AMD Smarter Choice

AMD Developer Central

Register | Login

Processor-Based Virtualization, AMD64 Style, Part II

Virtualization is the next big hardware breakthrough, and while it's possible by extending standard 32-bit architectures, it's much easier to build virtual systems using AMD64 as a more robust platform. Part 2 in a series explains how AMD's Secure Virtual Machine technology works.

Overview

In the first article in this series, we reviewed how virtualization works, and how a VMM (either a Virtual Machine Manager or Virtual Machine Monitor, your preference) works to allocate hardware resources to guest operating systems on a server, desktop or notebook PC. We saw how some resources are allocated exclusively to a guest operating system, other resources are split on a time-sharing basis, and still others are shared on a cooperative basis, such as a network adapter.

We also saw that there are some challenges for software-based VMMs, because they have trouble intercepting the "privileged" instructions that an operating system tries to pass to the microprocessor. Processors support two types of operations, privileged-mode instructions (sometimes called Ring 0 instructions on the x86/x64 architecture) that give the operating system kernel and device drivers unfettered access to the IO stream, memory and buses, and which are used for things like fetching or storing memory, or assigning memory maps; and user-mode instructions (also known as Ring 3 instructions) which are used by applications.

The operating system can manage how the application's user-mode instructions are executed; they can pass them to the processor directly, they can modify them and then perform the appropriate functions, or they can block them and throw an exception or terminate the application. But how can a VMM perform an analogous process for permitting, blocking or modifying the Ring 0 instructions from its guest operating systems? It has to do so, to ensure that resources are shared or allocated properly, to stop one operating system from stomping all over another operating system's memory, disk and IO, and to enforce security.

Today, VMMs do that through really clever programming. They've found ways to trick the x86/x64 processor and the guest operating systems into letting the VMM have that control; some of those techniques are proprietary to the VMM makers, like VMware or Microsoft. However, they come at a price. It would be better to have what I like to call a "Ring -1" mode which gave the VMM full control over the Ring 0 instructions issued by the guest operating system. However, the x86 architecture doesn't support that – no more rings, sorry. So, chipmakers have had to find another approach.

The solution, developed by Advanced Micro Devices, is AMD-Virtualization™. AMV-V™, is an extension to the AMD64 architecture, introduces two modes, called Host Mode and Guest Mode, and a new instruction, called VMRUN, which wrapper the x86/x64 architecture, within Guest Mode, with that functionality, and let virtual machines (and their guest operating systems and applications, of course) work faster, more efficiently and more securely.

AMD-V has been be introduced in versions of AMD's Athlon 64™ and AMD Turion™ processors already, and is expected to appear in new versions of the AMD Opteron™ processor in the second half of 2006. Virtualization products from VMWare and Microsoft are expected to be able to detect and take advantage of the AMD-V features shortly thereafter. Xen 3.0.2 supports AMD-V today!

Host Mode and VMRUN

An AMD64-based processor boots up in legacy x86 mode in order to maintain compatibility with older 32-bit operating system; the boot loader for a 64-bit operating system throws a one-time switch to active x64 mode, and turn the chip into a 64-bit processor. Similarly, a processor with AMD-V processor will boot up in legacy "guest mode," with all of its extra capabilities dormant – until a compatible VMM is turned on, either as a boot loaded or when activated by a legacy operating system like Windows or Linux, and issues the new VMRUN commands. When that happens, the processor shifts into Host mode.

In Host mode, the VMM has a number of new capabilities. For example, it can define – at the processor level in hardware, rather than in software – the characteristics of each of its guest virtual machines, including which processor and IO resources they can access, and which blocks of real memory they are assigned. The VMM does this by populating a data structure called the VMCB, or Virtual Machine Control Block, which defines all of the processor, memory and IO mappings for each guest operating system instance. For example, the VMM might say that a certain guest operating system instance has exclusive access to a Fibre Channel host adapter and a Gigabit Ethernet network adapter, as well as a 4GB block of memory – and where that physical memory is located.

Once the VMCB has been defined, the VMM switches the system into Guest Mode, passing control of the system to the guest operating system in that virtual machine, which loads up (thinking that it's running on a dedicated server, desktop or notebook). The operating system runs quite merrily, using its own Ring 0 privileged-mode instructions, and handing the Ring 3 instructions and interrupts from its own happy application. La la la, everyone's happy.

In the background, however, the processor's AMD-V circuitry is monitoring everything, particularly the guest operating system's Ring 0 instructions. When one of them calls for a virtualized resourced managed by the VMCB, the processor switches back into Host mode again. Depending on the resource that's been requested, the microprocessor might simply apply the VMCB's data to map the proper resource, or if the request is more complex, would pass control back to the VMM software itself. The VMM would then evaluate the guest operating system's request, deciding whether to permit it, and if so, how to handle the situation – or it might throw an interrupt (such as by passing an error back to the guest operating system) or even shut down that guest operating system if it was particularly naughty (such as trying to hack into the VMM by issuing a VMRUN instruction itself).

In other words, the VMM uses Host mode to manage its multiple guest operating systems in an analogous manner that an operating system manages its multiple applications.

The use of Host mode is more efficient and more secure than virtualization carried out strictly in software. For example, every time a guest operating system causes an interrupt or executes a Ring 0 command, control passes back to Host mode and the VMM takes charge – there's no way to circumvent it. The VMM stays in control, and the AMD-V circuitry keeps it that way.

Also, every time the VMM passes control back to the guest operating system using the VMRUN instruction, this triggers a consistency check. Just to make sure that nothing bad has happened – and if there's a problem, the VMRUN instruction fails, and control passes back to the VMM again, to see what's wrong and remediate the problem.

Because all of this is enabled in hardware, by the way, the work of building VMM systems will become vastly easier – leaving software companies like Xen, VMware and Microsoft freer to focus on adding their own unique value to the virtualization process, and also potentially making it easier for other companies to enter the virtualization market, or to add their own virtualization features of existing applications.

The virtualization engine inside AMD-V mode, by the way, is very robust, and can automatically handle the deployment of 16-bit, 32-bit *and* 64-bit guest operating systems simultaneously. The mappings handled by the VMCB already incorporate that functionality – and also contain the circuitry to allow VMM to intelligently map memory and processor resources on multicore and multiprocessor computers to take best advantage of the HyperTransport bus's NUMA (non-uniform memory access) architecture.

VMMCALL – Asking For More

In all of these discussions, until this point, we have assumed that the guest operating system and its applications are totally unaware that they're running in a virtualized environment. That's been the status quo for modern systems design, whether we're talking Linux, Unix or Windows: virtualization is invisible.

But does it have to be that way? Maybe, maybe not. Imagine if an operating system or its applications *knew* that they were running on a virtualized platform. Imagine if they knew that there were extra hardware resources available. It would be great if an operating system, if it detected that it was running out of physical memory, could see if it was running in a virtual machine, and if so, ask the VMM to give it more memory. Or, if it saw that a disk was getting full, it could ask for more.

Of course, that would only work if the operating system was running in a virtualized environment. To that end, the AMD-V architecture supports a standard Ring 0 privileged mode instruction called VMMCALL, which lets the operating system and VMM communicate directly.

There are a number of new instructions associated with Host mode and VMM design. Most of them are really of interest only for VMM developers – which won't be most of us. However, it is worth noting that the benefits of AMD-V-based virtualization extend beyond simply enabling better hardware utilization through server consolidation (which is the most frequently cited benefit).

Just as certain memory-hungry 32-bit Windows applications run faster under 64-bit Windows than they do on 32-bit Windows (see " Optimizing Applications for 64-Bit Windows, Part I"), so it is possible that even if you're only planning to run one guest operating system on a computer, it might be more efficient to run it under a AMD-V-enhanced VMM, particularly when you're trying to make sure that your hardware resources are fully utilized in a way that a single operating system rarely can achieve, but which a fully laden VMM-based system can really take advantage of. This is purely conjecture on my part, since I haven't any hard experience with such a system, but I wanted to share my thinking.

One of the elements of the AMD-V architecture (or more precisely, the AMD-V extension of the AMD64 architecture) is a new construct called ASID, or application space ID. The VMM can fine-tune the ASID and define it for a particular guest operating system, which gives you (through the VMM) more control over the TLBs than the operating system itself has. Likewise, the VMM has control over shadow paging, and can monitor how the guest operating system edits shadow pages reads and writes. Because the VMM is actually delivering shadow pages, the guest never sees the "real" page tables – and performance can be faster than with the default x86/x64 architecture.

Okay, that's still conjecture; there's much that's still needs to be understood about AMD-V mode, and how virtualization based on how AMD-V performs out of the lab with real servers, real VMMs, real operating systems and real applications. However, I'm bullish about virtualization, particularly in servers; I have been for years, even though all of my hands-on experience has been on software-based systems. I can't to get my hands on hardware-based virtualization using AMD-V, making a great thing even better.

Alan Zeichick is a technology consultant and analyst who focuses on software development and microprocessor technology. Reach him at zeichick@camdenassociates.com.