ANANDTECH
Your Source For Hardware Analysis and News

Hardware Virtualization: the Nuts and Bolts

by **Johan De Gelas** on 3/17/2008 3:00:00 AM
Posted in **IT Computing**

If you like to understand how binary translation, paravirtualization, and hardware virtualization work, this article is for you. We delve deep into the heart of VMware ESX, Intel VT-x, and AMD's nested paging to understand the impact on the performance of your virtualized server. Dig in!

Page 1

Introduction

First dual-core in 2005, then quad-core in 2007: the multi-core snowball is rolling. The desktop market is still trying to find out how to wield all this power; meanwhile, the server market is eagerly awaiting the octal-cores in 2009. The difference is that the server market has a real killer application, hungry for all that CPU power: virtualization.

While a lot has been written about the opportunities that virtualization brings (consolidation, hosting legacy applications, resource balancing, faster provisioning...), most publications about virtualization are rather vague about the "nuts and bolts". We talked to several hypervisor architects at VMWorld 2008. In this article, we'll delve a bit deeper as we look to understand the impact of virtualization on performance.

Performance? Isn't that a non-issue? Modern virtualization solutions surely do not lose more than a few percent in performance, right? We'll show you that the answer is quite a bit different from what some of the sponsored white papers want you to believe. We'll begin today with a look at the basics of virtualization, and we will continue to explore the subject in future articles over the coming months.

In this first article we discuss "hardware virtualization", i.e. the technology that makes it possible to offer several virtualized server such as VMware's ESX, Xen, and Windows 2008's Hyper-V. We recently provided an introduction to application virtualization using Thinstall, SoftGrid, and others software packages at our new IT portal, it.anandtech.com. These articles are all about quantifying the performance of virtualized servers and understanding virtualization technologies a bit better.

Hardware or Machine Virtualization versus "Everyday" Virtualization

Every one of us has already used virtualization in some degree. In fact, most of us wouldn't be very productive without the virtualization that a modern OS offers us. A "natively running" server or workstation with a modern OS already virtualizes quite a few resources: memory, disks, and CPUs for example. For example, while there may only be only 4GB RAM in a Windows 2003 server, each of the tens of running application is given the illusion that they can use the full 2GB (or 3GB) user-mode address space. There might only be three disks in a RAID-5 array available, but as you have created 10 volumes (or LUNs), it appears as if there are 10 disks in the machine. Although there might only be two CPUs in the server, you get the impression that five actively running applications are all working in parallel at full speed.

So why do we install a hypervisor (or VMM) to make fully virtualized servers possible if we already have some degree of virtualization in our modern operating systems? Operating systems isolate the applications weakly by giving each process a well-defined memory space, separating data from instructions. At the same time, processes share the same files, may have access to some shared memory, and share the same OS configuration. In many situations, this kind of isolation was and is not sufficient. One process that takes up 100% of the CPU time may slow the other applications to snail speed for example, despite the fact that modern OSes use preemptive multitasking. In case of pure hardware virtualization, you will have completely separate virtual servers with their own OS (guest OS), and communication is only possible via a virtual network.

Page 2

A Matter of Privileges

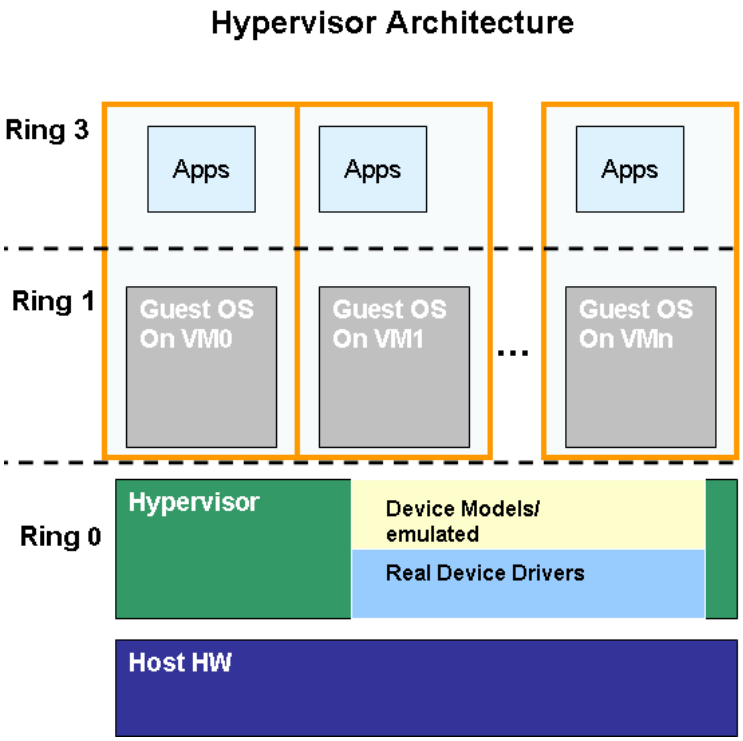
To create several virtual servers on one physical machine, a new software layer is necessary: the hypervisor, also called **Virtual Machine Monitor** (VMM). The most important role is to arbitrate the access to the underlying hardware, so that guest OSes can share the machine. You could say that a VMM manages virtual machines (Guest OS + applications) like an OS manages processes and threads.

To understand how the VMM actually works, we first have to understand how a modern operating systems works. Most modern operating system work with two modes:

A kernel mode that is allowed to run almost any CPU instructions, including "privileged" instructions that deal with interrupts, memory management, and so on. This is of course the mode that the operating system runs in.
A user mode that allows only instructions that are necessary to calculate and process data. Applications run in this mode and can only make use of the hardware by asking the kernel to do some work (a system call).

The whole user/kernel mode arrangement is based on the fact that RAM is divided into pages. (It is also possible to work with segment registers and tables, but that is a discussion for another article.) Before a privileged instruction is executed, the CPU first checks if the page from where the instruction originates actually has the right 2-bit code. The most privileged instructions require a 00 "privilege code". This 2-bit code allows four levels of code, with "11" being the lowest level.

To illustrate this, this 2-bit code is graphically represented in many publications by four "onion rings" (as you can see in this article). Ring 0 is the most privileged layer, ring 1 is a bit less privileged, and ring 3 is where the user applications reside with no privileges to manage the hardware resources at all.



Ring deprivileging with software virtualization: the guest OSes are no longer running in ring 0, but with less rights in ring 1.

A technique that all (software based) virtualization solutions use is thus ring deprivileging: the operating system that runs originally on ring 0 is moved to another less privileged ring like ring 1. This allows the VMM to control the guest OS access to resources. It avoids for example one guest OS kicking another out of memory, or a guest OS controlling the hardware directly.

Page 3

Virtualization Challenges

The grandfathers of virtualization, such as the IBM S/370, used a very robust system to allow the hypervisor to control the virtual machines. Every privileged instruction by a virtual machine caused a "trap", an error, as it was trying to execute a "resource management" instruction while running in a less privileged ring. The VMM intercepts all those traps and emulates the instruction, without jeopardizing the integrity of the other guests. In order to improve performance, the developers of the guest OS and VMM (both at IBM) tried to minimize the number of traps and reduce the time required to take care of the various traps.

This kind of virtualization was not possible on x86 as the 32/64-bit Intel ISA does not trap every incident that should lead to VMM intervention. One example is the POPF instruction that disables and enables interrupts. The problem is that if this instruction is executed by a guest OS in ring 1, an x86 CPU does not make a fuss about it, but simply ignores it. The result is that if a guest OS is trying to disable an interrupt, the interrupt is not disabled at all, and the VMM has no way of knowing that this is happening. As always, the good old x86 ISA is a bit chaotic: it has 17 of these "non-interceptable, cloaked for the VMM" instructions [1]. The conclusion is that x86 cannot be virtualized the way that the old mainframes were virtualized. Incidentally, the PowerPC and Alpha ISA's are clean enough to be virtualized in the classic manner.

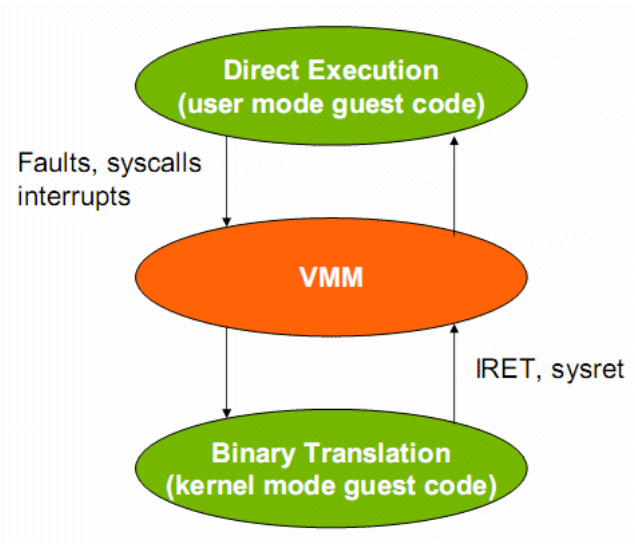
The above is much more than a quick simplified history lesson. Keep this in mind when we discuss what Intel and AMD have been doing with VT-x and AMD-V.

Page 4

Binary Translation

VMware didn't wait for Intel or AMD to solve the "x86 stealth instructions" problem and launched their solution at the end of the previous century (1999). To uncloak the stealthy x86 instructions, VMware used Binary translation (unfortunately, a Tachyon detection grid proved too expensive) . VMware's Binary Translation is a lot lighter than the binary translation technology that the Intel Itanium (x86 to IA64), Transmeta (x86 to VLIW), Digital FX!32 (Alpha to x86), or Rosetta software use. It doesn't have to translate from one Instruction Set Architecture (ISA) to another but it is based on an x86 to x86 translator. In fact, in some cases it just makes an exact copy of the original instruction.

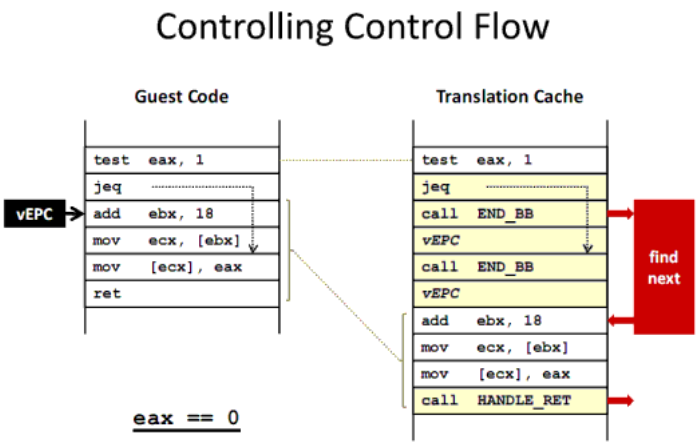
VMware translates the binary code that the kernel of a guest OS wants to execute on the fly and stores the adapted x86 code in a Translator Cache (TC). User applications will not be touched by VMware's Binary Translator (BT) as it knows/assumes that user code is safe. User mode applications are executed directly as if they were running natively.



User applications are not translated, but run directly. Binary Translation only happens when the guest OS kernel gets called.

It is the kernel code that has go through the "x86 to slightly longer x86" code translation. You could say that the kernel of the guest OS is no longer running. The kernel code in the memory is nothing more than an input for the BT; it is the BT translated kernel that will run in ring 1.

In many cases, the translated kernel code will be an exact copy. However, there are several cases where BT must make the "translated" kernel code a bit longer than the original code. If the kernel of the guest OS has to run a privileged instruction, the BT will change this kind of code into "safer" user mode code. If the kernel needs to get control of the physical hardware, the BT will replace that binary code with code that manipulates the virtual hardware.



Binary translation from x86 to x86 virtualized in action. (Image: VMware[2])

Binary translation is all about scanning the code that the kernel of the guest OS should execute at a certain moment in time and replacing it with something safe (virtualized) on the fly. With "safe", we mean safe for the other guest OSes and the VMM. VMware also keeps the overhead of the translation as low as possible. The BT does not optimize the binary instruction stream, and an instruction stream that has been translated is kept in a cache. In case of a loop, this means that the translation is done only once.

The TC is not only a Translator Cache but also a bit of a Trace Cache as it keeps track of the control flow of the program. Each time the kernel jumps to another address location, the BT cannot copy this exactly. If the original code had to jump 100 bytes for example, it is very unlikely that the translated part of the kernel in the TC has to jump the same number of bytes. The BT has probably lengthened the "in between" code a bit.

It is clear that replacing code with "safer" code is a lot less costly than letting privileged instructions result in traps and then handling those traps afterwards. Nevertheless, that doesn't mean that the overhead of this kind of virtualization is always low. The "Translator overhead" is rather low, and its impact gets lower and lower over time, courtesy of the Translator cache. However, BT cannot completely crack several hard nuts:

System Calls
Accesses to chipset and I/O, interrupts and DMA
Memory management
"Weird and complex code" (Self-modifying, indirect control flows, etc.)

Especially the first three are interesting. The last one is hard in an OS running in "native mode" too, so it is only normal that this doesn't get any better if you run more than one OS.

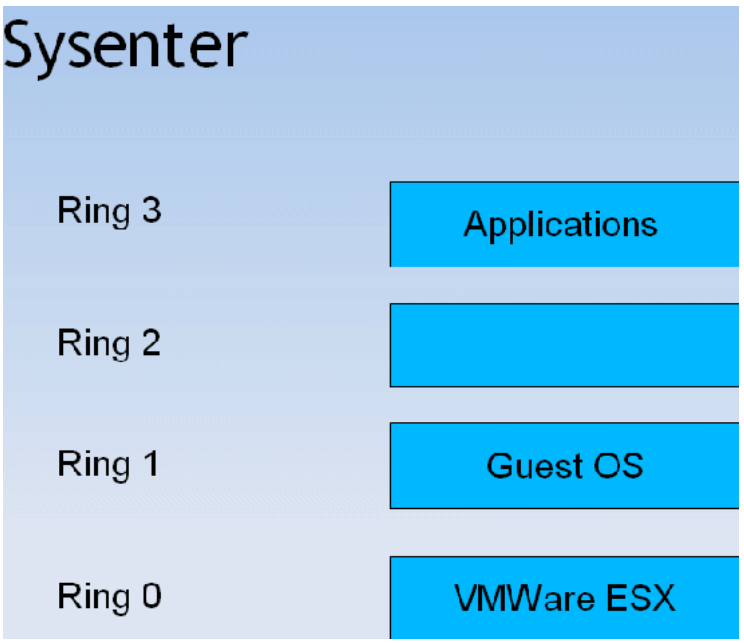
Page 5

System Calls

Much has been written about kernels, but it remains one of the most confusing subjects. Some publications give the impression that the kernel is some kind of "overlord" process that is always watching in the background. This is wrong of course, because this would mean that modern multitasking operating systems would not work on a single-threaded, single-core CPU. When only one thread can be active at a given time, how can the OS keep control?

A kernel is just another process that gets time slices from the multitasking CPU. The difference from other processes is that it has privileged access to CPU instructions that other processes don't have. Therefore, "normal" (user) processes will have to switch to the kernel to perform a privileged task like getting access to the hardware. If they don't, the CPU will cause an exception and the kernel will take over anyway. At the same time, a scheduler of the kernel uses the timer interrupt to intervene from time to time, making sure that no process tries to keep the CPU to itself (preemption) for too long. You could also say that the CPU is forced to load the OS scheduler process from time to time[5].

A system call is thus the result of a user application that requests a service of the kernel. x86 provides a very low latency way to get system calls done: SYSENTER (or SYSCALL) and SYSEXIT. A system call will give the Virtual Machine Monitor, especially with binary Translation (BT), quite a bit of extra work. As we have stated before, software virtualization techniques (such as BT) place the (32-bit) operating system at a slightly less privileged ring than normal (1 instead of 0). The problem is that a SYSENTER (request the service of the kernel) is sent to a page with privilege 0. It expects to find the operating system but it arrives in the VMM. So the VMM has to emulate every system call, translate the code, and then hand over the control to the translated kernel code which runs in ring 1[3].



A system call is a lot more complex when it happens on a virtualized machine.

When the binary translated guest OS code is done, it will use a SYSEXIT to return to the user application. However, the guest OS is running at level one and doesn't have the necessary privileges to perform SYSEXIT, so the CPU faults to the level zero and the VMM has to emulate what the guest OS should have done. It is clear that system calls cause a lot of overhead. A system call on virtualized machine will cost roughly 10 times more than on a native machine. Engineers at VMware measured on a 3.8 GHz Pentium 4 [4]:

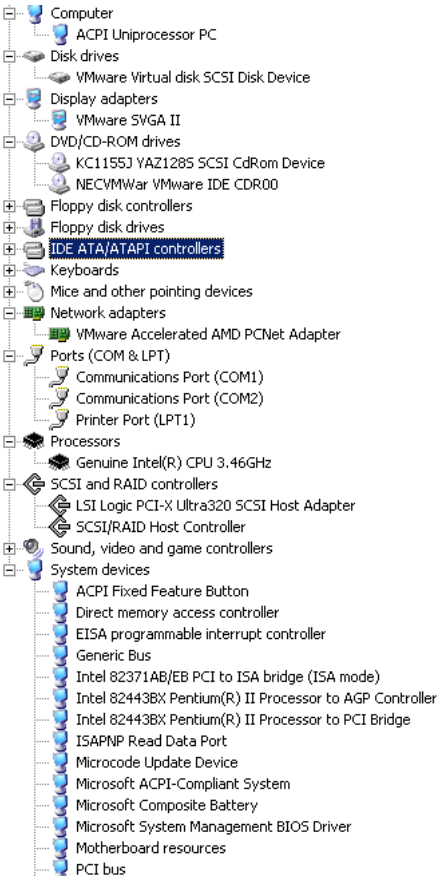
A native system calls takes 242 cycles
A binary translated one with the 32-bit guest OS running on ring 1 takes 2308 cycles

If you have a few of those virtualized machines running, system calls are suddenly much more than the background noise they were on a modern OS running on a native machine.

Page 6

I/O Virtualization

I/O is a big issue for any form of virtualization. If your virtualized server lacks CPU power, you can just add more CPUs or cores (i.e. replace dual-core CPUs with quad-cores). However, the memory bandwidth, the chipset, and storage HBA are in most cases shared by all virtual machines and a lot harder to "upgrade". Moreover, contrary to the CPU, the rest of the hardware in most virtualization software is emulated. This means that each access to the driver of a virtual hardware component must be translated to the real driver.



A real 3.46 GHz Intel Xeon processors runs on an emulated BX-chipset: we are running inside a VM.

If you inspect the hardware of a virtual machine in ESX for example, you can see that modern CPUs have to work together with the good but nine years old BX chipset, and that your HBA is always an old bus logic or LSI card. This also means that the newest tricks that your hardware uses to improve performance cannot be used.

Page 7

Memory Management

An OS maintains page tables to translate the virtual memory pages into physical memory addresses. All modern x86 CPUs provide support for virtual memory in hardware. The translation from virtual to physical addresses is performed by the memory management unit, or MMU. The current address is in the CR3 register (hardware page table pointer), and the most used parts of the page table are cached in the TLBs.

It is clear that a guest OS running on a virtual machine cannot have access to the real page tables. Instead, the guest OS sees page tables which run on an emulated MMU. These tables give the guest OS the illusion that it can translate the virtual guest OS addresses into real physical addresses, but in reality the VMM is dealing with this "in the shadow", out of sight of the guest OS. The real page tables are hidden and managed by the VMM and still run on the real MMU. So the real page tables consist of "shadow page tables", which are used to translate the virtual addresses of the guest OS into the real physical pages.

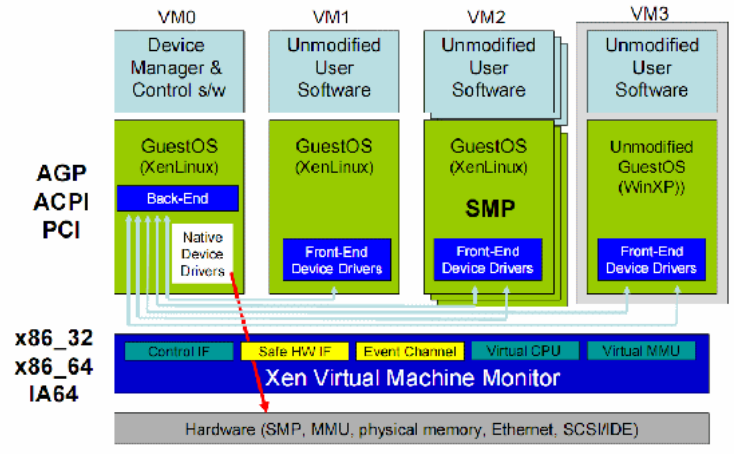
Every time the guest OS modifies its page mapping, the virtual MMU module will capture (trap) the modification and adjust the shadow page tables accordingly. As you've likely guessed, this costs a lot of CPU cycles. Depending on the virtualization technique and the changes made in the page tables, this bookkeeping takes 3 to 400 (!) times more cycles than in the native situation[3]. The result is that in memory intensive applications, memory management causes the largest part of the performance penalty you have to pay for virtualization.

Page 8

Paravirtualization

Paravirtualization is not that different from Binary Translation. BT changes "critical" or "dangerous" code into harmless code on the fly; paravirtualization does the same thing, but in the source code. Of course, changing the source code allows a bit more flexibility than changing everything on the fly, which has to happen quickly. One advantage is that paravirtualization eliminates a lot of unnecessary "traps" by the VMM (or Hypervisor), even more than BT.

The hypervisor provides hypercall interfaces for critical kernel operations such as memory management, interrupt handling, and time keeping. These hypercalls will only happen when it is necessary. For example, most of the memory management is done by the different guest OSes. The Hypervisor will only be "called" for things like page table updates and DMA accesses.

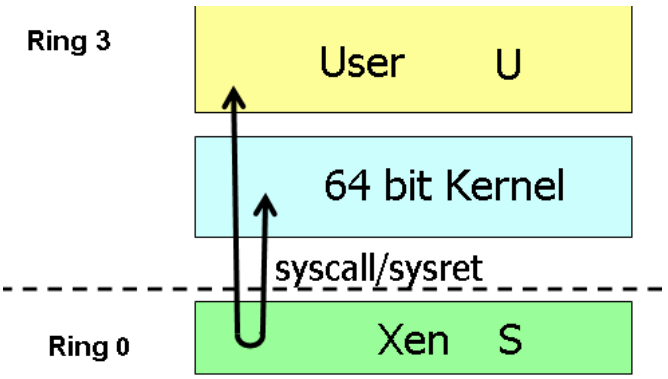


Simplified front end drivers interface to "normal" Linux backend drivers.

The best feature of the Xen implementation of virtualization is the way I/O is handled. I/O devices in the VM are just simplified interfaces that link to real native drivers in a privileged VM (called Domain 0 in Xen). This means there is no emulation involved, and the overhead is significantly reduced. That this is more than Xen propaganda is illustrated by VMware ESX: while VMs running on early ESX versions had rather low network performance, VMs running on ESX 3.x have very acceptable network performance thanks to a cleverly implemented paravirtualized vmxnet network driver.

To resume, paravirtualization is an excellent concept, as you eliminate (binary) translation overhead completely, I/O driver overhead significantly, and system call overhead somewhat. Very frequent interrupts and system calls can still cause overhead. We'll study Xen in more detail in later articles. The biggest disadvantages are:

You cannot use unmodified OS guests (if you use paravirtualization only)
64-bit guests have to run in a non-privileged ring (ring 3). [6]



Software paravirtualization doesn't handle 64-bit OSes very well.

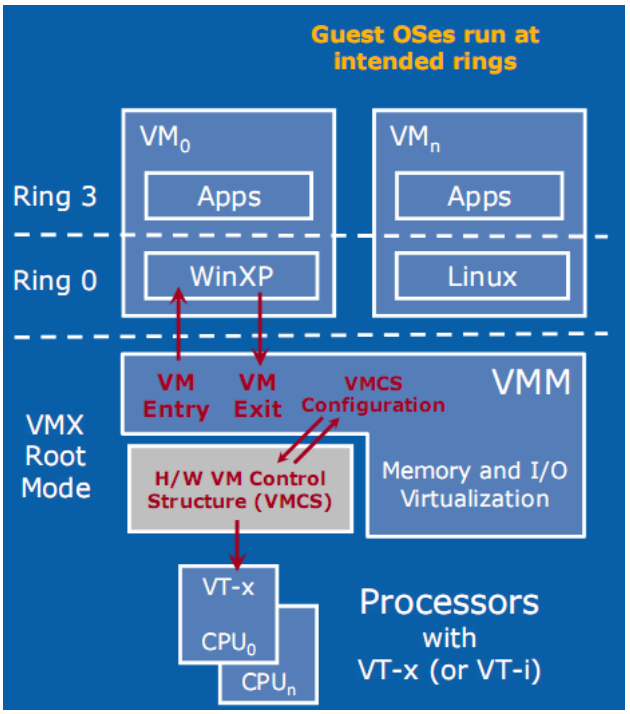
The second problem might not seem like a big problem, but in order to protect the OS, page table switching is necessary. This results in two system calls and a TLB flush, which is very costly. Let us see what Intel's VT-x and AMD-V can offer us.

Page 9

Hardware Accelerated Virtualization: Intel VT-x and AMD SVM

Hardware virtualization should reduce all that overhead to a minimum, right? Unfortunately, that is not the case. Hardware virtualization is not an improved version of binary translation or paravirtualization. No, the first idea behind hardware virtualization is to fix the problem that the x86 instructions architecture cannot be virtualized. This means that hardware virtualization is based on the philosophy of trying to trap all exceptions and privileged instructions by forcing a transition from the guest OS to the VMM, called a "VMexit". You could call this an improved version of the IBM S/370 virtualization methods.

One big advantage is the fact that the guest OS runs at its intended privilege level (ring 0), and that the VMM is running at a new ring with an even higher privilege level (Ring -1, or **"Root mode**"). System calls do not automatically result in VMM interventions: as long as system calls do not involve critical instructions, the guest OS can provide kernel services to the user applications. That is a big plus for hardware virtualization.

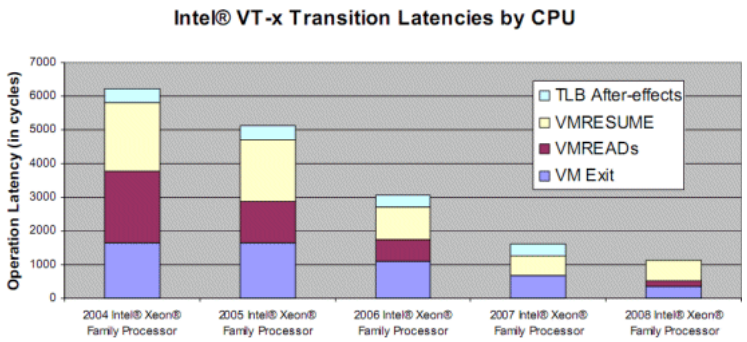


With HW virtualization, the guest OS is back where it belongs: ring 0.

The problem is that - even though it is implemented in hardware - each transition from the VM to the VMM (VMexit) and back (VMentry) requires a fixed (and large) number of CPU cycles. The specific number of these "overhead cycles" depends on the internal CPU architecture. Depending on the exact operation (VMexit, VMentry, VMread, etc.), these kinds of events can take a few hundred up to a few thousand CPU cycles!

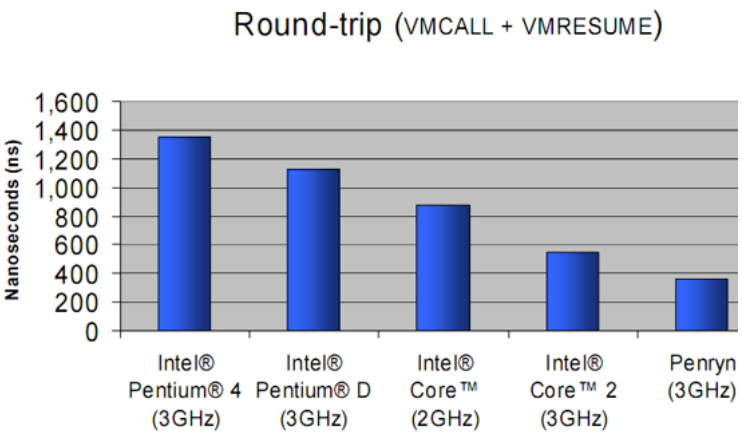
The VM/VMM roundtrip of hardware virtualization is thus a rather heavy event. When Intel VT-x or AMD SVM (or AMD-V) have to handle relatively complex operations such as system calls (which take a lot of CPU cycles to handle anyway), the VMexit/VMentry switching penalty has little impact. On the other hand, if the actual operation that the VMM has to intercept and emulate is rather simple, the overhead of switching back and forth to and from the VMM is huge!

Relatively simple operations such as creating processes, context switches, small page table updates, etc. take a few cycles when run natively, so the "switching to VMM and back" time wastes a proportionally (compared to the non-virtualized native situation) huge amount of cycles. With BT, the translator simply replaces the code with slightly longer code that the VMM handles. The same is true for paravirtualization, which is **a lot faster than hardware virtualization** at handling these kinds of events.



The enter VMM and exit VMM latency has been lowered over time with the different Xeon families.

The first way Intel and AMD countered this problem is to reduce the number of cycles that the VT-x instructions take. For example, the VMentry latency was reduced from 634 (Xeon Paxville or Xeon 70xx) to 352 cycles in the Woodcrest (Xeon 51xx), Clovertown (Xeon 53xx), and Tigerton (Xeon 73xx). As you can see in the graph above, the first implementations of VT-x back in 2005 were not exactly doing wonders for the speed of virtualized machines. The newest Xeon 54xx ("Harpertown") has reduced the typical VMM latencies even more with 12%-25% for the most important ones, and up to 75% for some less frequent instructions. We found a few numbers that are more precise, as you can see below.



Enter and exit VMM numbers (in ns) for the different Intel families.

The second strategy is to reduce the number of VMM events. After all, total virtualization overhead equals the numbers of events times the cost per event. In equation form:

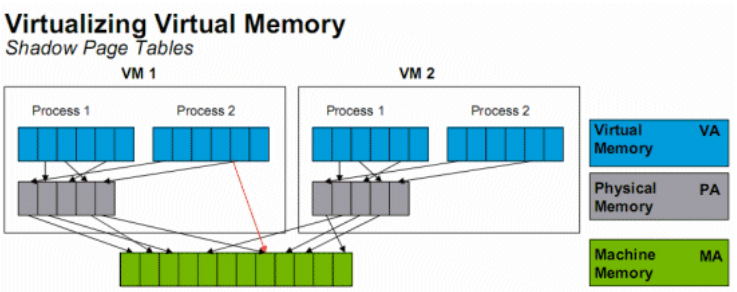
Total VT overhead = Sum of (Frequency of "VMM to VM" events * Latency of event)

The Virtual Machine Control Block - a sort of table that is place in memory (in cache) and which is part of VT-x and AMD SVM - can help. It contains the state of the virtual CPU(s) for each guest OS. It allows the guest OSes to run directly without interference from the VMM. Depending on control bits set in the VMCB, the VMM can allow the guest OS to handle some hardware parts, interrupts, or perform some of the page table operations. The VMM thus configures the VMCS to cause the VM (guest OS) to exit on certain behaviors, while the VMM can let the guest OS continue on others. This can potentially reduce the number of times that the CPU forces the guest OS to stop (VMexit), after which the CPU switches to VMX root mode (ring -1) and the VMM takes over.

Page 10

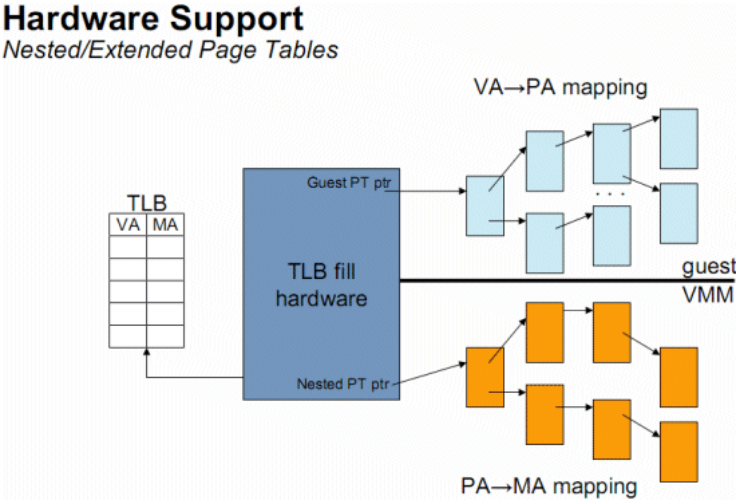
The second generation: Intel's EPT and AMD's NPT

As we discussed in "memory management", managing the virtual memory of the different guest OS and translating this into physical pages can be extremely CPU intensive.



Without shadow pages we would have to translate virtual memory (blue) into "guest OS physical memory" (gray) and then translate the latter into the real physical memory (green). Luckily, the "shadow page table" trick avoids the double bookkeeping by making the MMU work with a virtual memory (of the guest OS, blue) to real physical memory (green) page table, effectively skipping the intermediate "guest OS physical memory" step. There is catch though: each update of the guest OS page tables requires some "shadow page table" bookkeeping. This is rather bad for the performance of software-based virtualization solutions (BT and Para) but wreaks havoc on the performance of the early hardware virtualization solutions. The reason is that you get a lot of those ultra heavy VMexit and VMentry calls.

The second generation of hardware virtualization, AMD's nested paging and Intel's EPT technology partly solve this problem by brute hardware force.



EPT or Nested Page Tables is based on a "super" TLB that keeps track of both the Guest OS and the VMM memory management.

As you can see in the picture above, a CPU with hardware support for nested paging caches both the Virtual memory (Guest OS) to Physical memory (Guest OS) as the Physical Memory (Guest OS) to real physical memory transition in the TLB. The TLB has a new VM specific tag, called the Address Space IDentifier (ASID). This allows the TLB to keep track of which TLB entry belongs to which VM. The result is that a VM switch does not flush the TLB. The TLB entries of the different virtual machines all coexist peacefully in the TLB… provided the TLB is big enough of course!

This makes the VMM a lot simpler and completely annihilates the need to update the shadow page tables constantly. If we consider that the Hypervisor has to intervene for each update of the shadow page tables (one per VM running), it is clear that nested paging can seriously improve performance (up to 23% according to AMD). Nested paging is especially important if you have more than one (virtual) CPU per VM. Multiple CPUs have to sync the page tables often, and as a result the shadow page tables have to update a lot more too. The performance penalty of shadow page tables gets worse as you use more (virtual) CPUs per VM. With nested paging, the CPUs simply synchronize TLBs as they would have done in a non-virtualized environment.

There is only one downside: nested paging or EPT makes the virtual to real physical address translation a lot more complex if the TLB does not have the right entry. For each step we take in the blue area, we need to do all the steps in the orange area. Thus, four table searches in the "native situation" have become 16 searches (for each of the four blue steps, four orange steps).

In order to compensate, a CPU needs much larger TLBs than before, and TLB misses are now extremely costly. If a TLB miss happens in a native (non-virtualized) situation, we have to do four searches in the main memory. A TLB miss then results in a performance hit. Now look at the "virtualized OS with nested paging" TLB miss situation: we have to perform 16 (!) searches in tables located in the high latency system RAM. Our performance hit becomes a performance catastrophe! Fortunately, only a few applications will cause a lot of TLB misses if the TLBs are rather large.

Page 11

Standardization Please!

AMD and Intel are doing it again: incompatible x86 extensions. The specialized hardware virtualization extensions are not standardized. This means that software developers have to develop and support separate modules to support Intel's VT-x and AMD SVM. Xen 3.0.2 supports both technologies, accounting for about 9500 of the lines or 8% of the code base of Xen 3.0.2. AMD's and Intel's lack of standardization is causing the Xen VMM to expand by about 4%, which is still manageable. Let us hope that this doesn't get out of hand, because the difference between AMD's and Intel's extensions is so small that you really need to ask yourself what the point of these two different extensions is.

So which CPU's have support? The table below lists the most important server CPUs.

CPUs with Virtualization Support		
Processor	**Type of Virtualization**	**Degree of support**
Xeon 50xx, Xeon 70xxx and Xeon 71xx	Hardware Virtualization support (HVT only)	Rather slow
Opteron Socket-F, Xeon 53xx	Hardware Virtualization support (HVT only)	Medium
Xeon 54xx	Hardware Virtualization support (HVT only)	Relatively fast
Nehalem, Quad-core Opteron	HVT and Nested paging (EPT/NPT)	Unknown

It important to note that the AMD Opteron 8xx and 2xx with support for DDR do not support HVT; however, they were quite a bit faster than comparable Xeons with early HVT support.

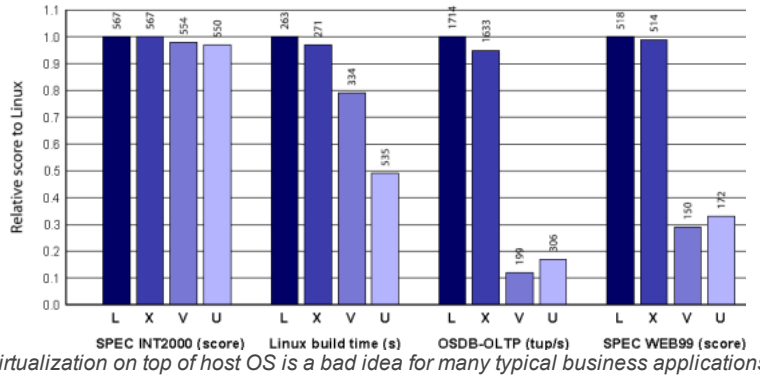
Page 12

The Benchmarks

Performance is somewhat a less popular subject among the virtualization evangelists and little detail is provided in the glossy brochures. A superficial look at most of the published benchmarks seems to justify this lack of interest: who's worried about a 3% to 10% performance loss with the current powerful quad-core CPUs? Indeed, hypervisor or VMM based virtualization - ESX, Xen - perform quite well, especially if you compare it with a typical

host-based solution such as VMware server and Microsoft Virtual Server 2005. The latter run their virtualization layer on top of a host OS, which results in rather low performance.

Look at the graph[6] below. It shows the benchmarks performed by the university of Cambridge. The benchmark compares the native Linux performance (L) with the Xen performance (X) and with VMware Workstation (V) and User Mode Linux. The latter are based on host OS virtualization: the virtualization layer runs on top of a host OS.



Virtualization on top of host OS is a bad idea for many typical business applications.

The most quoted benchmark by the virtualization vendors is SPEC CPU 2000 integer. As you can see in those links, every kind of virtualization technology scores very well. According to these numbers, Xen performs just as well as native. However, once you start a memory intensive benchmark such as a Linux kernel compile, it is clear that the OS hosted virtualization solutions cannot keep up. Throw in OLTP and web applications and performance is simply abysmal.

There is a reason why SPEC CPU 2000 integer is quite popular. It's a CPU intensive benchmark that rarely accesses any other hardware, and it avoids using the OS kernel much of the time. It is also quite remarkable that the "2000" version is used. The 2006 version has a larger memory footprint, which will probably cause a bit more performance loss when virtualized.

Anyway, it is clear that SPEC CPU 2000 integer numbers on virtualized machines prove very little. This kind of software completely avoids the more challenging code that a VMM has to deal with. What happens with Specjbb2005? That server benchmark is also mentioned a lot in the performance numbers of the virtualization vendors. It's true that they show some results, but most Specjbb2005 benchmarks are run one CPU, and if they are run on more than one virtual CPU, only one VM is active. That is of course not very realistic: you do not virtualize your server to run only one VM.

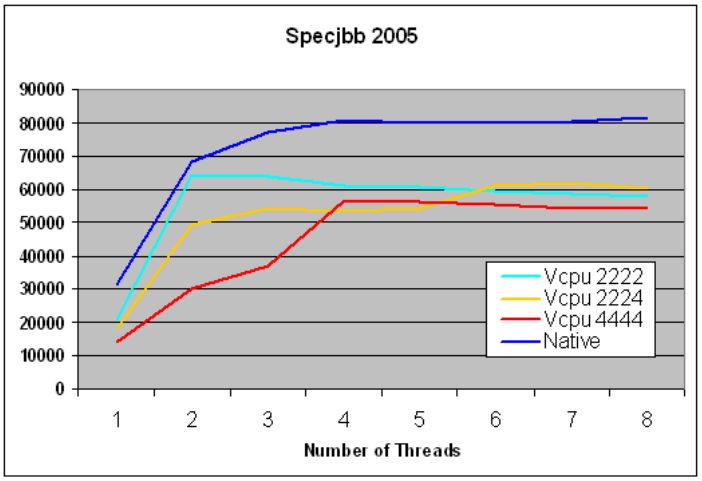
We will keep our full benchmark report for the next article, but let's take a quick look at some of the benchmarks that we ran. We used our quad Xeon MP Intel SR6850HW4 (4 x 3.2GHz dual-core Xeon 7130M, Full configuration here) and ran Specjbb2005 the same way we configured it here. Hyper-Threading was disabled to avoid inconsistencies in our benchmarks.

Virtualization Performance Testing			
Number of VMs	**CPUs per VM**	**SUSE SLES 10 Xen 3.0.3**	**VMware ESX 3.0.2**
1	1	1%	3%
1	4	3%	7%
4	2	5%	15%
4	4	7%	19%

For our impatient readers: yes, there is a fully updated report with the newest Xeons, Opterons, and hypervisors coming (ESX 3.5 etc.). We are well aware that these numbers do not give you a decent picture of the virtualization landscape, but they are not meant to be comprehensive. They are nothing more than a teaser.

However, even these slightly outdated results are very interesting. If you do not pay attention, a "one CPU on one VM" benchmark tells you that virtualization comes without any performance loss whatsoever. However, a much more realistic setup is that you run four virtual machines and you assign two virtual CPUs to each, as there are eight CPU cores available. The performance loss is not dramatic, but it is measurable now.

In the last line (four vCPUs per VM), we still used exactly the same load as in the 2-2-2-2 configuration. At the highest load, we only ran eight threads. The performance loss was on average a bit higher, but the table above is not telling the whole story. Look at the graph below.



If you assign more virtual CPUs than you have, it is important to know that the virtualized performance that you get at lower loads can be a lot lower when you assign one virtual CPU for each real CPU.

Don't get us wrong: the current virtualization products offer very good performance in most cases. Nevertheless, the impression that virtualization comes without any significant performance loss in almost any situation is not accurate either. For example, we have noticed that even a simple OLTP sysbench load loses more than 20% on a very powerful Xeon 5472 server. We have even seen performance losses in 40% range in some cases. It is too early to analyze this as the testing efforts are still in progress, but we feel that more performance research will yield some interesting results.

Page 13

Conclusion

When we first heard about Intel's VT-x and AMD's SVM technology we expected to see performance improvements over the software based solutions such as Binary Translation and Paravirtualization. Both AMD and Intel gave the impression that they were about to "enhance" and "accelerate" the current purely software based solutions.

What AMD and Intel did was extend x86 to make "Classic Virtualization" possible, very similar to the old IBM mainframe virtualization. In other words, hardware virtualization support does not really "enhance" Binary Translation or Paravirtualization; it is a completely different approach. First generation hardware virtualization was even a step back from a performance view, but one that enabled many steps forward. The first generation of virtualization has been improved, and is now adopted by VMware to support 64-bit guest OSes and by Xen to run unmodified OSes (such as Windows). So right now, hardware virtualization still has a long way to go while software virtualization is mature and represents the current standard.

Second generation virtualization (VT-x+EPT and AMD-V+NPT) is more promising, but while it can improve performance significantly it is not guaranteed that it will improve performance across all applications due to the heavy TLB miss cost. On the flip side of the coin: software virtualization is very mature, but there is very little headroom left to improve. The smartest way is to use a hybrid approach, and that is exactly what VMware, Xen, and Microsoft have been doing.

VMware ESX is the best example of this. ESX uses paravirtualized drivers for the most critical I/O components (but not for the CPU), uses emulation for the less important I/O, Binary Translation to avoid the high "trap and emulate" performance penalty, and hardware virtualization for 64-bit guests. In this way, virtualized applications perform quite well, in some cases almost as if there is no extra layer (the VMM).

That doesn't mean that there are no performance issues at all. The huge number of people that populated the numerous VMWorld 2008 sessions about performance and our own (early) benchmark results tell us that the real world performance of virtual servers is still a very interesting challenge despite the multi-core powerhouses they are running on. As long as you are running CPU intensive applications, there is no problem at all - they are running directly after all. However, consider applications with one or more of the following characteristics:

Have a high frequency of system calls or interrupts
Access the memory intensively (DMA)
Perform a lot accesses to I/O devices
Require SMP to perform well

These applications will need more attention to perform well on a virtualized server. Now, it's time for more in-depth benchmarking. Stay tuned….

Bibliography

[1] Robin/Irvine, 9th USENIX Security Symposium Paper 2000: "Analysis of Pentium's Ability to Support a Secure VMM"

- http://www.usenix.org/events/sec00/full_papers/robin/robin_html/index.html

[2] Scott Devine, Co-Founder & Principal Engineer VMware, Inc., "Introduction to Virtual Machines Introduction to Virtual Machines"

[3] Ole Agesen, VMware, "Performance aspects of x86 virtualization", VMWorld 2008

[4] Keith Adams, Ole Agesen, VMware "A Comparison of Software and Hardware Techniques for x86 Virtualization"

[5] Daniel P. Bovet, Marco Cesati, "Understanding the Linux Kernel 3rd Edition" , O'Reilly

November 2005, ISBN: 0-596-00565-2, section 1.6

[6] Paul Barham , Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer , Ian Pratt, Andrew Wareld, "Xen and the Art of Virtualization", University of Cambridge Computer Laboratory