Survey of System Virtualization Techniques

Robert Rose

March 8, 2004

Abstract

This paper discusses two main trends in the development of virtual machine systems: *full system virtualization*, where an entire hardware architecture is replicated virtually, and *paravirtualization*, where an operating system is modified so that it can be run concurrently with other operating systems that have also been designed for paravirtualization.

1 Introduction

When computer systems were first invented they were mammoth systems that were large and expensive to operate. Due to their size, expense, and demand for their usage, computer systems quickly evolved to become time-sharing systems so that multiple users (and applications) could use them simultaneously. As computers became more prevalent however, it became apparent that simply time-sharing a single computer was not always ideal. For example, misuse of the system, intentional or unintentional, could easily bring the entire computer to a halt for all users. For organizations that could afford it, they simply purchased multiple computer systems to mitigate these pitfalls.

Having multiple computer systems proved beneficial for the following reasons:

Isolation. In many situations it is beneficial to have certain activities running on separate systems. For instance an application may be known to contain bugs, and it might be possible for the bugs to interfere with other applications on the same system. Placing the application on a separate system guarantees it will not effect the stability of other applications.

Performance. Placing an application on it's own system allows it to have exclusive access to the system's resources, and thus have better performance than if it had to share that system with other applications. User-level separation of applications on the same machine does not effectively performance isolate applications–scheduling priority, memory demand, network I/O and disk I/O of one process can effect the performance of others [1]. (For example, one application thrashing the hard disk can slow all other applications on the same system).

Most organizations at the time weren't so fortunate to be able to purchase multiple computer systems. It was also recognized that purchasing multiple computer systems was often wasteful, as having more computers made it even harder to keep them busy all the time. However having multiple computers obviously had it's benefits, so taking cost and waste into consideration IBM in 1960's began developing the first *virtual machines* that allowed one computer to be shared as if it were several.

This paper discusses two main trends in the development of virtual machine systems: *full system virtualization*, where an entire hardware architecture is replicated virtually, and *paravirtualization*, where an operating system is modified so that it can be run concurrently with other operating systems that have also been designed for paravirtualization.

1.1 Virtual Machines

IBM developed the virtual machine concept as a way of time-sharing very expensive mainframe computers. Typically an organization could only afford one mainframe, and this single mainframe had to be used for development of applications and deployment of applications. Developing an application on the same system you intend to deploy on while other applications are "in production" on that system was generally considered bad practice. This is still true today. Development activities may require reboots or cause instabilities in the system, and it is undesirable to have these activities interfere with production applications.

The virtual machine concept allows the same computer to be shared as if it were several. IBM defined the virtual machine as a fully protected and isolated copy of the underlying physical machine's hardware[2]. IBM designed their virtual machine systems with the goal that applications, even operating systems, run in the virtual machine would behave *exactly* as they would on the original hardware.

1.2 Virtual Machine Monitor

The idea of a Virtual Machine Monitor (VMM) goes hand-in-hand with virtual machines. The VMM is the software component that hosts guest virtual machines. In fact, the VMM is often referred to as the *host* and the virtual machines as *guests*. The VMM is a software layer that abstracts the physical resources for use by the virtual machines. Because the VMM provides an abstraction, it can run multiple virtual machines on the same system. Figure 1 demonstrates the relationship between the VMM and virtual machines.

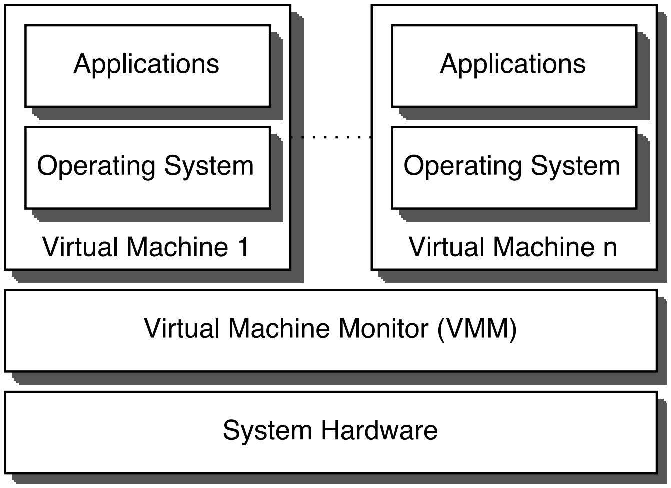

Figure 1: Virtual Machine Monitor - Virtual Machine Relationship

The VMM provides a virtual processor and other virtualized versions of system devices such as I/O devices, storage, memory, etc. The VMM also provides isolation between the virtual machines it hosts so that problems in one cannot effect another.

1.3 Virtualizing Internet Services

In the age of the Internet boom a new kind of system was popularized for building Internet applications: the multi-tier system architecture. This common way of building Internet applications separates the web server physically from the database server and easily enables load-balancing and clustering of an application's components. This approach is scalable, easier to manage, and extremely fault-tolerant; however IT departments are quickly realizing that these benefits are often outweighed by the cost of operating the additional hardware required for physical server independence. Virtual machines provide the same benefits of componitization while dramatically lowering hardware and operational costs.

The benefits of running multiple services on one piece of hardware are reinforced by Zipf's law [3]. Zipf's law states that the frequency of an event is proportional to $x^{-\alpha}$, where x is the rank of the event in comparison to all other events. Studies have found that frequency of access to web servers and other Internet Services can be fit to Zipfian distributions by observing the usage behaviors of web caches [4]. Zipfian distributions have shown that most Internet Services are relatively unpopular, and a small few make up the majority of accesses by customers. This is true for web caches and server environments that host a large number of diverse Internet Services.

If only a small number of Internet Services are frequently active, and the rest are dormant most of the time, then does isolation for performance still make sense? No. It is clearly a waste of resources to have a small set of computers be busy most of the time and have the rest lay idle. Consolidating Internet Services to a single machine or group of machines by using virtual machines is an elegant solution that provides the benefit of isolation while simultaneously reduces wasted computing power and maintenance of additional computers.

1.4 Requirements for Virtual Machines

In 1974 Popek and Goldberg defined what they believed where the formal requirements for a virtualizable computer architecture [5]: *For any computer a virtual machine monitor may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions.* In other words, the most essential requirement a computer architecture must exhibit in order to be virtualizable is that *privileged instructions must trap*, meaning when a guest virtual machine (while running directly on the real processor) attempts to execute a privileged instruction, the processor stops and returns control to the VMM so it can either decide whether or not to execute the instruction, or simulate the instruction by some other means.

Popek and Goldberg also stated that a virtual machine architecture has three essential characteristics:

1. *Any program run under the VMM should exhibit an effect identical*

with that demonstrated if the program had been run on the original machine directly. They offered one exception to this rule, timing. The software (or hardware) aiding the virtual machine needs to manage the resources used by the virtual machine(s), and this requires it to intervene occasionally, thus altering the timing characteristics of the running virtual machine(s).

2. *A statistically dominant subset of the virtual processor's instructions are executed directly by the real processor.* Popek and Goldberg say that a virtual machine is different from an emulator. An emulator intervenes and analyzes every instruction performed by the virtual processor, whereas a virtual machine occasionally relinquishes the real processor to the virtual processor. For efficiencies sake, this relinquishment must make up the majority of the real processor's workload.

3. *The VMM is in complete control of system resources.* A virtual machine running on the system does not have direct access to any of the system's real resources, it must go through the VMM.

These characteristics, although interesting on the surface, prove to be difficult or undesirable to meet, as shall be discussed in further sections.

2 Full System Virtualization

Full system virtualization provides a virtual replica of the system's hardware so that operating systems and software may run on the virtual hardware exactly as they would on the original hardware. The first full system virtualization system developed was the CP-67 software system which ran on the IBM 360/67 mainframe computer [6]. The CP-67 program was designed as a specialized time-sharing system which exposed to each user a complete virtual System/360 computer.

2.1 Virtualizing System/370

The performance of CP-67 and later software virtualization systems for System/360 was less than spectacular, so IBM decided to create a computer that had a specialized architecture intended to aid virtualization. This architecture was first delivered in the Virtual Machine Facility/370 (VM/370)

[7]. The VM/370 was a VMM that ran on the System/370 Extended Architecture (370-XA). 370-XA provided specific CPU instructions designed to maximize the performance of running virtual machines. To the user, the virtual machine exposed was a virtual replica of the System/370 computer.

To increase the performance of VM/370, the 370-XA platform provided "assists" in the architecture that boosted the performance of certain operations performed repeatedly by the VM/370 VMM. Before assists, the VMM had to simulate many of the architectures instructions for each virtual machine so they could be performed safely without interfering with other virtual machines. The development of assists allowed some of these simulated instructions to be executed in safely in hardware. Other assists supplanted frequently executed instructions that still required VMM intervention. Assists were enabled by placing the system in *interpretive-execution* mode. A few of the assists used by the VM/370 are highlighted below:

1. *Virtual Machine Assist (VMA).* The VMA provides specific instructions for accelerating aspects of the VMM. It consisted of 13 functions, 12 of which replaced guest virtual machine instructions that would have otherwise been simulated by the VMM, and another function that managed certain data structures used by the VMM. VMA alone offered a 35% performance increase for most applications.

2. *Extended Control Program Support (ECPS).* ECPS assists were a collection of 35 functions that targeted specific application programs. ECPS took advantage of how certain programs utilized the System/370 architecture to accelerate common tasks, and took over some of these functions that were previously handled by the VMM.

3. *Shadow-table-bypass.* The VM/370 architecture used a "shadow table" to map virtual memory in the virtual machine space to actual memory in the physical machine space. This multi-level mapping was very expensive, so the VM/370 opened up assists in the hardware that allowed *trusted* guests direct access to the memory mapping tables. Although this is a significant risk–one incorrect memory access made by on virtual machine could influence other virtual machines–IBM found that for most guest operating systems of their own design they could expect them to be well behaved.[1]

[1] Popek and Goldberg may have reservations about this arrangement, as this violates their requirement that the VMM be in complete control of machine resources.

In all, IBM developed over 100 assists to enhance the performance of the VM/370 system. Many of the assists were developed to speed specific application programs. The development of assists greatly improved the performance of the System/370 virtual machine systems, and did not require the code running in the virtual machine environments to be modified.

2.2 Virtualizing IA-32

IA-32 (x86) is the most dominate computer architecture today. IA-32 is used everywhere from client applications to high-end 24x7 reliability server applications. Virtualizing IA-32, especially in the Internet Services domain mentioned previously, would have tremendous benefits. But IA-32 was never intended to be virtualized. This design decision is most apparent in a small set of essential IA-32 instructions that are not required to be executed in privileged mode, but can severely damage the stability of the system. On a system running only one operating system this is not an issue, but when attempting to run multiple operating systems virtually it is very significant. Another aspect of IA-32 that makes it difficult to be virtualized is the platforms "open" nature–there are a great deal of different devices and device drivers available for IA-32, which makes virtualization extremely difficult.

Creative programming however has overcome IA-32s virtualization weaknesses. Robin and Irvine outline the procedure typically used on IA-32 to virtualize the system [8]:

1. *Non-sensitive, non-privileged instructions may be run directly on the processor.* If an instruction is known to be safe it can be run directly on the processor without intervention.

2. *Sensitive, privileged instructions trap.* The virtual machine is run in user mode on the processor, so when the virtual machine attempts to execute an instruction that is privileged it causes the CPU to issue an interrupt. The VMM traps this interrupt and performs whatever steps necessary to emulate the instruction for the virtual machine.

3. *Sensitive, non-privileged instructions detected.* Unfortunately, the IA-32 architecture has 17 "problem instructions" that are extremely sensitive to be running in a virtualized environment, but cannot be trapped. The VMM must monitor the running virtual machine to insure it does not execute these instructions.

Obviously, because a VMM is required to go to these great lengths to virtualize the IA-32 architecture, IA-32 does not meet the Popek and Goldberg requirements described above for a virtualizable architecture. But clever hacking can overcome this, as is evident with products such as Plex86 [9], User Mode Linux [10], Virtual PC [11], and VMWare [12].

2.3 VMWare

VMWare is a popular virtualization tool for the IA-32 platform [12]. VMWare takes a unique approach to virtualization in that it runs it's VMM entirely within a host operating system. Rather than needing to support the wide variety of devices available for IA-32, VMWare leaves it up to the host operating system to provide abstractions for these devices. VMWare refers to this as the *Hosted Virtual Machine Architecture*.

VMWare does not run completely in application space however, it also installs a special operating system driver called the *VMDriver* that allows virtual machines instantiated by VMWare to have faster access to the devices on the system. VMDriver installs itself in the operating system kernel to access devices, thus bypassing the need to support the wide array of devices available on IA-32.

Providing a virtual replica of the IA-32 architecture for each virtual machine would also be difficult, so VMWare instead provides only a *generic* set of devices to each virtual machine. Each virtual machine is exposed a PS/2 keyboard, PS/2 mouse, floppy drive, IDE controller, ATAPI CD-ROM, Soundblaster 16 audio card, serial ports, parallel ports, a standard graphics display card, and any number of AMD PCNet ethernet adapters. Exposing only a generic set of set of devices greatly simplifies the implementation of VMWare.

VMWare's network interface card implementation is particularly interesting. VMDriver places the physical network card in promiscuous mode and creates a virtual ethernet bridge that receives all of the network card's packets. VMDriver can then analyze each packet and either route it back to the host operating system, os route it to a particular virtual machine's virtual network interface card. This implementation also allows VMWare to provide Network Address Translation (NAT) in the virtual bridge so that each virtual machine believes it has it's own IP address.

2.4 Virtualizing IA-64

While the IA-64 (Itanium) architecture is not widely popular yet, some groups are beginning to asses the IA-64 platform's ability to host virtual machines. Hewlett-Packard has already developed an IA-64 framework for virtualizing HP/UX and Linux, and is beginning work on Windows Server [13]. Virtualizing IA-64 in part has similar issues as virtualization of IA-32, probably the most troublesome of which is the wide array of devices available that can be run on the platform.

IA-64 does provide one important feature not available on IA-32: ring compression. Guest virtual machines can be run in a ring higher than the host VMM, and on IA-64 traps on the higher rings can be captured by the lower rings. This greatly eases virtualization of many common instructions.

2.5 Drawbacks of Full System Virtualization

Full system virtualization has the benefit that operating systems and applications may run on it unmodified, completely oblivious to the environment in which they are actually running. This has it's drawbacks however. For example, full virtualization was never part of the IA-32 (x86) design goals, and VMMs running on this platform must use special tricks (described above) to virtualize the hardware for each virtual machine. Efficiently virtualizing virtual memory management (referred to as "shadow mapping" by the VM/370 architecture described above) is also extremely difficult, especially on the IA-32 architecture (Denali, described below, does away with virtual memory).

The problems with virtualizing virtual memory cannot be understated. Take, for example, what happens in a typical operating system that uses virtual memory. When an application makes a request for a page of memory, the operating system translates the memory address from the applications "virtual" space into the system's real space using a page table. Unused pages can also be written to disk when they become inactive or when other applications require more memory. This is typically accomplished using special instructions on the CPU for memory management.

Virtualizing virtual memory is difficult because the VMM must intercept all virtual memory calls to the CPU, translate "virtual machine" space into the system's real space using yet another page table, retrieve the memory (which may be in memory or on disk), and then return the memory to the virtual machine. On the surface this doesn't sound too terrible, but keep in mind that before the VMM received the memory access call the virtual machine already performed it's own page table lookup to see where the memory was located. By the time the virtual machine has received it's memory at least two context switches between the virtual machine and the VMM had to take place. That's very expensive for a simple memory access.

3 Paravirtualization

Numerous systems have been developed that use the techniques described above: specialized architectures for running virtual machines in, or completely emulating a machine environment. These systems have the disadvantage that they either require specialized hardware, offer less than desirable performance, or cannot support commodity operating systems. Today, the world where the PC architectures have come to dominate the server room, supporting commodity operating systems has become extremely important. There is a great wealth of server software written for commodity operating systems, and porting it to a special architecture so it can be virtualized is unreasonable. The PC architecture however doesn't lend itself to virtualization very easily.

Full virtualization on PC architectures is extremely complex, as was discussed in the proceeding section. Fully virtualizing the IA-32 (x86) architecture, for example, yields very poor performance of the virtual machine because the VMM software must intervene too often to perform protected tasks–the architecture provides no assistance for virtualization. *Paravirtualization* attempts to mitigate this problem by modifying the operating system so that instead of going directly to the CPU to perform protected tasks it goes to the VMM. The two paravirtualization systems discussed here, Denali and Xen, have been shown to yield significantly better performance over full system virtualization systems.

3.1 Denali

Denali is a paravirtualization system developed at the University of Washington [3]. Denali strives to provide "lightweight protection domains"–minimalistic, fast containers for running virtual machines within.

The interface the Denali VMM provides for the virtual machines is not an exact replica of the system hardware, and thus violates Popek and Goldberg "identical behavior" requirement given above. Not providing an exact replica requires the operating system and software running with that operating system to be modified to run in the Denali environment. However, by not providing an exact replica, Denali has developed methods that substantially improve performance over fully virtualized systems. The most notable ways Denali has challenged the traditional view of virtual machines are given below:

1. *Idle loops.* Many operating systems sit in a "busy wait" while they are waiting for something to happen–disk I/O, network I/O, or simply they have nothing else to do. The Denali VMM introduces a new "idle" instruction that the operating system running in each virtual machine is expected to call when it enters these states, rather than busy wait. When the virtual machine calls the idle instruction a context switch to the VMM occurs so that it can schedule other virtual machines. This promotes a higher overall CPU utilization on the system, as the virtual machine is no longer charged for busy waits. Denali's idle instruction also provides a time parameter, the maximum amount of time the virtual machine is willing to wait.

2. *Interrupt queueing.* In a normal virtual machine environment, when an interrupt arrives at the CPU, the VMM must take over and immediately context switch to the correct virtual machine and dispatch the interrupt. The probability that the interrupt is destined for the virtual machine that is currently running substantially decreases as the number of virtual machines increases. Because the cost of context switching to the correct virtual machine is substantial, Denali does away with immediately dispatching the interrupt. Instead, Denali queues the interrupt so that the interrupt is dispatched the next time the virtual machine is run.

3. *Interrupt semantics.* On most systems an interrupt means that *something just happened.* A side effect of interrupt queuing (above) is that Denali must alter the semantics of the interrupt to mean *something happened recently.* This is actually beneficial to the speed of the system when handling timers–rather than the operating system in the virtual machine periodically checking the "elapsed ticks" timer in a CPU register, it requests that a timer interrupt be scheduled with the VMM.

This saves a lot of processor time context switching just to check a timer.

4. *No virtual memory.* Supporting virtual memory between the VMM and the virtual machine is extremely difficult, especially on architectures that are not designed to be virtualized, as demonstrated in the proceeding section by systems such as VMWare, Plex86, Virtual PC, etc. Because Denali is not concerned with running commodity operating systems within the virtual machine, Denali does away with virtual memory–constraining each virtual machine to a single address space. Denali justifies this change because it targets uses where each virtual machine runs very few, small applications. If an application is known to cause problems, then simply run it in its own virtual machine.

5. *No BIOS.* Other system virtualization systems for IA-32 provide access to the BIOS for backwards-compatibility and to provide bootstrapping information for the operating system running in the virtual machine. Denali does away with this, providing system information in read-only virtual registers.

6. *Generic I/O devices.* Denali provides no specialized access to devices on the physical system. Instead, Denali exposes only a small set of "generic" devices, such as a network interface card, serial port, a timer, and keyboard. This greatly improves the performance of the guest operating systems–rather than going through complex drive I/O routines to send a network packet, for example, Denali exposes a virtual I/O instruction that handles everything for the guest OS.

Because of these significant deviations Denali makes from the typical virtual machine paradigm, it is not trivial to port an operating system to run on Denali. Denali instead provides a guest operating system named Ilwaco that runs on the Denali architecture. Ilwaco provides a simple TCP/IP stack, a subset of libc, and operating system assisted threading.

3.2 Xen

Xen is a paravirtualization system developed by the University of Cambridge [1]. Xen has the unique goal of paravirtualizing commodity operating systems such as Linux, NetBSD and Windows XP, rather than support it's own special operating system. Xen also aims for 100% binary compatibility for applications running in it's virtual machines. In other words, to run your application on Xen, you only need a version of your operating system that has been ported to Xen and everything should behave as expected.

Xen's approach to paravirtualizing the IA-32 architecture can be summarized as follows:

1. *Partial access to hardware page tables.* Each virtual machine is given read-only access to the hardware page tables; updates to the page tables are queued and processed by the VMM.

2. *Lower privilege levels.* Xen runs it's guest virtual machines one ring lower than the VMM.

3. *Trap handlers registered with VMM.* Rather than registering trap handlers directly with the (virtual) CPU, guest operating systems must register them with the Xen VMM.

4. *System calls registered with processor.* On most operating systems system calls are processed using a lookup table and a special trap sequence. When virtualized, the trap sequence causes the VMM to be invoked on every system call. Xen sidesteps this inefficiency by allowing guest operating systems to install their system call handlers directly with the processor, bypassing the context switches necessary for the VMM to process the trap sequence.

5. *No hardware interrupts.* Hardware interrupts are replaced by a lightweight event system.

6. *Generic devices.* Like Denali, Xen only provides to the guest virtual machine a small set of fast, generic devices.

Xen's ambitious goals appear to have been met. Today, a special Xen-compatible version of Linux, XenoLinux [14], is available that runs in the Xen environment. According to the Xen researchers, 100 XenoLinux instances can be run simultaneously on a single Xen VMM with minimal performance degradation. Xen-compatible versions of Windows XP and NetBSD are actively being developed at the time of this writing.

4 Conclusion

System virtualization is an age-old art that will continue as long as applications need isolation and performance independence. Because the dominant system architecture of today, IA-32, is not an architecture designed to be virtualized, clever programming and techniques that push the boundaries of what virtualization means will continue to be employed.

References

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauery, I. Pratt, and A. Wareld, "Xen and the Art of Virtualization," in *SOSP'03*. ACM, 2003.

[2] R. J. Creasy, "The Origin of the VM/370 Time-Sharing System," *IBM Journal of Research and Development*, vol. 25, no. 5, p. 483, 1981.

[3] A. Whitaker, M. Shaw, and S. Gribble, "Denali: Lightweight Virtual Machines for Distributed and Networked Applications," 2002. [Online]. Available: citeseer.nj.nec.com/whitaker02denali.html

[4] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web Caching and Zipf-like Distributions: Evidence and Implications," in *INFOCOM (1)*, 1999, pp. 126–134. [Online]. Available: citeseer.nj.nec.com/breslau98web.htm

[5] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Commun. ACM*, vol. 17, no. 7, pp. 412–421, 1974.

[6] T. V. Vleck, "The IBM 360/67 and CP/CMS," 1997. [Online]. Available: http://www.multicians.org/thvv/360-67.html

[7] P. H. Gum, "System/370 Extended Architecture: Facilities for Virtual Machines," *IBM Journal of Research and Development*, vol. 27, no. 6, p. 530, 1983.

[8] J. S. Robin and C. E. Irvine, "Analysis of the Intel Pentiums Ability to Support a Secure Virtual Machine Monitor," *USENIX Security Symposium*, pp. 129–144, 2000.

[9] "The Plex86 Project," 2003. [Online]. Available: http://plex86.sourceforge.net/

[10] S. T. King, G. W. Dunlap, and P. M. Chen, "Operating System Support for Virtual Machines," in *USENIX Technical Conference*, 2002.

[11] "Virtual PC Technical Overview," 2004. [Online]. Available: http://www.microsoft.com/windowsxp/virtualpc/evaluation/techoverview.asp

[12] G. Venkitachalam and B.-H. Lim, "Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor." [Online]. Available: citeseer.nj.nec.com/venkitachalam01virtualizing.html

[13] C. de Dinechine, T. Kjos, and J. Ross, "Itanium Virtual Machine," 2003.

[14] U. of Cambridge Computer Laboratory, "The Xen Virtual Machine Monitor," 2004. [Online]. Available: http://www.cl.cam.ac.uk/Research/SRG/netos/xen/