD supporting double-precision floating-point execution.
  3. Advanced SIMD supporting full IEEE 754[17] execution.
- Include instruction-level support for cryptography, two encode and two decode instructions for AES, SHA-1, and SHA-256 support.
- Have 31 general-purpose registers accessible at all times.
- Provide revised exception handling in the AArch64 state.
- Support virtualization.
- Support the Trust Zone and the Global Trust TEE.

The *ARM Trust Zone* (ATZ) splits an ARM-based system into the Secure World, a trusted subsystem, responsible for the boot and the configuration of the entire system and the Non Secure World (NSW) intended for hosting operating systems such as Linux and Android, as well as user applications. A CPU has banked registers for each World.

Security-specific configurations can only be performed in the Secure World mode, while access to AMBA peripherals such as fingerprint readers, cryptographic engines, and others can be restricted to the Secure World. A secure context switch procedure routs interrupts either to the Secure or to the Non Secure World, depending upon the configuration and enables the two Worlds to communicate with

[17] IEEE Standard for Floating-Point Arithmetic (IEEE 754) defines arithmetic formats, interchange formats, rounding rules, operations, and exception handling for floating-point numbers, see "IEEE Standard for Floating-Point Arithmetic" IEEE Computer Society (August 29, 2008). doi:10.1109/IEEESTD.2008.4610935.

one another. ATZ is enabled by a set of hardware security extensions including: (i) a CPU with ARM Security Extensions (SE); (ii) a compliant Memory Management Unit (MMU); (iii) an AMBA system bus[18]; and (iv) interrupt and cache controllers.

T-KVM, a KVM-based trusted hypervisor for ARMv8, combines Trust Zone with GlobalPlatform TEE and SELinux [388]. T-KVM implements: (a) a trusted boot; (b) support for Trusted Computing inside a Virtual Machine; (c) a zero copy shared memory mechanism for data sharing between the two Trust Zone Worlds and between the VM and the host; and (d) a secure, ideally real-time, reliable, and error-free OS running in the Secure World.

The challenge of a secure boot is to eliminate vulnerabilities when the security mechanisms are not yet in place. The solution implemented in T-KVM is a four-stage boot process. A small program stored in the on-chip ROM, along with the public key needed for the attestation of the second stage loader, is activated in the first stage.

The second stage loads the microkernel in the Secure World zone and activates it. The third stage checks the integrity of the Linux kernel, a Non Secure World binary, and of its loader and, finally, the fourth stage runs it. The failure of any check in this chain of events brings the system to a secure state stop. T-KVM boot sequence is shown in Fig. 5.12(a).

The main challenge for supporting Trusted Computing inside a VM is the virtualization of the TEE APIs. To allow the TEE Client API to execute directly in the Guest OS, a specific QEMU device implements the TEE control plane and sets up its data plane, see Fig. 5.12(b). Requests for service, such as initialization/close session, command invocation, and notification of response, are sent to the TEE Device, which delivers them either to the trusted applications or to the client applications running on the guest OS. The data plane uses the shared memory. The TEE device notifies its driver upon receiving a response notification from the Trust Zone Secure World (TZSW), and the driver forwards the information to the Guest Client application.

Zero-copy shared memory is based on the fact that TAs can read/write VMs shared memory because TZSW can access the entire NSW workspace. The TEE Device control plane extends T-KVM shared memory mechanism, enabling it to send the shared memory address to the Secure World applications.

5.12 Paravirtualization of Itanium architecture

We now analyze some of the findings of a Xen project at HP-Laboratories [322]. This analysis will help us better understand the impact of the computer architecture on the ability to virtualize efficiently a given computer architecture. The goal of the project was to create a hypervisor for the Itanium family of IA64 Intel processors.

Itanium[19] is a processor developed jointly by HP and Intel based on a new architecture, the Explicitly Parallel Instruction Computing. This architecture enables the processor to execute multiple instructions in each clock cycle and implements a form of Very Long Instruction Word (VLIW) archi-

[18] The Advanced Microcontroller Bus Architecture (AMBA) is an open-standard, on-chip interconnect specification for connection and management of a large numbers of controllers and peripherals.

[19] In the late 2000s, Itanium was the fourth-most deployed microprocessor architecture for enterprise-class systems. The first three were Intel's x86-64, IBM's Power Architecture, and Sun's SPARC.

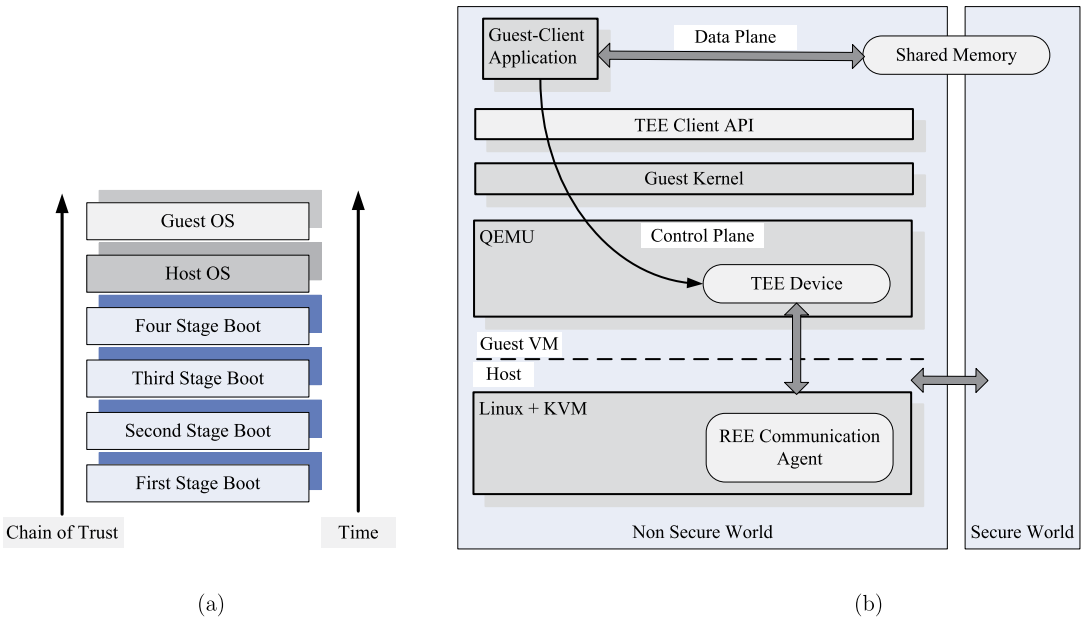

FIGURE 5.12

T-KVM. (a) The boot sequence; a trusted application runs the Non Secure loader in stage three and in stage four a Non Secure OS is booted. TEE attestation and SELinux permissions are enforced while the host OS is running. Guest Client applications (CSs) use the Secure TEE Services while the Guest OS is running. (b) Communication between the Non Secure and the Secure Worlds, as described in [388].

tecture. In VLIW, a single instruction word contains multiple instructions, see http://www.dig64.org/about/Itanium2_white_paper_public.pdf.

The design mandated that the hypervisor should be capable of supporting execution of multiple operating systems in isolated protection domains with security and privacy enforced by the hardware. A hypervisor was also expected to support optimal server utilization and allow comprehensive measurement and monitoring for detailed performance analysis.

Virtualization of the IA64 architecture. The discussion in Section 5.3 shows that to be fully virtualizable, the ISA of a processor must conform to a set of requirements. Unfortunately, the IA64 architecture does not meet these requirements and that made the Xen project more challenging.

We first review the features of the Itanium processor important for virtualization and start with the observation that the hardware supports four *privilege rings*, PL0, PL1, PL2, and PL3. Privileged instructions can only be executed by the kernel running at level PL0, while applications run at level PL3 and can only execute nonprivileged instructions; PL1 and PL2 rings are generally not used.

The hypervisor uses *ring compression* and runs itself at PL0 and PL1, while forcing a guest OS to run at PL2. A first problem called *privilege leaking* is that several nonprivileged instructions allow an

application to determine the Current Privilege Level (CPL). As a result, a guest OS may not accept to boot or run, or may itself attempt to make use of all four privilege rings.

Itanium was selected because of its multiple functional units and multithreading support. The Itanium processor has 30 functional units: six general-purpose ALUs, two integer units, one shift unit, four data cache units, six multimedia units, two parallel shift units, one parallel multiply, one population count, three branch units, two 82-bit floating-point multiply–accumulate units, and two SIMD floating-point multiply–accumulate units. A 128-bit instruction word contains three instructions; the fetch mechanism can read up to two instruction words per clock from the L1 cache into the pipeline. Each unit can execute a particular subset of the instruction set.

The hardware supports 64-bit addressing. The processor has 32 64-bit general-purpose registers numbered from R0 to R31 and 96 automatically renumbered registers, R32 through R127, used by procedure calls. When a procedure is entered, the *alloc* instruction specifies the registers the procedure could access by setting the bits of a seven-bit field that controls the register usage; an illegal *read* operation from such a register out of range returns a zero value, while an illegal *write* operation to it is trapped as an illegal instruction.

The Itanium processor supports isolation of the address spaces of multiple processes with eight privileged *region* registers; the *processor abstraction layer* firmware allows the caller to set the values in the region register. The hypervisor intercepts the privileged instruction issued by the guest OS to its processor abstraction layer and partitions the set of address spaces among the guests OS to ensure isolation. Each guest is limited to 2^{18} address spaces.

The hardware has an *IVA register* to maintain the address of the *interruption vector table*; the entries in this table control both the interrupt delivery and the interrupt state collection. Different types of interrupts activate the interrupt handlers pointed at from this table, provided that the particular interrupt is not disabled. Each guest OS maintains its own version of this vector table and has its own IVA register; the hypervisor uses the guest OS IVA register to give control to the guest interrupt handler when an interrupt occurs.

CPU virtualization. When a guest OS attempts to execute a privileged instruction, the hypervisor traps and emulates the instruction. For example, when the guest OS uses the *rsm psr.i* instruction to turn off delivery of a certain type of interrupts, the hypervisor does not disable the interrupt but records the fact that interrupts of that type should not be delivered to the guest OS and, in this case, the interrupt should be masked.

There is a slight complication because the Itanium does not have an Instruction Register (IR) and the hypervisor has to use state information to determine if an instruction is privileged. Another complication is caused by the *register stack engine,* which operates concurrently with the processor and may attempt to access memory (load or store) and generate a page fault. Normally, the problem is solved by setting up a bit indicating that the fault is due to the register stack engine and, at the same time, the engine operations are disabled. The handling of this problem by the hypervisor is more intricate.

A number of *privileged-sensitive* instructions behave differently at different privilege levels. The hypervisor replaces each one of them with a privileged instruction during the dynamic transformation of the instruction stream. Among the instructions in this category are: (i) *cover*, saves stack information into a privileged register; the hypervisor replaces it with a *break.b* instruction; (ii) *thash* and *ttag*, access data from privileged virtual-memory control structures and have two registers as arguments. The hypervisor takes advantage of the fact that an illegal read returns a zero and an illegal write to a register in the range 32 to 127 is trapped and translates these instructions as $thash\ Rx = Ry \rightarrow tpa\ Rx =$

$R(y+64)$ and *ttag* $Rx = Ry \rightarrow$ *tak* $Rx = R(y+64)$, $0 \leq y \leq 64$; and (iii) access to performance data from performance data registers is controlled by a bit in the *Processor Status Register* with the *PSR.sp* instruction.

Memory virtualization. The virtualization is guided by the realization that a hypervisor should not be involved in most of memory read and write operations to prevent a significant degradation of the performance, but, at the same time, the hypervisor should exercise tight control and prevent a guest OS from acting maliciously. The Xen hypervisor does not allow a guest OS to access the memory directly, but instead, it inserts an additional layer of indirection called *metaphysical addressing* between virtual and real addressing.

A guest OS is placed in the metaphysical addressing mode. If the address is virtual, then the hypervisor first checks if the guest OS is allowed to access that address, and, if so, the hypervisor provides the regular address translation. The hypervisor is not involved when the address is physical. The hardware distinguishes between virtual and real addresses using bits in the Processor Status Register.

5.13 A performance comparison of virtual machines

There is well-documented evidence that hypervisors negatively affect the performance of applications [48,345,346]. The topic of this section is a quantitative analysis of the performance of VMs. The performance of two virtualization techniques is compared with the performance of a plain-vanilla Linux. The two VM systems are Xen and OpenVZ based [385] on paravirtualization and full virtualization, respectively.

OpenVZ, a system based on OS-level virtualization, uses a single-patched Linux kernel. The guest operating systems in different containers may be different software distributions, but must use the same Linux kernel version that the host uses. An OpenVZ container emulates a separate physical server; it has its own files, users, process tree, IP address, shared memory, semaphores, and messages. Each container can have its own disk quotas.

OpenVZ's lack of virtualization flexibility is compensated by a lower overhead. OpenVZ memory allocation is more flexible than in hypervisors based on paravirtualization. The memory not used in one virtual environment can be used by other virtual environments. The system uses a common file system; each virtual environment is a directory of files isolated using *chroot*. To start a new VM, one needs to copy the files from one directory to another, create a *config* file for the VM, and launch the VM.

OpenVZ has a two-level scheduler: at the first level, the fair-share scheduler allocates CPU time slices to containers based on *cpuunits* values. The second-level scheduler is a standard Linux scheduler deciding what process to run in that container. The I/O scheduler is also two-level; each container has an I/O priority, and the scheduler distributes the available I/O bandwidth according to priorities.

The discussion in [385] is focused on user's perspective, thus, the performance measures analyzed are the throughput and the response time. The general question is whether consolidation of the applications and the servers is a good strategy for cloud computing. The specific questions examined are: How does performance scale up with the load? What is the impact on performance of a mixture of applications? What are the implications of the load assignment on individual servers?

There is substantial experimental evidence that the load placed on system resources by a single application varies significantly in time. A time series displaying CPU consumption of a single application clearly illustrates this fact and justifies CPU multiplexing among threads and/or processes. The

concept of *application and server consolidation* is an extension of the idea of creating an aggregate load consisting of several applications and aggregating a set of servers to accommodate this load. The peak resource requirements of individual applications are unlikely to be synchronized, therefore, the aggregate average resource utilization is expected to increase.

The application used for comparison in [385] is a two-tier system consisting of an Apache web server and a MySQL database server. A client of the web application starts a session as the user browses through various items in the database, requests information about individual items, and buys or sells items. Each session requires the creation of a new thread; thus, an increased load means an increased number of threads. To understand the potential discrepancies in performance among the three systems, a performance-monitoring tool reports the counters that enable the estimation of: (i) the CPU time used by a binary; (ii) the number of L2-cache misses; and (iii) the number of instructions executed by a binary.

The experimental setup for three different experiments are shown in Fig. 5.13. The two tiers of the application, the web and the database, each run on a single server for the Linux, the OpenVZ, and the Xen systems in the first group of experiments. When the workload increases from 500 to 800 threads, the throughput increases linearly with the workload.

Response time increases only slightly for the base system and for the OpenVZ system, while it increases 600% for the Xen system. For 800 threads, the response time of the Xen system is four times larger than for OpenVZ. CPU consumption grows linearly with the load in all three systems. DB consumption represents only 1–4% of CPU consumption.

For a given workload the web-tier CPU consumption for the OpenVZ system is close to the base system, Linux, and it is about half of that for the Xen system. The performance analysis tool shows that the OpenVZ execution has two times more L2-cache misses than the base system, while the Xen *Dom0* has 2.5 times more, and the Xen application domain has $nine$ times more cache misses.

The base system and the OpenVZ run a Linux OS, and the sources of cache misses can be compared directly, while Xen runs a modified Linux kernel. For the Xen-based system, the procedure $hypervisor\_callback$, invoked when an event occurs, and the procedure $evtchn\_do\_upcall$, invoked to process an event, are responsible for 32% and 44%, respectively, of the L2-cache misses. The percentage of the instructions invoked by these two procedures are 40% and 8%, respectively.

Most L2-cache misses in OpenVZ and the base system occur in: (i) $do\_anonimous\_pages$, a procedure used to allocate pages for a particular application with the percentage of cache misses 32% and 25%, respectively; (ii) procedures $\_copy\_to\_user\_ll$ and $\_copy\_from\_user\_ll$ used to copy data from user to system buffers and back with the percentage of cache misses $(12 + 7)\%$ and $(10 + 1)\%$, respectively. The first figure refers to copying from user to system buffers and the second to copying from system buffers to the user space.

The second group of experiments use two servers, one for the web and the other for the DB application, for each one of the three systems. When the load increases from 500 to 800 threads, the throughput increases linearly with the workload. The response time of the Xen system increases only 114%, compared with 600% reported for the first experiments.

The CPU time of the base system, the OpenVZ system, the Xen *Dom0,* and the *User Domain* are similar for the web application; for the DB application, the CPU time of the OpenVZ system is twice as large as that of the base system, while *Dom0* and the *User Domain* require CPU times of 1.1 and 2.5 times larger than the base system.

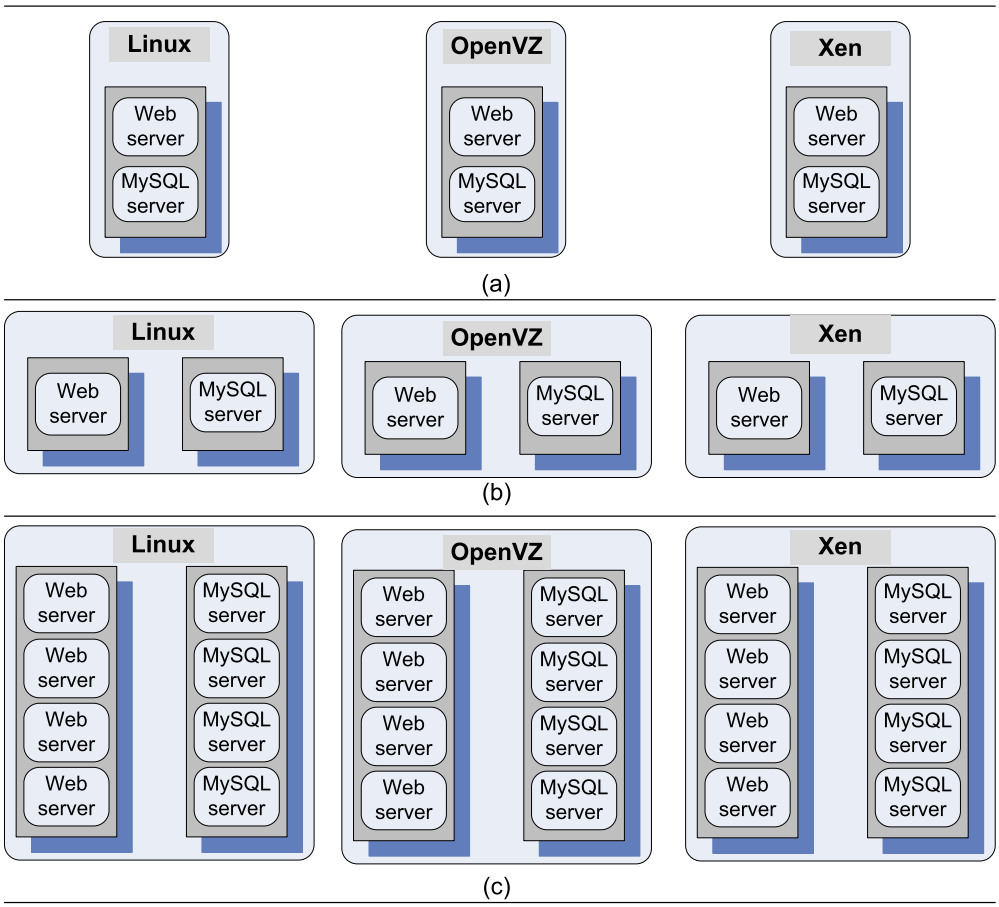

FIGURE 5.13

The setup for the performance comparison of a native Linux system with OpenVZ and the Xen systems. The applications are a web server and a MySQL database server. (a) In the first experiment, the web and the DB share a single system; (b) In the second experiment, the web and the DB run on two different systems; (c) In the third experiment, the web and the DB run on two different systems, and each has four instances.

The L2-cache misses for the web application relative to the base system are: the same for OpenVZ and 1.5 larger for *Dom0* of Xen and 3.5 times larger for the *User Domain*. The L2-cache misses for the DB application relative to the base system are: *two* times larger for the OpenVZ, 3.5 larger for *Dom0* of Xen, and seven times larger for the *User Domain*.

The third group of experiments uses two servers, one for the web and the other for the DB application, for each one of the three systems, but runs four instances of the web and the DB application on the two servers. The throughput increases linearly with the workload for the range used in the previous

two experiments, from 500 to 800 threads. The response time remains relatively constant for OpenVZ and increases five times for Xen.

The main conclusion drawn from these experiments is that the virtualization overhead of Xen is considerably higher than that of OpenVZ and that this is due primarily to L2-cache misses. Xen performance degradation is noticeable when the workload increases. Another important conclusion is that hosting multiple tiers of the same application on the same server is not optimal.

5.14 Open-source software platforms for private clouds

Private clouds provide a cost effective alternative for very large organizations. A private cloud has essentially the same structural components as a commercial one: the servers, the network, hypervisors running on individual systems, an archive containing disk images of VMs, a front-end for communication with the user, and a cloud control infrastructure. Open-source cloud computing platforms such as Eucalyptus [368], OpenNebula, and Nimbus can be used as a control infrastructure for a private cloud.

Schematically, a cloud infrastructure carries out the following steps to run an application: (i) retrieves the user input from the front-end; (ii) retrieves the disk image of a VM (Virtual Machine) from a repository; (iii) locates a system and requests the hypervisor running on that system to set up a VM; and (iv) invokes the DHCP (see Section 6.1) and the IP bridging software to set up a MAC and IP address for the VM.

Eucalyptus (http://www.eucalyptus.com/) can be regarded as an open-source counterpart of Amazon's EC2. The system supports several operating systems, including: CentOS 5 and 6, RHEL 5 and 6, Ubuntu 10.04 LTS, and 12.04 LTS. The components of the system are:

1. Virtual Machine. Runs under several hypervisors including Xen, KVM, and VMware.
2. Node Controller. Runs on every server/node designated to host a VM and controls the activities of the node. Reports to a cluster controller.
3. Cluster Controller. Controls a number of servers. Interacts with the node controller on each server to schedule requests on that node. Cluster controllers are managed by cloud controller.
4. Cloud Controller. Provides the cloud access to end-users, developers, and administrators. It is accessible through command line tools compatible with EC2 and through a web-based dashboard. Manages cloud resources, makes high-level scheduling decisions, and interacts with cluster controllers.
5. Storage Controller. Provides persistent virtual hard drives to applications. It is the correspondent of EBS. Users can create snapshots from EBS volumes. Snapshots are stored in Walrus and shared across availability zones.
6. Storage Service (Walrus). Provides persistent storage and, similarly to S3, allows users to store objects in buckets.

The system supports a strong separation between the user space and administrator space; users access the system via a web interface, while administrators need root access. The system supports a decentralized resource management of multiple clusters with multiple cluster controllers, but a single head node for handling user interfaces. It implements a distributed storage system called Walrus, the analog of Amazon's S3 system. The procedure to construct a VM is based on the generic one described in [445]:

Table 5.4 A side-by-side comparison of Eucalyptus and OpenNebula.

	Eucalyptus	OpenNebula
Design	Emulate EC2	Customizable
Cloud type	Private	Private
User population	Large	Small
Applications	All	All
Customizability	Administrators limited users	Administrators and users
Internal security	Strict	Loose
User access	User credentials	User credentials
Network access	To cluster controller	–

(i) The *euca2ools* front-end is used to request a VM.
(ii) The VM disk image is transferred to a compute node.
(iii) The disk image is modified for use by the hypervisor on the compute node.
(iv) The compute node sets up network bridging to provide a virtual NIC with a virtual MAC address.
(v) The head node the DHCP is set up with the MAC/IP pair.
(vi) The hypervisor activates the VM.
(vii) The user can now *ssh* directly into the VM.

The system can support a large number of users in a corporate enterprise environment. Users are shielded from the complexity of disk configurations and can choose for their VM from a set of five configurations of available processors, memory, and hard-drive space setup by the system administrators.

Open-Nebula (http://www.opennebula.org/) is a private cloud with users actually logging into the head node to access cloud functions. The system is centralized, and its default configuration uses the NFS filesystem. The procedure to construct a VM consists of several steps: (i) a user signs in to the head node using ssh; (ii) next, it uses the *onevm* command to request a VM; (iii) the VM template disk image is transformed to fit the correct size and configuration within the NFS directory on the head node; (iv) the *oned* daemon on the head node uses *ssh* to log into a compute node; (v) the compute node sets up network bridging to provide a virtual NIC with a virtual MAC; (vi) the files needed by the hypervisor are transferred to the compute node via the NFS; (vii) the hypervisor on the compute node starts the VM; and (viii) the user is able to ssh directly to the VM on the compute node.

According to the analysis in [445], the system is best suited for an operation involving a small-to-medium sized group of trusted and knowledgeable users who are able to configure this versatile system based on their needs.

Table 5.4 summarizes the features of the two systems [445]. Eucalyptus is best suited for a large corporation with its own private cloud because it ensures a degree of protection from user malice and mistakes; OpenNebula is best suited for a testing environment with a few servers.

OpenStack is an open source project started in 2009 at NASA in collaboration with Rackspace (http://www.rackspace.com) to develop a scalable cloud OS for farms of servers using standard hardware. Though recently NASA has moved its cloud infrastructure to AWS, in addition to Rackspace, several other companies including HP, Cisco, IBM, and Red Hat have an interest in *OpenStack*. The current

version of the system supports a wide range of features such as: APIs with rate limiting and authentication, live VM management to run, reboot, suspend, and terminate instances, role-based access control, and the ability to allocate, track, and limit resource utilization. The administrators and the users control their resources using an extensible web application called the *Dashboard*.

5.15 The darker side of virtualization

Can virtualization empower the creators of malware[20] to carry out their mischievous activities with impunity and minimal danger of being detected? How difficult is it to implement such a system? What are the means to prevent this type of malware to be put in place? The answers to these questions are discussed in this section.

It is well understood that, in a layered structure, a defense mechanism at some layer can be disabled by malware running at a layer below it. Thus, the winner in the continuous struggle between the attackers and the defenders of a computing system is the one in control of the lowest layer of the software stack, the one which controls the hardware.

A hypervisor allows a guest OS to run on virtual hardware; the hypervisor offers to the guest operating systems a hardware abstraction and mediates its access to the physical hardware. We argue that a hypervisor is simpler and more compact than a traditional OS, thus, it is more secure; but what if the hypervisor itself is forced to run above another software layer, and, thus, it is prevented from exercising direct control of the physical hardware?

A 2006 paper [274] argues that it is feasible to insert a "rogue hypervisor" between the physical hardware and an OS, as shown in Fig. 5.14(a). Such a rogue hypervisor is called a *Virtual-Machine Based Rootkit* (VMBR). The term *rootkit* refers to malware with a privileged access to a system. The name comes from *root*, the most privileged account on a Unix system, and *kit*, a set of software components.

It is also feasible to insert the VMBR between the physical hardware and a legitimate hypervisor, as in Fig. 5.14(b). Since a VM running under a legitimate hypervisor sees a virtual hardware, the guest OS will not notice any change in the environment. The only trick is to present the legitimate hypervisor with a hardware abstraction, rather than allow it to run on the physical hardware.

Before we address the question how such an insertion is possible, we should point out that in this approach, the malware runs either inside a hypervisor or with the support of a hypervisor. A hypervisor is a very potent engine for the malware: It prevents the software of the guest OS or the application from detecting malicious activities. A VMBR can record key strokes, system states, data buffers sent to, or received from the network, and data to be written to, or read from the disk with impunity; moreover, it can change any data at will.

The only way for a VMBR to take control of a system is to modify the boot sequence and to first load the malware and only then load the legitimate hypervisor, or the OS; this is only possible if the attacker has root privileges. Once the VMBR is loaded, it must also store its image on the persistent storage.

[20] Malware, an abbreviation for *malicious software*, is software designed specifically to circumvent the authorization mechanisms and gain access to a computer system, gather private information, block access to a system, or disrupt the normal operation of a system; computer viruses, worms, spyware, and Trojan horses are examples of malware.

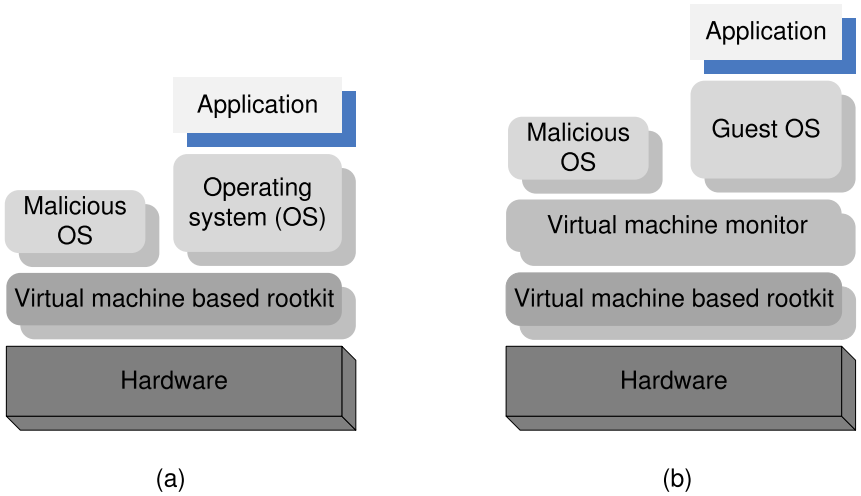

FIGURE 5.14

The insertion of a *Virtual-Machine Based Rootkit* (VMBR) as the lowest layer of the software stack running on the physical hardware; (a) below an OS; (b) below a legitimate hypervisor. The VMBR enables a malicious OS to run surreptitiously and makes it invisible to the genuine or the guest OS and to the application.

The VMBR can enable a separate malicious OS to run surreptitiously and make this malicious OS invisible to the guest OS and to the application running under it. Under the protection of the VMBR, the malicious OS could: (i) observe the data, the events, or the state of the target system; (ii) run services such as spam relays or distributed denial-of-service attacks; or (iii) interfere with the application.

A proof-of-concept VMBRs to subvert Windows XP and Linux and several services based on these two platforms are described in [274]. We should stress that modifying the boot sequence is by no means an easy task, and once an attacker has root privileges she is in total control of a system.

5.16 Virtualization software

Several virtualization software packages, including hypervisors, OS-level virtualization software, and desktop virtualization software, are available. There are two types of hypervisors, native and hosted. The set of *native hypervisors* includes:

Red Hat Virtualization (RHV)—enterprise virtualization based on KVM hypervisor.
Hyper—creates VMs on x86-64 systems running Windows.
z/VM—current version of IBM's VM operating systems.
VMware ESXi—enterprise-class, type-1 hypervisor from VMware.
Oracle VM Server for x86—server virtualization from Oracle Corporation. Incorporates the free, open-source Xen. Supports Windows, Linux, and Solaris guests.

Adeos—Adaptive Domain Environment for Operating Systems is a nanokernel hardware abstraction layer.
XtratuM—bare-metal hypervisor for embedded real-time systems. Available for the instruction sets x86, ARM Cortex-R4F processors, and others.

There are several *hosted independent hypervisors* including: (i) VMware Fusion—software hypervisor developed for Intel-based Macs to run Microsoft Windows, Linux, NetWare, or Solaris on VMs, along with the OS X OS, based on paravirtualization, hardware virtualization, and dynamic recompilation; (ii) PearPC—architecture-independent PowerPC platform emulator for PowerPC operating systems, including pre-Intel versions of OS X, Darwin, and Linux; (iii) Oracle VM VirtualBox—free and open-source hypervisor for x86 computers; and (iv) QEMU (Quick Emulator)—free and open-source hosted hypervisor. There are also *hosted specialized hypervisors* including: (i) coLinux—Cooperative Linux allows Microsoft Windows and the Linux kernel to run simultaneously; (ii) MoM—Mac-on-Mac is a port of Mac-on-Linux for Mac OS X; (iii) Mac-on-Linux—open-source VM for running the classic Mac OS or OS X on PowerPC computers running Linux; (iv) bhyve—a type-1 hypervisor included in FreeBSD running FreeBSD 9+, OpenBSD, NetBSD, Linux and Windows desktop, and Windows Server; and (v) L4Linux—a variant of Linux kernel running virtualized on L4 microkernel; L4Linux kernel runs a service on L4.

5.17 History notes and further readings

Virtual memory was the first application of virtualization concepts to commercial computers; it allowed multiprogramming and eliminated the need to tailor applications to the physical memory available on individual systems. Paging and segmentation are the two mechanisms supporting virtual memory. Paging was developed for the Atlas Computer built in 1959 at the University of Manchester. Independently, in 1961 Burroughs Corporation developed B5000, the first commercial computer with virtual memory; B5000 virtual memory used segmentation rather than paging.

In 1967, IBM introduced 360/67, the first IBM system with virtual memory expected to run on a new OS, called TSS. Before TSS was released, an operating system called CP-67 was created; CP-67 gave the illusion of several standard IBM-360 systems without virtual memory. The first hypervisor supporting full virtualization was the CP-40 system which ran on a S/360-40 that was modified at the IBM Cambridge Scientific Center to support Dynamic Address Translation, a key feature that allowed virtualization. In CP-40, the hardware's supervisor state was virtualized as well, allowing multiple operating systems to run concurrently in separate VM contexts.

Virtualization was driven by the need to share very expensive hardware among a large population of users and applications in the early age of computing. VM/370 system, released in early 1970s for large IBM mainframes, was very successful; it was based on a reimplementation of CP/CMS. In VM/370, a new VM was created for every user, and this VM interacted with the applications. The hypervisor managed hardware resources and enforced the multiplexing of resources. Modern-day IBM mainframes, such as the zSeries line, retain backwards-compatibility with the 1960s-era IBM S/360 line.

Microprocessor development coupled with advances in storage technology contributed to the rapid decrease of hardware costs and led to introduction of personal computers at one end of the spectrum and of large mainframes and massively parallel systems at the other end. The hardware and the operating

systems of 1980s and 1990s gradually limited virtualization and focused instead on efficient multitasking, user interfaces, and support for networking and security problems brought in by interconnectivity.

Advancements in computer and communication hardware and the explosion of the Internet partially due to the success of the World Wide Web in late 1990s renewed the interest in virtualization to support server security and isolation of services. In their review paper, Rosenbloom and Garfinkel write [423]: "hypervisors give OS developers another opportunity to develop functionality no longer practical in today's complex and ossified operating systems, where innovation moves at a geologic pace." Nested virtualization was first discussed in early 1970s by Popek and Goldberg [201,402].

Further readings. The text of Saltzer and Kaashoek [430] is a very good introduction to virtualization principles. Virtual machines are dissected in a paper by Smith and Nair [450], and architectural principles for virtual computer systems are analyzed in [200,201].

An insightful discussion of hypervisors is provided by the paper of Rosenblum and Garfinkel [423]. Several papers [48,345,346] discuss in depth the Xen hypervisor and analyze its performance, while [519] is a code repository for Xen. The Denali system is presented in [514].

Modern systems such as Linux Vserver (http://linux-vserver.org/), OpenVZ (Open VirtualiZation) [376], FreeBSD Jails [415], and Solaris Zones [403] implement *OS-level virtualization technologies*. Reference [385] compares the performance of two virtualization techniques with a standard OS.

A 2001 paper [100] argues that virtualization allows new services to be added without modifying the OS. Such services are added below the OS level, but this process creates a semantic gap between the VMs and these services. Reflections on the design of hypervisors are the subject of [101] and a discussion of Xen is reported in [108]. The state of the art and the future of nested virtualization are the subject of [127]. An implementation of nested virtualization for KVM is discussed in [57]. [541] surveys security issues in virtual systems and [283] covers reliability in virtual infrastructures. Virtualization technologies in HPC are analyzed in [410], and [452] provides a critical view on virtualization. [508] reports on IBM virtualization strategies.

5.18 Exercises and problems

Problem 1. Identify the milestones in the evolution of operating systems during the half century from 1960 to 2010 and comment on the statement from [423] "Hypervisors give OS developers another opportunity to develop functionality no longer practical in today's complex and ossified operating systems, where innovation moves at a geologic pace."

Problem 2. Virtualization simplifies the use of resources, isolates users from one another, and supports replication and mobility but exacts a price in terms of performance and cost. Analyze each one of these aspects for: (i) memory virtualization, (ii) processor virtualization, and (iii) virtualization of a communication channel.

Problem 3. Virtualization of the processor combined with virtual memory management pose multiple challenges; analyze the interaction of interrupt handling and paging.

Problem 4. In Section 5.2, we stated that a hypervisor is a much simpler and better specified system than a traditional OS. Hypervisor vulnerability is reduced because the systems expose a much smaller number of privileged functions. Compare the number of lines of code and of system calls for several operating systems including Linux, Solaris, FreeBSD, Unbuntu, AIX, and Windows with the corresponding figures for several system VMs.

Problem 5. In Section 5.4 we state that a hypervisor for a processor with a given ISA can be constructed if the set of *sensitive instructions* is a subset of the privileged instructions of that processor. Identify the set of sensitive instructions for the x86 architecture and discuss the problem each one of these instructions poses.

Problem 6. Table 5.3 summarizes the effects of Xen network performance optimization reported in [346]. The send-data rate of a guest domain is improved by a factor of more than four, while the improvement of the receive data rate is very modest. Identify several possible reasons for this discrepancy.

Problem 7. VMware EX Server supports full virtualization of x86 architecture. Analyze how VMware provides the functions discussed in Table 5.2 for Xen.

Problem 8. In 2012, Intel and HP announced that *Itanium* architecture will be discontinued. Review the architecture discussed in Section 5.12, and identify several possible reasons for this decision.

Problem 9. Read [385] and analyze the results of performance comparison discussed in Section 5.13.