Hypervisors and Virtual Machines

Implementation Insights on the x86 Architecture

DON REVELLE

Don is a performance engineer and Linux systems/kernel programmer, specializing in high-volume UNIX, Web, virtualization, and TCP/IP networking performance. He has 15 years of UNIX systems engineering and programming experience with a few large high-profile companies in the finance and Internet news media sectors.
don@js-objects.com

Hypervisor and virtualization technology is used to drive cloud computing, server consolidation, clustering, and high availability solutions. The x86 processor line is now the dominant platform for virtualization. Although the x86 processor has a few virtualization challenges, many solutions have been architected. This article's purpose is to briefly explore the principles and operational concepts behind these solutions.

VMware, Microsoft Hyper-V, XEN, KVM, and other Linux-based hypervisors are among the growing number of popular implementations. Most hypervisors use optimized forms of binary translation, paravirtualization, or CPU-supported solutions for full virtualization implementation.

The Challenges of Virtualization and the Popek and Goldberg Principles

Some of the main challenges of x86 virtualization are efficient OS isolation and virtualization overhead. Operating systems expect to execute with unchallenged privilege in order to support their own internal services and the services they offer to user processes. In order to share the x86 CPU with multiple operating systems, an intermediary protocol and a privilege mechanism will be needed. Without an intermediary, CPU partitioning arrangement, or specialized operating system, any attempt to run multiple operating systems would simply bring down the entire system.

Other significant problems of virtualization include execution speed, security, memory management, multiplexing, and isolation of devices such as network cards.

The Popek and Goldberg principles of virtualization [1, 2] define a set of specifications for efficient processor virtualization. The standard suggests the ideal models and expectations of privileged instructions. Ideal instructions should behave in expected ways no matter what the current operating privilege level is and should trap any problems. Not all of the x86 processor instruction set meets the Popek and Goldberg principles, so an intermediary must be used to resolve the issues regarding the problematic small subset of the entire instruction set of the x86 architecture.

The Hypervisor/VMM Abstraction as an Intermediary

Hypervisors supervise and multiplex multiple operating systems by using highly efficient and sophisticated algorithms. The hypervisor is a well-isolated, additional but minimal software layer. The hypervisor must work with minimal overhead and maintain supervisory privileges over the entire machine at all times. The hypervisor seeks to define and enforce strong separation policies and finite boundaries for which operating systems can operate cooperatively.

The x86 processor line uses privilege levels known as rings, and a stand-alone hypervisor takes advantage of this fact. By obtaining privilege ring 0, the highest privilege level, the hypervisor can supervise and delegate all of the system resources. Of course the other operating systems will have to be kicked down to lesser privilege levels if the CPU does not support virtualization internally. X86 processor privilege protections provide the means to completely isolate the hypervisor. Often the hypervisor is referred to as a virtual-machine monitor (VMM), and these terms are used in this article interchangeably.

The x86 processor is dependent on lookup tables for runtime management. Global tables such as the interrupt descriptor vector and the memory segment descriptors are examples of such structures. Controlling access to these tables is a must for any VMM. In a multiplexed OS environment, these and other processor control tables are maintained by the hypervisor only and therefore must be emulated for each virtual context. In particular, it is the hypervisor's role to manage all processor control tables and other processor facilities that cannot be shared in a multiplexed OS environment.

Overview of Virtualization Mechanics

Emulation with Bochs

Bochs [9] is a software emulation of a CPU and the various PC chipset components; it implements the entire processor instruction set and emulates a fetch, decode, and execution cycle the way a physical CPU does. Bochs executes all instructions internally by calling its internal functions to mimic the real ones, which never hit the CPU. The Bochs emulation engine is implemented as a user-space application, and it uses its allocatable memory address space to model a physical memory address space.

This type of translation loses the battle when it comes to execution speed. The additional overhead of nested memory access, opcode parsing, and execution of emulated instructions using procedures in memory results in multiple real processor instructions for each of the emulated instructions. Bochs is an example of a simpler base case of virtualization.

Transition Algorithms and Direct Execution

Translators such as the versions used by VMware's ESX Server [10] and QEMU [11, 12] use intelligent forms of translation. Translators of this type have a huge performance advantage over a simpler interpretive type of emulation such as Bochs. The basic idea is to translate the source of instructions into a cache of instructions that can directly execute on the processor.

Forms of binary translation are intermediary algorithms that are used for processors that do not have virtualization support. As stated earlier, the x86 does not meet the standards provided by Popek and Goldberg. There are a few processor instructions that do not behave in a manner suitable for virtualization. The translation process scrubs and replaces problematic instructions with alternate instructions that will emulate the original.

An overly simplified example follows. While in the fetch and decode phase, the translator comes across the cli instruction. cli is a privileged instruction that disables interrupts on a x86 CPU. The translator can instead replace it with instructions that disable interrupts only on the abstraction that represents the virtual CPU for the virtual machine, not the real CPU. Again, this is a basic conceptual example to help get the idea of translation across.

The instructions that have been translated are cached into blocks that are used for direct execution on a CPU, and at the end of each translated block are instructions that lead back into the hypervisor. Any CPU exception or error that occurs while the translated stream is executing forces the CPU back to the hypervisor. This is critical for security and isolation; the virtualized operating system has no chance of getting control of the CPU, unless the hypervisor itself has been compromised.

Note that VMware has developed a very sophisticated version of binary translation. The algorithm is called Adaptive Binary Translation [3]. In addition, it is a VMM that has features that facilitate mass rollout and management of machines such as virtual-machine migration and memory management. QEMU is itself a user-space program, while ESX is implemented as a kernel. The advantage is that ESX is a hypervisor in the more strict definition which gives it full operational range over the processors.

Para-virtualization with XEN

Para-virtualization under XEN [4, 8] provides a software service interface for replacing privileged CPU operations. Operating systems must be specifically modified to be aware of the services that the XEN hypervisor provides.

To make an OS XEN-aware, the developer has to modify highly complex kernel procedures and replace privileged instructions in the source code with calls to the XEN interface which will emulate the operation in its isolated address space. After the OS is recompiled against the XEN Interface, a directly executable operating system is created. After some administrative setup, XEN can load and schedule this OS for direct processor execution.

The communications gateway that the para-virtualized OS uses to request XEN operations is based on interrupts, recast as hypercalls in XEN terminology. Hypercalls tie the operating system to the hypervisor. When a hypercall is issued, the CPU transfers control to a hypervisor procedure which completes the request in privileged mode. This is the same as the system call interface that system programmers are used to, except the requests are from the kernel to hypervisor. The XEN privileged operations exist in an address space only accessible by XEN, and this addressing method mimics the kernel/process address space split in standard x86_32 processors.

Hypercalls are used for registering guest local trap tables, making memory requests, guest page table manipulation, and other virtual requests. Kernel subsystems such as the slab cache object allocation and threading are not virtualized, but devices and page tables are virtualized. A full list of hypercalls can be viewed in xen.h in the source tree.

XEN, which is a stand-alone bare-bones kernel, maintains ultimate control over the processor as it is the supervisor of the system and sits isolated in ring 0, and the para-virtualized guest OS executes with reduced privileges. While any guest OS is executing on a processor, any processor exceptions or errors are trapped and handled by the hypervisor, thus providing strong isolation and security. For efficiency, XEN allows Linux to directly handle its system calls (0x80h) while it is on the CPU, thus bypassing the XEN layer. This is known as a fast-trap handler.

A XEN para-virtualized guest operating system has a startup and boot-strapping procedure that is different from the stand-alone OS. There is no BIOS available to query for things such as the installed memory size. XEN provides a special table that is mapped into the guest OS address space as a replacement. This is a C structure which is called the "start_info" page. The xen.h include file shows the specifics of the contents of this structure.

It is generally accepted that para-virtualization provides very good performance. The major downside is that the operating system source code must be modified by the maker or a third party. This presents a problem for systems, such as the Microsoft OS product line, which are not open source. Popular open sourced operating systems such as Linux and some versions of the BSD kernels have been successfully para-virtualized to run under XEN without CPU-supported virtualization. Note that XEN does support CPUs with full embedded virtualization.

Full Virtualization with CPU Supported Extensions

AMD, Intel, and others have now embedded virtualization properties directly in the processor. The advantage of this is that any x86-based operating system can execute directly in a virtualized machine context without any binary translation or source code modification. AMD calls its virtualization implementation AMD-V, and Intel calls its implementation Virtualization Technology (VT).

The AMD and Intel virtualization chipsets support the concept of a guest operating system and a new additional privilege mode exclusively for hypervisor use. The hypervisor executes with full authority over the entire machine, while the guest operates fully within its virtual-machine environment without any modification. From the guest OS point of view, there are no new or different privilege levels, and the guest OS is not aware that it is itself a guest. The usual 0/3 ring setup is maintained for the kernel and user processes.

Both AMD and Intel use the key idea of a VMM management table as a data structure for virtual-machine definitions, state, and runtime tracking. This data structure stores guest virtual machine configuration specifics such as machine control bits and processor register settings. Specifically, these tables are known as VMCB (AMD [5]) and VMCS (Intel [6]). These are somewhat large data structures but are well worth reviewing for educational purposes. These structures reveal the many complexities and details of how a CPU sees a virtual machine and shows other details that the VMM needs to manage guests.

The VMCB/VMCS management data structures are also used by the hypervisor to define events to monitor while a guest is executing. "Events" are processor-specific conditions that can be intercepted, such as attempts to access processor control registers and tables. Note that the VMCB/VMCS data structures are only mapped into hypervisor-accessible pages of memory. The guest OS cannot be allowed to access these tables, as this would violate isolation constraints.

The triggering of a monitored event is called a VM-EXIT (virtual-machine exit). When this happens the CPU saves the current executing state in the active VMCB or VMCS and transitions back into the hypervisor context. Here the hypervisor can monitor and fix the issue that caused the exit from the guest. Each specific processor uses model-specific registers to store the location of VMCB/VMCS tables.

Any errors, unexpected problems, or conditions that may require emulation that occur while the guest is executing force a VM-EXIT and switch to the hypervisor host context. This is somewhat similar to the exception/trap transition sequence used by x86 processors running standard operating systems.

Note that XEN, VMware, KVM, and Hyper-V support CPU-extended virtualization.

KVM under Linux

KVM is a very popular Linux-based hypervisor with full virtualization. KVM is a kernel module that brings virtualization directly into the host Linux kernel. The KVM is not completely stand-alone, as it uses the Linux host kernel subsystems. KVM is implemented using CPUs with full virtualization support, along with a QEMU software back-end. QEMU provides device emulation for guest operating systems under KVM control. I/O requests made to virtual devices are intercepted by KVM and queued up to a QEMU instance which is then scheduled for processor time in order to complete the request.

One of the differences with KVM is that it uses the host Linux kernel subsystems. Guest virtual machines are Linux tasks that execute in processor guest mode. KVM supports most operating systems by utilizing CPUs with virtualization support. See the KVM documentation [7] for the supported list of operating systems.

IOMMU (I/O Memory Management Unit)

IOMMU chipsets are becoming mainstream on x86_64 architectures. An IOMMU allows a hypervisor to manage and regulate DMA (direct memory access) from devices. In a stand-alone OS environment, a device can access any system memory address that it has address lines for. The operating system and its devices see a single system memory address space.

But in a multi-OS virtual machine environment, multiple system address spaces are defined by the hypervisor for guest use. Devices, however, still only see a single system address space. The IOMMU creates a virtual system address space for devices and effectively correlates, translates, and sets finite bounds on a device's range of addressable memory.

The IOMMU is positioned in the hierarchy of system buses where it can identify source devices and intercept their memory requests and use its internal IOMMU page tables to allow/deny and translate the requested memory address. The power of the IOMMU allows the hypervisor to assign devices to specific guest operating systems and restrict the devices' memory access to pages in the address space of the guest. This IOMMU isolation and mapping feature is used for PCI-Passthrough.

PCI-Passthrough permits a guest operating system to access a device natively. The guest OS is not aware that it is being redirected by an IOMMU and does not have the access or ability to make modifications to the IOMMU chip. Doing so would open up a huge security hole: for example, OS#2 could program its assigned device to write to a specific page of memory owned by OS#3 or overwrite a page owned by the hypervisor.

An IOMMU is somewhat analogous to the MMU used in x86 CPUs for virtual memory translation. Both IOMMU and MMU use page tables to hold address translation metadata. And, like the x86 CPU MMU, the IOMMU generates exceptions and faults which the hypervisor must catch and resolve.

Intel brands its IOMMU chipset Virtualization Technology for Directed I/O (VT-d), and AMD brands its chipset as I/O Memory Management Unit (IOMMU).

Conclusion

Virtualization innovations are accelerating and overcoming previous efficiency limitations. A good understanding of the internal structure is increasingly vital and will assist in understanding the implications of issues such as performance, scaling, security, and the proper architecting needs of the virtualization layer in your infrastructure.

At this point, full virtualization with CPU support will likely be the main focus of solutions going forward. It provides more flexibility and leverage for hypervisor implementors and more choices for the end user. VMware, XEN, Hyper-V, and KVM support CPUs that have embedded virtualization capabilities.

References

[1] Gerald J. Popek and Robert P. Goldberg, "Formal Requirements for Virtualizable Third Generation Architectures," *Communications of the ACM*, vol. 17, no. 7: 412–421.

[2] en.wikipedia.org/wiki/Popek_and_Goldberg_virtualization_requirements.

[3] Keith Adams and Ole Ageson, "A Comparison of Software and Hardware Techniques for x86 Virtualization," 2006: http://www.vmware.com/pdf/asplos235_adams.pdf.

[4] David Chisnall, *The Definitive Guide to the Xen Hypervisor* (Prentice-Hall, 2007).

[5] AMD Developer Manuals: http://developer.amd.com/documentation/guides/Pages/default.aspx.

[6] Intel Processor Developer Manuals: http://www.intel.com/products/processor/manuals/.

[7] KVM: http://www.linux-kvm.org/page/Main_Page.

[8] XEN: http://www.xen.org/.

[9] Bochs: http://bochs.sourceforge.net/.

[10] http://www.vmware.com/virtualization/.

[11] http://wiki.qemu.org/Main_Page.

[12] Fabrice Bellard, "QEMU, a Fast and Portable Dynamic Translator," 2005 USENIX Annual Technical Conference: http://www.usenix.org/events/usenix05/tech/freenix/full_papers/bellard/bellard_html/.