UNIT – 2

Virtual Machines and Virtualization of Clusters and Data Centers

1. The massive usage of virtual machines (VMs) opens up new opportunities for parallel, cluster grid, cloud and distributed computing. Virtualization enables the users to share expensive hardware resources by multiplexing (i.e., multiple analog/digital are combined into one signal over a shared medium [2]) VMs on the same set of hardware hosts like servers or data centers.
2. **Implementation Levels of Virtualization**: Virtualization is a concept by which several VMs are multiplexed into the same hardware machine. The purpose of a VM is to enhance resource sharing by many users and improve computer performance in terms of resource utilization and application flexibility. Hardware resources (CPU, memory, I/O devices etc.) or software resources (OS and apps) can be virtualized at various layers of functionality.

The main idea is to separate hardware from software to obtain greater efficiency from the system. Ex: Users can gain access to more memory by this concept of VMs. With sufficient storage, any computer platform can be installed in another host computer [1], even if processors' usage and operating systems are different.

1. **Levels of Virtualization Implementation**: A traditional computer system runs with a host OS specially adjusted for its hardware architecture. This is depicted in Figure 3.1a [1].

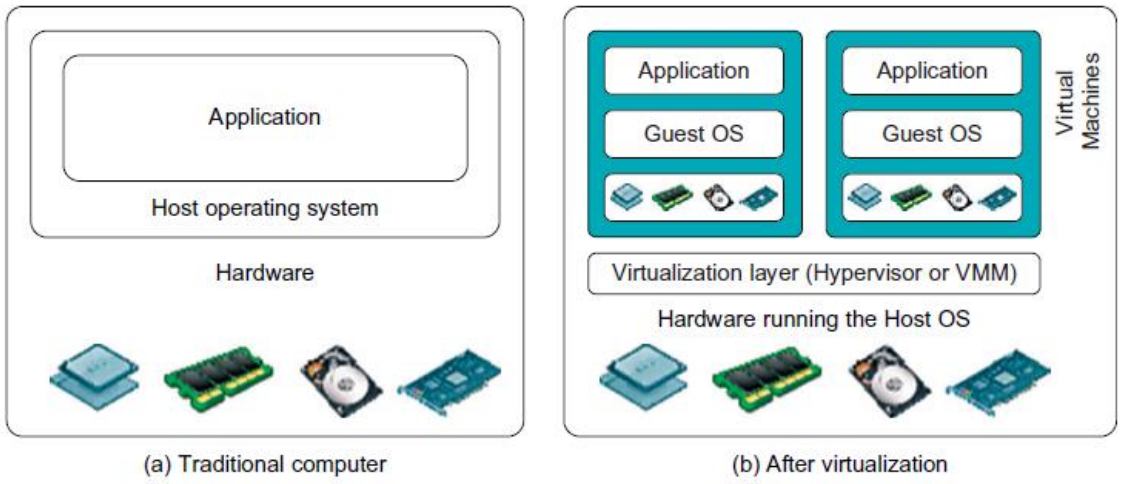

FIGURE 3.1

The architecture of a computer system before and after virtualization, where VMM stands for virtual machine monitor.

After virtualization, different user apps managed by their own OS (i.e., guest OS) can run on the same hardware, independent of the host OS. This is often done by adding a virtualization layer as shown in Figure 3.1b [2].

This virtualization layer is called VM Monitor or hypervisor. The VMs can be seen in the upper boxes where apps run on their own guest OS over a virtualized CPU, memory and I/O devices.

2. The main function of the software layer for virtualization is to virtualize the physical hardware of a host machine into virtual resources to be saved by the VMs. The virtualization software creates the abstract of VMs by introducing a virtualization layer at

various levels of a computer. General virtualization layers include the instruction set architecture (ISA) level, hardware level, OS level, library support level, and app level. This can be seen in Figure 3.2 [1]. The levels are discussed below.

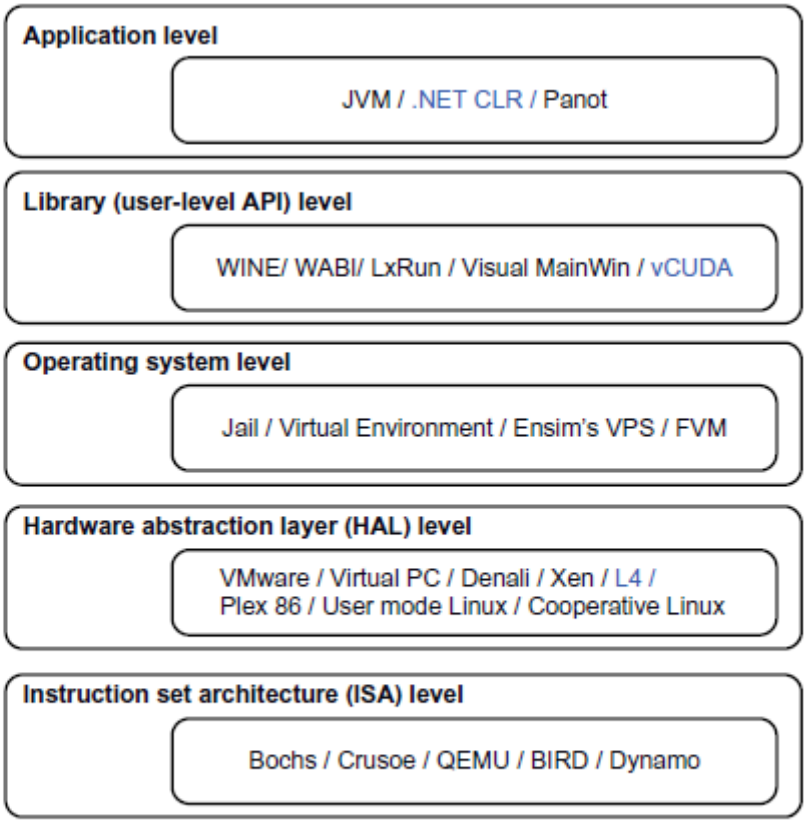

FIGURE 3.2

Virtualization ranging from hardware to applications in five abstraction levels.

1. **Instruction Set Architecture Level**: At the ISA level, virtualization is performed by emulation (imitate) of the given ISA by the ISA of the host machine. Ex: MIPS binary code can run on an x86-based host machine with the help of ISA simulation. Instruction emulation leads to virtual ISAs created on any hardware machine.

   Basic level of emulation can be traced at code interpretation. An interpreter (line-by-line compiler) program works on the instructions one-by-one and this process is slow. To speedup, *dynamic binary translation* can be used where it translates blocks of dynamic source instructions to target instructions. The basic blocks can also be extended to program traces or super blocks to increase translation efficiency. This emulation requires binary translation and optimization. Hence, a Virtual-ISA requires a processor specific translation layer to the compiler.

2. **Hardware Abstraction Level**: Hardware level virtualization is performed on the bare hardware. This approach generates a virtual hardware environment and processes the hardware in a virtual manner. The idea is to virtualize the resources of a computer by utilizing them concurrently. Ex: IBM Xen hypervisor (VMM) runs Linux or other guest OS applications. [Discussed later]

3. **OS Level:** This refers to an abstraction layer between the OS and the user apps. The OS level virtualization creates isolated *containers* on a single physical server and OS instances to utilize software and hardware in data centers. The containers behave like real servers. OS level virtualization is used in creating virtual hosting environments to allocate hardware

resources among a large number of 'distrusting' users. It can also be used to indirectly merge server hardware by moving resources on different hosts into different containers or VMs on one server.

4. **NOTE:** ***Containers*** [3] use the host operating system as their base, and not the hypervisor. Rather than virtualizing the hardware (which requires full virtualized operating system images for each guest), containers virtualize the OS itself, sharing the host OS kernel and its resources with both the host and other containers.

5. **Library Support Level**: Most applications use APIs exported by user-level libraries rather than lengthy system calls by the OS. Virtualization with library interfaces is possible by controlling the communication link between apps and the rest of the system through API hooks.

Ex: (a) **Wine** (recursive acronym for *Wine Is Not an Emulator*) is a free and open source compatibility layer software application that aims to allow applications designed for MS-Windows to run on Linux OS.

(b) vCUDA by NVIDIA. (CUDA – No acronym)

NOTE: Library [4] in computing is a collection of non-volatile (stable) resources used by computer programs to develop software. These include configuration data (organised data), documentation, help data, message templates, code subroutines classes and specifications.

6. **User-App Level**: An app level virtualization brings out a real VM; this process is also known as process level virtualization. Generally HLL VMs are used where virtualization layer is an app above the OS; it can run programs written and compiled to an abstract machine definition. Ex: JVM and .NET CLR (Common Language Runtime).

Other forms of app level virtualization are app isolation, app sandboxing or app streaming. Here, the app is wrapped in a layer and is isolated from the host OS and other apps. This makes the app more much easier to distribute and remove from user workstations. Ex: LANDesk (an app virtualization platform) – this installs apps as self-contained, executable files in an isolated environment. No actual installation is required and no system modifications are needed.

Table 3.1 Relative Merits of Virtualization at Various Levels (More "X"'s Means Higher Merit, with a Maximum of 5 X's)

Level of Implementation	Higher Performance	Application Flexibility	Implementation Complexity	Application Isolation
ISA	X	XXXXX	XXX	XXX
Hardware-level virtualization	XXXXX	XXX	XXXXX	XXXX
OS-level virtualization	XXXXX	XX	XXX	XX
Runtime library support	XXX	XX	XX	XX
User application level	XX	XX	XXXXX	XXXXX

Note from Table 3.1 [1] that hardware and OS support will yield the highest performance. At the same time, the hardware and app levels are most expensive to implement. User isolation is difficult to archive and ISA offers best flexibility.

3. **VMM Design Requirement and Providers**: As seen before, hardware-level virtualization inserts a layer between real hardware and traditional OS. This layer (VMM/hypervisor) manages the hardware resources of the computer effectively. By the usage of VMM, different traditional operating systems can be used with the same set of hardware simultaneously.

4. **Requirements for a VMM**:

a. For programs, a VMM should provide an identical environment, same as the original machine.
b. Programs running in this environment should show only minor decreases in speed.
c. A VMM should be in complete control of the system resources.

Some differences might still be caused due to availability of system resources (more than one VM is running on the same system) and differences caused by timing dependencies.

The hardware resource requirements (like memory) of each VM is reduced, but the total sum of them is greater that of the real machine. This is needed because of any other VMs that are concurrently running on the same hardware.

A VMM should demonstrate efficiency in using the VMs. To guarantee the efficiency of a VMM, a statistically dominant subset of the virtual processor's instructions needs to be executed directly by the real processor with no intervention by the VMM. A comparison can be seen in Table 3.2 [1]:

Table 3.2 Comparison of Four VMM and Hypervisor Software Packages

Provider and References	Host CPU	Host OS	Guest OS	Architecture
VMware Workstation [71]	x86, x86-64	Windows, Linux	Windows, Linux, Solaris, FreeBSD, Netware, OS/2, SCO, BeOS, Darwin	Full Virtualization
VMware ESX Server [71]	x86, x86-64	No host OS	The same as VMware Workstation	Para-Virtualization
Xen [7,13,42]	x86, x86-64, IA-64	NetBSD, Linux, Solaris	FreeBSD, NetBSD, Linux, Solaris, Windows XP and 2003 Server	Hypervisor
KVM [31]	x86, x86-64, IA-64, S390, PowerPC	Linux	Linux, Windows, FreeBSD, Solaris	Para-Virtualization

The aspects to be considered here include (1) The VMM is responsible for allocating hardware resources for programs; (2) a program can't access any resource that has not been allocated to it; (3) at a certain juncture, it is not possible for the VMM to regain control of the resources already allocated. Note that all processors might not satisfy these requirements of a VMM.

A VMM is tightly related to the architectures of the processors. It is difficult to implement a VMM on some types of processors like x86. If a processor is not designed to satisfy the

requirements of a VMM, the hardware should be modified – this is known as hardware assisted virtualization.

5. **Virtualization Support at the OS Level**: CC is transforming the computing landscape by shifting the hardware and management costs of a data center to third parties, like banks. The challenges of CC are: (a) the ability to use a variable number of physical machines and VM instances depending on the needs of the problem. Ex: A work may need a single CPU at an instance but multi-CPUs at another instance (b) the slow operation of instantiating new VMs.

As of now, new VMs originate either as fresh boots or as replicates of a VM template – unaware of the current status of the application.

6. **Why OS Level Virtualization** (Disadvantages of hardware level virtualization):

a. It is slow to initiate a hardware level VM since each VM creates its own image from the beginning.

b. Redundancy content is high in these VMs.

c. Slow performance and low density

d. Hardware modifications maybe needed

To provide a solution to all these problems, OS level virtualization is needed. It inserts a virtualization layer inside the OS to partition the physical resources of a system. It enables multiple isolated VMs within a single OS kernel. This kind of VM is called a Virtual Execution Environment (VE) or Virtual Private System or simply a **container**. From the user's point of view, a VE/container has its own set of processes, file system, user accounts, network interfaces (with IP addresses), routing tables, firewalls and other personal settings.

Note that though the containers can be customized for different people, they share the same OS kernel. Therefore this methodology is also called single-OS image virtualization. All this can be observed in Figure 3.3 [1].

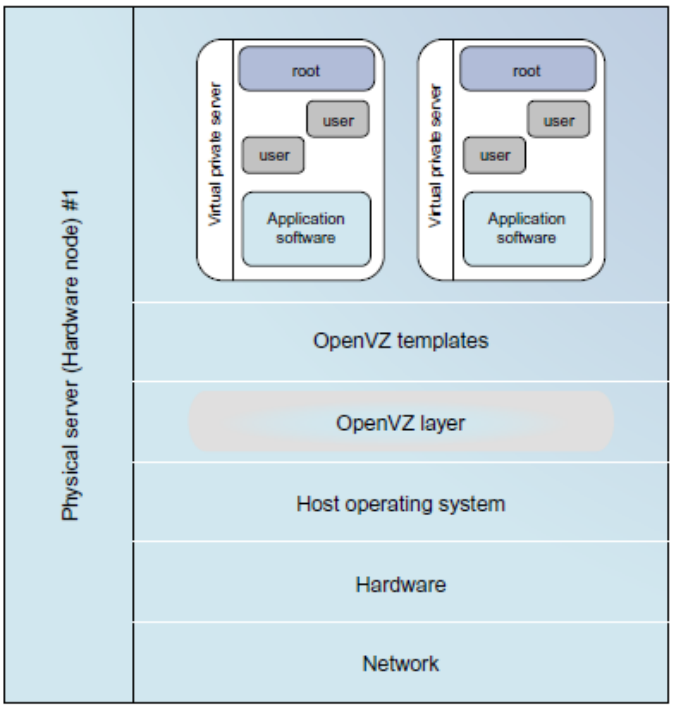

FIGURE 3.3

The OpenVZ virtualization layer inside the host OS, which provides some OS images to create VMs quickly.

7. **Advantages of OS Extensions**:

a. VMs at the OS level have minimal start-up shutdown costs, low resource requirements and high scalability.

b. For an OS level VM, the VM and its host environment can synchronise state changes

These can be achieved through two mechanisms of OS level virtualization:

a. All OS level VMs on the same physical machine share a single OS kernel

b. The virtualization layer can be designed in way that allows processes in VMs can access as many resources as possible from the host machine, but can never modify them.

8. **Disadvantages of OS Extension:** The main disadvantage of OS extensions is that all VMs at OS level on a single container must have the same kind of guest OS. Though different OS level VMs may have different OS distributions (Win XP, 7, 10), they must be related to the same OS family (Win). A Windows distribution can't run on a Linux based container.

As we can observe in Figure 3.3, the virtualization layer is inserted inside the OS to partition the hardware resources for multiple VMs to run their applications in multiple virtual environments. To implement this OS level virtualization, isolated execution environments (VMs) should be created based on a single OS kernel. In addition, the access requests from a VM must be redirected to the VM's local resource partition on the physical machine. For example, '*chroot*' command in a UNIX system can create several virtual root directories within an OS that can be used for multiple VMs.

To implement the virtual root directories' concept, there exist two ways: (a) duplicating common resources to each VM partition or (b) sharing most resources with the host

environment but create private copies for the VMs on demand. It is to be noted that the first method incurs (brings up) resource costs and burden on a physical machine. Therefore, the second method is the apparent choice.

9. **Virtualization on Linux or Windows Platforms**: Generally, the OS-level virtualization systems are Linux-based. Windows based virtualization platforms are not much in use. The Linux kernel offers an abstraction layer to allow software processes to with and operate on resources without knowing the hardware details. Different Linux platforms use patched kernels to provide special support for extended functionality.

Note that most Linux platforms are not tied to a special kernel. In such a case, a host can run several VMs simultaneously on the same hardware. Examples can be seen in Table 3.3 [1].

10. **Middleware Support for Virtualization**: This is the other name for Library-level Virtualization and is also known as user-level Application Binary Interface or API emulation. This type of virtualization can create execution environments for running alien (new/unknown) programs on a platform rather than creating a VM to run the entire OS. The key functions performed here are API call interception and remapping (assign a function to a key).

Table 3.3 Virtualization Support for Linux and Windows NT Platforms

Virtualization Support and Source of Information	Brief Introduction on Functionality and Application Platforms
Linux vServer for Linux platforms (http://linux-vserver.org/)	Extends Linux kernels to implement a security mechanism to help build VMs by setting resource limits and file attributes and changing the root environment for VM isolation
OpenVZ for Linux platforms [65]; http://ftp.openvz.org/doc/OpenVZ-Users-Guide.pdf)	Supports virtualization by creating *virtual private servers (VPSes)*; the VPS has its own files, users, process tree, and virtual devices, which can be isolated from other VPSes, and checkpointing and live migration are supported
FVM (Feather-Weight Virtual Machines) for virtualizing the Windows NT platforms [78])	Uses system call interfaces to create VMs at the NY kernel space; multiple VMs are supported by virtualized namespace and copy-on-write

3. **Virtualization Structures/Tools and Mechanisms**: It should be noted that there are three classes of VM architecture [Page 1]. Before virtualization, the OS manages the hardware. After virtualization, a virtualization layer is inserted between the hardware and the OS. Here, the virtualization layer is responsible for converting parts of real hardware into virtual hardware. Different operating systems like Windows and Linux can run simultaneously on the same machine in this manner. Depending on the position of the virtualization layer, several

classes of VM architectures can be framed out: Hypervisor Architecture, para-virtualization and host-based virtualization.

1. **Hypervisor and Xen Architecture**: The hypervisor (VMM) supports hardware level virtualization on bare metal devices like CPU, memory, disk and network interfaces. The hypervisor software exists between the hardware and its OS (platform). The hypervisor provides **hypercalls** for the guest operating systems and applications. Depending on the functionality, a hypervisor can assume **micro-kernel** architecture like MS Hyper-V or **monolithic** hypervisor architecture like the VMware ESX for server virtualization.

   Hypercall: A hypercall is a software trap from a domain to the hypervisor, just as a syscall is a software trap from an application to the kernel. Domains will use hypercalls to request privileged operations like updating page tables.

   Software Trap: A trap, also known as an exception or a fault, is typically a type of synchronous interrupt caused by an exceptional condition (e.g., breakpoint, division by zero, invalid memory access). A trap usually results in a switch to kernel mode, wherein the OS performs some action before returning control to the originating process. A trap in a system process is more serious than a trap in a user process and might be fatal. The term trap might also refer to an interrupt intended to initiate a context switch to a monitor program or debugger.

   Domain: It is a group of computers/devices on a network that are administered as a unit with common rules and procedures. Ex: Within the Internet, all devices sharing a common part of the IP address are said to be in the same domain.

   Page Table: A page table is the data structure used by a virtual memory system in an OS to store the mapping between virtual addresses and physical addresses.
   Kernel: A kernel is the central part of an OS and manages the tasks of the computer and hardware like memory and CPU time.

   Monolithic Kernel: These are commonly used by the OS. When a device is needed, it is added as a part of the kernel and the kernel increases in size. This has disadvantages like faulty programs damaging the kernel and so on. Ex: Memory, processor, device drivers etc.

   Micro-kernel: In micro-kernels, only the basic functions are dealt with – nothing else. Ex: Memory management and processor scheduling. It should also be noted that OS can't run only on a micro-kernel, which slows down the OS.
   [SIM – Micro SIM]

2. The size of the hypervisor code of a micro-kernel hypervisor is smaller than that of monolithic hypervisor. Essentially, a hypervisor must be able to convert physical devices into virtual resources dedicated for the VM usage.

3. **Xen Architecture**: It is an open source hypervisor program developed by Cambridge University. Xen is a micro-kernel hypervisor, whose policy is implemented by Domain 0.

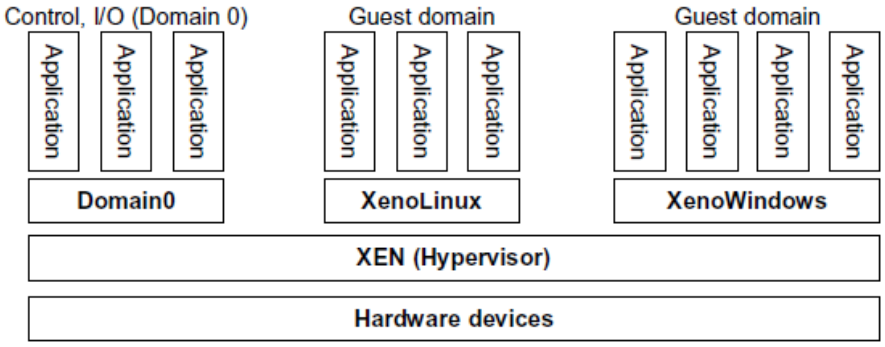

FIGURE 3.5

The Xen architecture's special domain 0 for control and I/O, and several guest domains for user applications.

As can be seen in Figure 3.5 [1], Xen doesn't include any device drivers; it provides a mechanism by which a guest-OS can have direct access to the physical devices. The size of Xen is kept small, and provides a virtual environment between the hardware and the OS. Commercial Xen hypervisors are provided by Citrix, Huawei and Oracle.

The core components of Xen are the hypervisor, kernel and applications. Many guest operating systems can run on the top of the hypervisor; but it should be noted that one of these guest OS controls the others. This guest OS with the control ability is called Domain 0 – the others are called Domain U. Domain 0 is first loaded when the system boots and can access the hardware directly and manage devices by allocating the hardware resources for the guest domains (Domain U).

Say Xen is based on Linux and its security level is some C2. Its management VM is named as Domain 0, which can access and manage all other VMs on the same host. If a user has access to Domain 0 (VMM), he can create, copy, save, modify or share files and resources of all the VMs. This is a huge advantage for the user but concentrating all the resources in Domain 0 can also become a privilege for a hacker. If Domain 0 is hacked, through it, a hacker can control all the VMs and through them, the total host system or systems. Security problems are to be dealt with in a careful manner before handing over Xen to the user.

A machine's lifetime can be thought of as a straight line that progresses monotonically (never decreases or increases) as the s/w executes. During this time, executions are made, configurations are changed, and s/w patches can be applied. VM is similar to tree in this environment; execution can go into N different branches where multiple instances of VM can be done in this tree at any time. VMs can also be allowed to rollback to a particular state and rerun from the same point.

4. **Binary Translation with Full Virtualization**: Hardware virtualization can be categorised into two categories: full virtualization and host-based virtualization.

Full Virtualization doesn't need to modify the host OS; it relies upon binary translation to trap and to virtualize certain sensitive instructions. Normal instructions can run directly on the host OS. This is done to increase the performance overhead – normal instructions are carried out in the normal manner, but the difficult and precise executions are first discovered using a trap and executed in a virtual manner. This is done to improve the security of the system and also to increase the performance.

Binary Translation of Guest OS Requests Using a VMM:

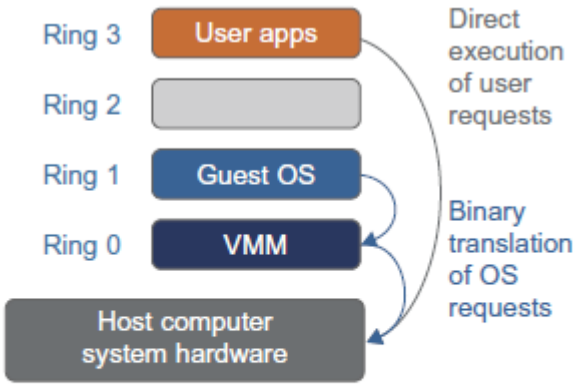

FIGURE 3.6

Indirect execution of complex instructions via binary translation of guest OS requests using the VMM plus direct execution of simple instructions on the same host.

This approach is mainly used by VMware and others. As it can be seen in Figure 3.6 [1], the VMware puts the VMM at Ring 0 and the guest OS at Ring 1. The VMM scans the instructions to identify complex and privileged instructions and trap them into the VMM, which emulates the behaviour of these instructions. Binary translation is the method used for emulation (A => 97 => 01100001) [5]. Note that full virtualization combines both binary translation and direct execution. The guest OS is totally decoupled from the hardware and run virtually (like an emulator).

Full virtualization is ideal since it involves binary translation and is time consuming. Binary translation also is cost consuming but it increases the system performance. (Same as 90% of the host).

In a **host-based** virtualization system both host and guest OS are used and a virtualization layer is built between them. The host OS is still responsible for managing the hardware resources. Dedicated apps might run on the VMs and some others can run on the host OS directly. By using this methodology, the user can install the VM architecture without modifying the host OS. The virtualization software can rely upon the host OS to provide device drivers and other low level services. Hence the installation and maintenance of the VM becomes easier.

Another advantage is that many host machine configurations can be perfectly utilized; still four layers of mapping exist in between the guest and host operating systems. This may hinder the speed and performance, in particular when the ISA (Instruction Set Architecture) of a guest OS is different from that of the hardware – binary translation MUST be deployed. This increases in time and cost and slows the system.

5. **Para-Virtualization with Compiler Support**: Para-Virtualization modifies the guest operating systems; a para-virtualized VM provides special APIs which take up user apps needing those changes. Para-virtualization tries to reduce the virtualization burden/extra-work to improve the performance – this is done by modifying only the guest OS kernel. This can be seen in Figure 3.7 [1].

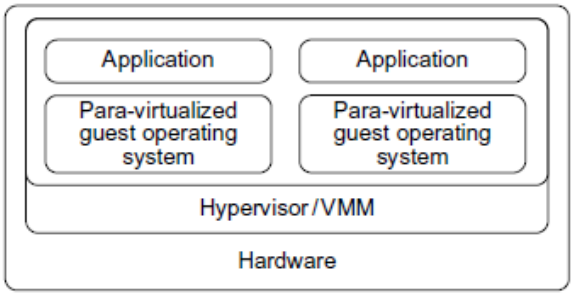

FIGURE 3.7

Para-virtualized VM architecture, which involves modifying the guest OS kernel to replace nonvirtualizable instructions with hypercalls for the hypervisor or the VMM to carry out the virtualization

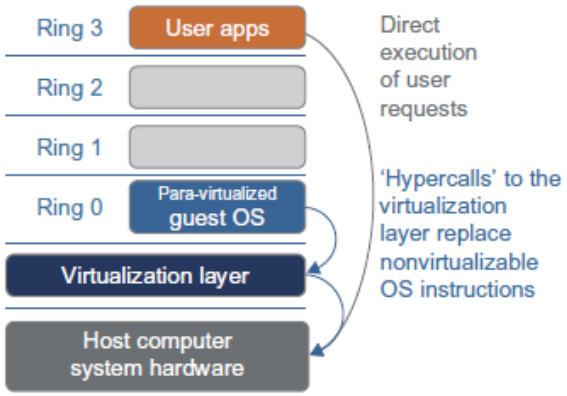

FIGURE 3.8

The use of a para-virtualized guest OS assisted by an intelligent compiler to replace nonvirtualizable OS instructions by hypercalls.

Ex: In a typical para-virtualization architecture, which considers an x86 processor, a virtualization layer is inserted between h/w and OS. According to the x86 'ring definition' the virtualization layer should also be installed at Ring 0. In Figure 3.8 [1], we can notice that para-virtualization replaces instructions that cannot be virtualized with hypercalls (placing a trap) that communicate directly with the VMM. Notice that if a guest OS kernel is modified for virtualization, it can't run the hardware directly – that should be done through the virtualization layer.

6. **Disadvantages of Para-Virtualization**: Although para-virtualization reduces the overhead, it has other problems. Its compatibility (suitability) and portability can be in doubt because it has to support both the modified guest OS and the host OS as per requirements. Also, the maintenance cost of para-virtualization is high since it may require deep kernel modifications. Finally, the performance advantage of para-virtualization is not stable – it varies as per the workload. But compared with full virtualization, para-virtualization is more easy and practical since binary translation is not much considered. Many products utiliza para-virtualization to overcome the less speed of binary translation. Ex: Xen, KVM, VMware ESX.

7. **Note**: Kernel based VM (**KVM**): This is a Linux para-virtualization system – it is a part of the Linux kernel. Memory management and scheduling activities are carried out by the existing Linux kernel. Other activities are taken care of by the KVM and this methodology makes it easier to handle than the hypervisor. Also note that KVM is hardware assisted para-virtualization tool, which improves performance and supports unmodified guest operating systems like Windows, Linux, Solaris and others.

8. **Virtualization of CPU, Memory and I/O Devices**: Processors employ a special running mode and instructions, known as hardware-assisted virtualization. Through this, the VMM and guest OS run in different modes; all sensitive instructions of the guest OS and its apps are caught by the 'trap' in the VMM.

1. **H/W Support for Virtualization**: Modern operating systems and processors permit multiple processes to run simultaneously. A protection mechanism should exist in the processor so that all instructions from different processes will not access the hardware directly – this will lead to a system crash.

   All processors should have at least two modes – user and supervisor modes to control the access to the hardware directly. Instructions running in the supervisor mode are called privileged instructions and the others are unprivileged.

   Ex: VMware Workstation

2. **CPU Virtualization**: A VM is a duplicate of an existing system; majority of instructions are executed by the host processor. Unprivileged instructions run on the host machine directly; other instructions are to be handled carefully. These critical instructions are of three types: privileged, control-sensitive and behaviour-sensitive.

   Privileged=> Executed in a special mode and are trapped if not done so.
   Control-Sensitive=> Attempt to change the configuration of the used resources
   Behaviour-Sensitive=> They have different behaviours in different situations (high load or storage or capacity)

   A CPU is VZ only if it supports the VM in the CPU's user mode while the VMM runs in a supervisor's mode. When the privileged instructions are executed, they are trapped in the VMM. In this case, the VMM acts as a mediator between the hardware resources and different VMs so that correctness and stability of the system are not disturbed. It should be noted that not all CPU architectures support VZ.

   Process:

   - System call triggers the 80h interrupt and passes control to the OS kernel.
   - Kernel invokes the interrupt handler to process the system call
   - In Xen, the 80h interrupt in the guest OS concurrently causes the 82h interrupt in the hypervisor; control is passed on to the hypervisor as well.
   - After the task is completed, the control is transferred back to the guest OS kernel.

3. **Hardware Assisted CPU VZ**: Since full VZ or para-VZ is complicated, this new methodology tries to simplify the situation. Intel and AMD add an additional mode called privilege mode level to the x86 processors. The OS can still run at Ring 0 and hypervisor at Ring 1. Note that all privileged instructions are trapped at the hypervisor. Hence, no modifications are required in the VMs at OS level.

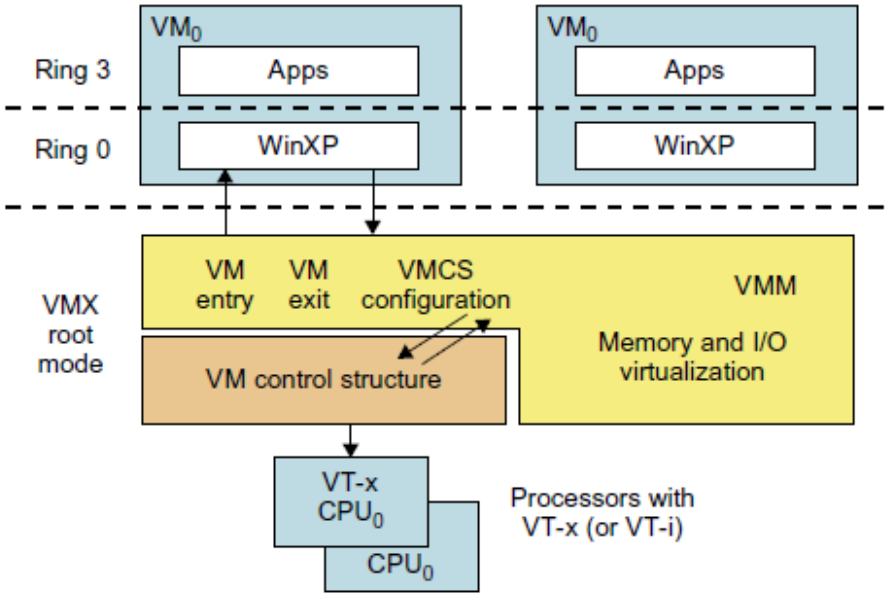

FIGURE 3.11

Intel hardware-assisted CPU virtualization.

VMCS=> VM Control System

VMX=> A virtual router

4. **Memory Virtualization**: In the **traditional** methodology, the OS maintains mappings between virtual memory to machine memory (MM) using page tables, which is a one-stage mapping from virtual memory to MM.

Virtual memory is a feature of an operating system (OS) that allows a computer to compensate for shortages of physical memory by temporarily transferring pages of data from random access memory (**RAM**) to disk storage.

Machine Memory [6] is the upper bound (threshold) of the physical memory that a host can allocate to the VM. All modern x86 processors contain memory management unit (MMU) and a translation look-aside buffer (TLB) to optimize (use in the best way) the virtual memory performance.

In a **virtual execution environment**, virtual memory VZ involves sharing the physical system memory in RAM and dynamically allocating it to the physical memory of the VMs.

Stages:

- Virtual memory to physical memory
- Physical memory to machine memory.

Other Points: MMU should be supported, guest OS controls to monitor mapping of virtual addresses to physical memory address of the VMs. All this is depicted in Figure 3.12 [1].

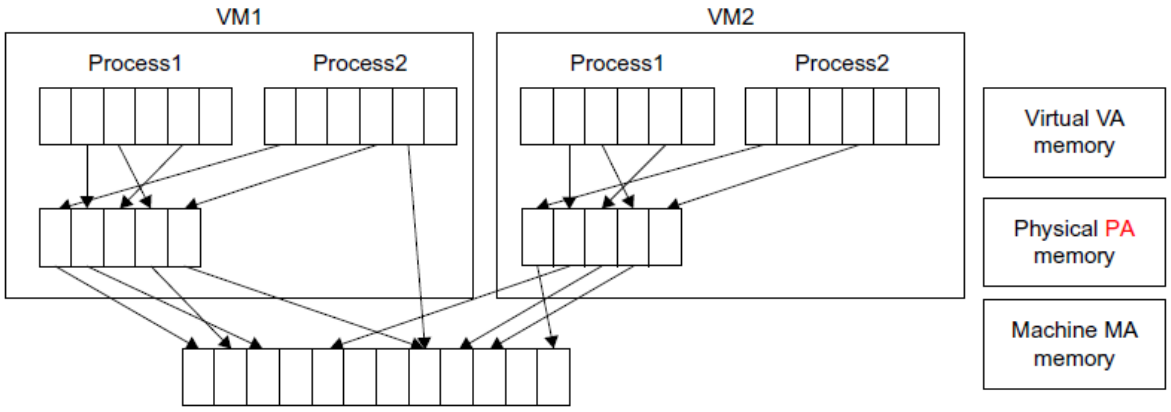

FIGURE 3.12

Two-level memory mapping procedure.

VA-Virtual Address; PA-Physical Address; MA-Machine Address

Each page table of a guest OS has a page table allocated for it in the VMM. The page table in the VMM which handles all these is called a **shadow page table**. As it can be seen all this process is nested and inter-connected at different levels through the concerned address. If any change occurs in the virtual memory page table or TLB, the shadow page table in the VMM is updated accordingly.

5. **I/O Virtualization**: This involves managing of the routing of I/O requests between virtual devices and shared physical hardware. The there are three ways to implement this are full device emulation, para-VZ and direct I/O.

- **Full Device Emulation**: This process emulates well-known and real-world devices. All the functions of a device or bus infrastructure such as device enumeration, identification, interrupts etc. are replicated in the software, which itself is located in the VMM and acts as a virtual device. The I/O requests are trapped in the VMM accordingly. The emulation approach can be seen in Figure 3.14 [1].

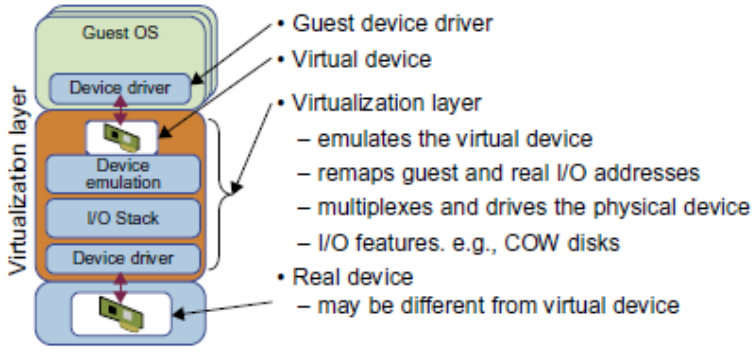

FIGURE 3.14

Device emulation for I/O virtualization implemented inside the middle layer that maps real I/O devices into the virtual devices for the guest device driver to use.

- **Para-VZ**: This method of I/O VZ is taken up since software emulation runs slower than the hardware it emulates. In para-VZ, the frontend driver runs in Domain-U; it manages the requests of the guest OS. The backend driver runs in Domain-0 and is responsible

for managing the real I/O devices. This methodology (para) gives more performance but has a higher CPU overhead.

- **Direct I/O VZ**: This lets the VM access devices directly; achieves high performance with lower costs. Currently, it is used only for the mainframes.

Ex: **VMware Workstation for I/O VZ:** ***NIC*=> *Network Interface Controller***

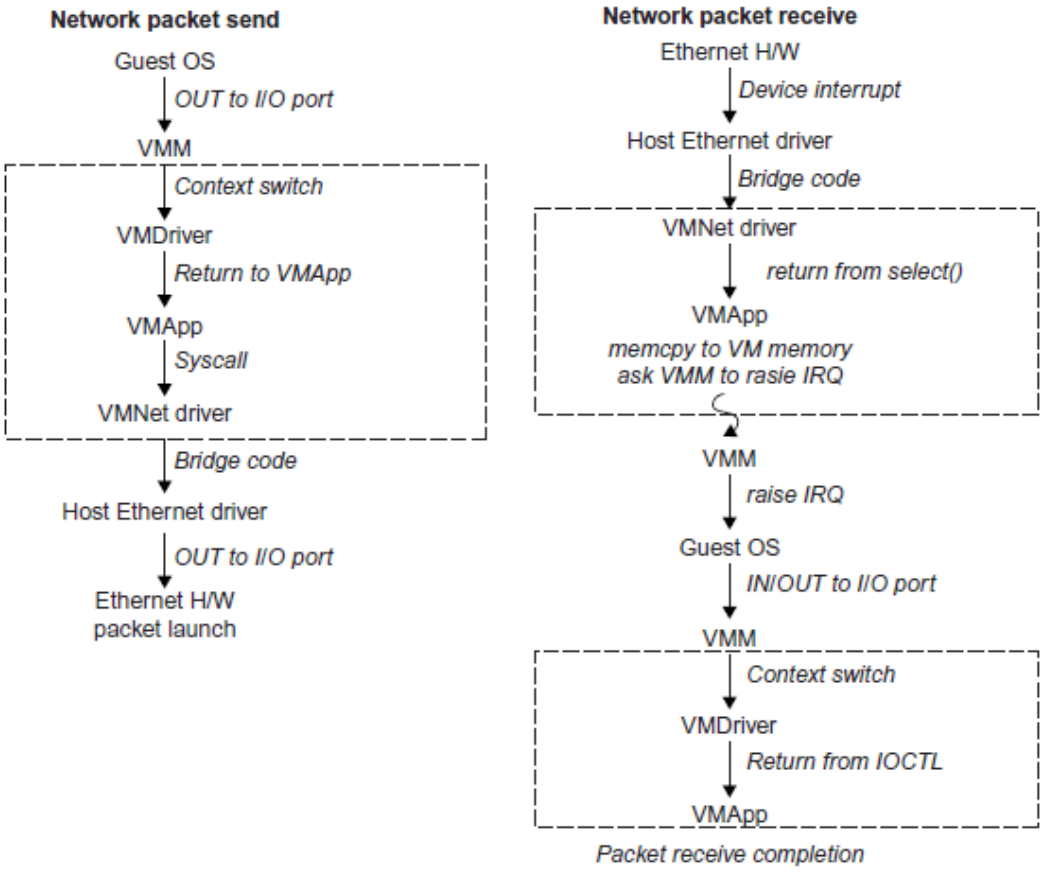

FIGURE 3.15

Functional blocks involved in sending and receiving network packets.

The vCUDA for Virtualization of General-Purpose GPU

It leverages the high performance of GPUs to run compute-intensive applications on host operating systems. However, it is difficult to run CUDA applications on hardware-level VMs directly. vCUDA virtualizes the CUDA library and can be installed on guest OSes. When CUDA applications run on a guest OS and issue a call to the CUDA API, vCUDA intercepts the call and redirects it to the CUDA API running on the host OS

The vCUDA library resides in the guest OS as a substitute for the standard CUDA library. It is responsible for intercepting and redirecting API calls from the client to the stub. when a CUDA application in the guest OS allocates a device's memory the vGPU can return a local virtual address to the application and notify the remote stub to allocate the real device

memory, and the vGPU is responsible for storing the CUDA API flow. The vCUDA stub receives CUDA application CUDA library

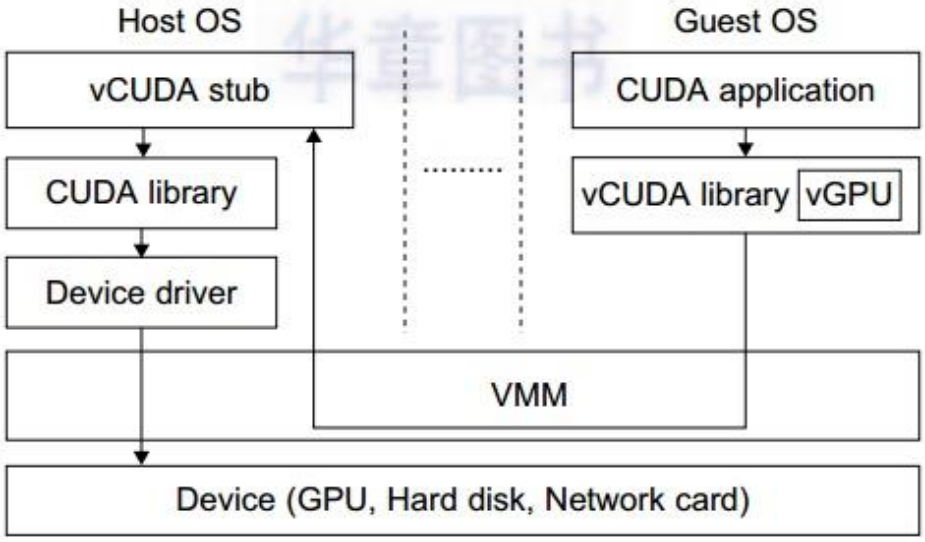

FIGURE 3.4

Basic concept of the vCUDA architecture.

and interprets remote requests and creates a corresponding execution context for the API calls from the guest OS, then returns the results to the guest OS. The vCUDA stub also manages actual physical resource allocation