Volume 10 Issue 03 Published, August 10, 2006 ISSN 1535-864X DOI: 10.1535/itj.1003

Intel® Technology Journal

Intel® Virtualization Technology

Extending Xen* with Intel® Virtualization Technology

More information, including current and past issues of Intel Technology Journal, can be found at:
http://developer.intel.com/technology/itj/index.htm

Extending Xen* with Intel® Virtualization Technology

Yaozu Dong, Core Software Division, Intel Corporation
Shaofan Li, Core Software Division, Intel Corporation
Asit Mallick, Core Software Division, Intel Corporation
Jun Nakajima, Core Software Division, Intel Corporation
Kun Tian, Core Software Division, Intel Corporation
Xuefei Xu, Core Software Division, Intel Corporation
Fred Yang, Core Software Division, Intel Corporation
Wilfred Yu, Core Software Division, Intel Corporation

Index words: Xen, Virtualization, Hypervisor, Intel® VT, virtual machine monitor

ABSTRACT

Xen* is an open source virtual machine monitor (VMM) developed at the University of Cambridge to support operating systems (OSs) that have been modified to run on top of the monitor. Intel has extended the Xen VMM to use Intel® Virtualization Technology$^{\Delta}$ (VT) to support unmodified guest OSs also. This was done for IA-32 Intel® Architecture processors as well as Itanium® architecture processors.

In this paper we describe the changes that have been made to Xen to enable this support. We also highlight the optimizations that have been made to date to deliver good virtualized performance.

INTRODUCTION

Xen is an open source virtual machine monitor (VMM) that allows the hardware resources of a machine to be virtualized and dynamically shared between OSs running on top of it [1]. Each virtual machine (VM) is called a Domain, in Xen terminology. Xen provides isolated execution for each domain, preventing failures or malicious activities in one domain from impacting another domain. The Xen hypervisor and Domain0 (Dom0) are a required part of any Xen-based server. Multiple user domains, called DomainU in Xen terminology, can be created to run guest OSs.

Unlike the full virtualization solutions offered by the IBM VM/370*, or VMware's ESX* and Microsoft's Virtual PC product*, Xen began life as a VMM for guest OSs that have been modified to run on the Xen hypervisor. User applications within these OSs run as is, i.e., unmodified. This technique is called "paravirtualization," and it delivers near native performance for the guest OS, only if the guest OSs source code can be modified.

Xen versions 1.0 and 2.0 use paravirtualization techniques to support 32-bit platforms and Linux* guests. They use the standard IA-32 protection and segmentation architecture for system resource virtualization. The hypervisor runs in the highest privilege level *ring 0* and has full access to all memory on the system. Guest OSs use privilege levels 1, 2, and 3 as they see fit. Segmentation is used to prevent the guest OS from accessing the Xen address space.

Xen 3.0 is the first open-source VMM that uses Intel Virtualization Technology (VT) to support unmodified guest OSs as well as paravirtualized guest OSs. Xen 3.0 also added support for 64-bit platforms and 64-bit guests [9]. Page-level protection is used to protect the 64-bit hypervisor from the guest.

In this paper, we begin with a brief overview of Intel VT and then we explain how we extended Xen to take advantage of VT. We highlight key virtualization issues for IA-32, Intel® EM64T$^{\Phi}$, and Itanium processors and explain how they are addressed in Xen 3.0. Finally, we highlight some of the changes that have been made to the hypervisor and the device models to improve performance.

INTEL® VIRTUALIZATION TECHNOLOGY

Intel VT is a collection of processor technologies that enables robust execution of unmodified guest OSs on Intel VT-enhanced VMMs [2]. VT-x defines the

extensions to the IA-32 Intel Architecture [3]. VT-i defines the extensions to the Intel Itanium architecture [4].

VT-x augments IA-32 with two new forms of CPU operation: virtual machine extensions (VMX) root operations and VMX non-root operations. The transition from VMX root operation to VMX non-root operation is called a VM entry. The transition from a VMX non-root operation to VMX root operation is called a VM exit.

A virtual-machine control structure (VMCS) is defined to manage VM entries and exits, and it controls the behavior of instructions in a non-root operation. The VMCS is logically divided into sections, two of which are the guest-state area and the host-state area. These areas contain fields corresponding to different components of processor state. VM entries load processor state from the guest-state area. VM exits save processor state to the guest-state area and then load processor state from the host-state area.

The VMM runs in root operation while the guests run in VMX non-root operation. Both forms of operation support all four privilege levels (i.e., rings 0, 1, 2, and 3). The VM-execution control fields in the VMCS allow the VMM to control the behavior of some instructions in VMX non-root operation and the events that will cause VM exits. Instructions like CPUID, MOV from CR3, RDMSR, and WRMSR will trigger VM exits unconditionally to allow the VMM to control the behavior of the guest.

VT-i expands the Itanium processor family (IPF) to enable robust execution of VMs. A new processor status register bit (PSR.vm) has been added to define a new operating mode for the processor. The VMM runs with this bit cleared while the guest OS runs with it set. Privileged instructions, including non-privileged instructions like *thash, ttag* and *mov cupid* that may reveal the true operating state of the processor, trigger virtualization faults when operating in this mode.

The PSR.vm bit also controls the number of virtual-address bits that are available to software. When a VMM is running with PSR.vm = 0, all implemented virtual-address bits are available. When the guest OS is running with PSR.vm = 1, the uppermost implemented virtual-address bit is made unavailable to the guest. Instruction or data fetches with any of these address bits set will trigger unimplemented data/instruction address faults or unimplemented instruction address traps. This provides the VMM a dedicated address space that guest software cannot access.

VT-i also defines the processor abstraction layer (PAL) interfaces that can be used by the VMM to create and manage VMs. A Virtual Processor Descriptor (VPD) is defined to represent the resources of a virtual processor. PAL procedures are defined to allow the VMM to configure logical processors for virtualization operations and to suspend or resume virtual processors. PAL run-time services are defined to support performance-critical VMM operations.

EXTENDING XEN* WITH INTEL VT

Xen 3.0 architecture (Figure 1) has a small hypervisor kernel that deals with virtualizing the CPU, memory, and critical I/O resources, such as the interrupt controller. Dom0 is a paravirtualized Linux that has privileged access to all I/O devices in the platform and is an integral part of any Xen-based system. Xen 3.0 also includes a control panel that controls the sharing of the processor, memory, network, and block devices. Access to the control interface is limited to Dom0. Multiple user domains, called DomainU (DomU) can be created to run paravirtualized guest OSs. Dom0 and DomU OSs use hypercalls to request services from the Xen hypervisor.

When Intel VT is used, fully virtualized domains can be created to run unmodified guest OSs. These fully virtualized domains are given the special name of HVMs (hardware-based virtual machines). Xen presents to each HVM guest a virtualized platform that resembles a classic PC/server platform with a keyboard, mouse, graphics display, disk, floppy, CD-ROM, etc. This virtualized platform support is provided by the Virtual I/O Devices module.

In the following sections we describe the extensions to each of these Xen components.

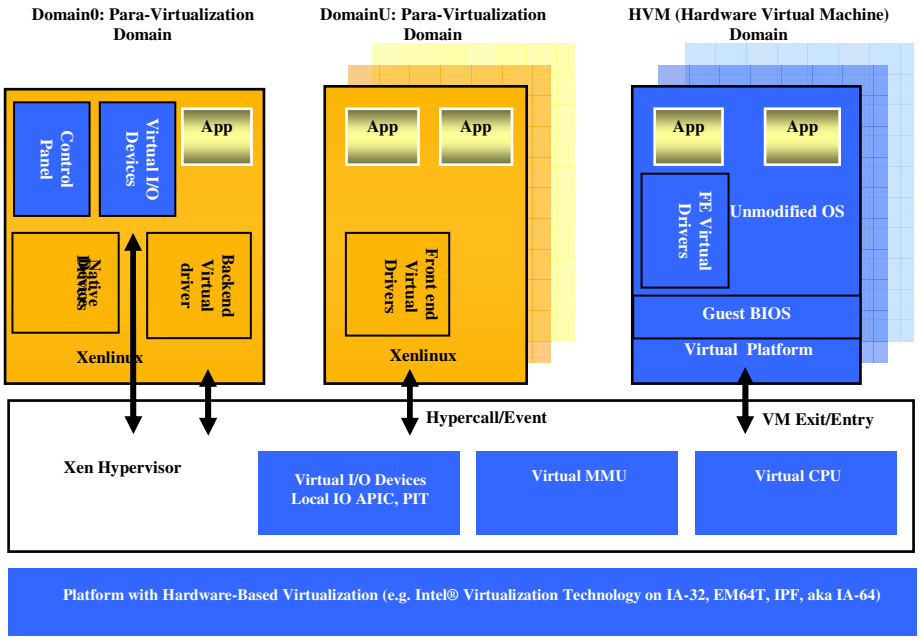

Figure 1: Xen 3.0 architecture

Control Panel

We have extended the control panel to support creating, controlling, and destroying HVM domains. The user can specify configuration parameters such as the guest memory map and size, the virtualized disk location, network configuration, etc.

The control panel loads the guest firmware into the HVM domain and creates the device model thread (explained later) that will run in Dom0 to service input/output (I/O) requests from the HVM guest. The control panel also configures the virtual devices seen by the HVM guest, such as the interrupt binding and the PCI configuration.

The HVM guest is then started, and control is passed to the first instruction in the guest firmware. The HVM guest executes at native speed until it encounters an event that requires special handling by Xen.

Guest Firmware

The guest firmware (BIOS) provides the boot services and run-time services required by the OS in the HVM. This guest firmware does not see any real physical devices. It operates on the virtual devices provided by the device models.

For VT-x, we are re-using the open source Bochs BIOS [5]. We extended the Bochs BIOS by adding Multi-Processor Specification (MPS) tables [6], Advanced Configuration and Power Interface (ACPI) tables [7], including the Multiple APIC Description Table (MADT). The BIOS and the early OS loader expect to run in real mode. To create the environment needed by these codes, we use VMXAssist to configure the VT-x guest to execute in virtual-8086 mode. Instructions that cannot be executed in this mode are intercepted and emulated with a software emulator.

For VT-i, we developed a guest firmware using the Intel® Platform Innovation Framework for Extensible Firmware Interface (EFI). This guest firmware provides all EFI boot services required by IPF guest OSs. It is compatible with the Developer's Interface Guide for 64-bit Intel® Architecture-based Servers (DIG64) and provides the System Abstraction Layer (SAL), ACPI 2.0, and EFI 1.10 tables required by IPF guest OSs.

Processor Virtualization

The Virtual CPU module in Xen provides the abstraction of a processor to the HVM guest. It manages the virtual processor(s) and associated virtualization events when the guest OS is executing. It saves the physical processor state when the guest gives up a physical CPU, and restores the guest state when it is rescheduled to run on a physical processor.

For the IA-32 architecture, a VMCS structure is created for each CPU in a HVM domain (Figure 2). The execution control of the CPU in VMX mode is configured as follows:

- Instructions such as CPUID, MOV from/to CR3, MOV to CR0/CR4, RDMSR, WRMSR, HLT,

INVLPG, MOV from CR8, MOV DR, and MWAIT are intercepted as VM exits.

- Exceptions/faults, such as page fault, are intercepted as VM exits, and virtualized exceptions/faults are injected on VM entry to guests.
- External interrupts unrelated to guests are intercepted as VM exits, and virtualized interrupts are injected on VM entry to the guests.
- Read shadows are created for the guest CR0, CR4, and time stamp counter (TSC). Read accesses to such registers will not cause VM exit, but will return the shadow values.

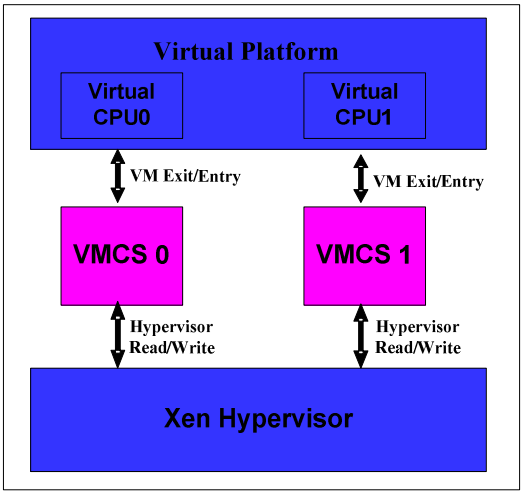

Figure 2: VMCS

For the Itanium architecture, a Virtual Processor Block (VPD) structure is created for each CPU in a HVM domain. The VPD has similar functionality as the VMCS in the IA-32 architecture. The virtualization control of the CPU is configured as follows:

- Instructions such as MOV from/to RR, MOV from/to CR, ITC/PTC, ITR/PTR, MOV from/to PKR, MOV from/to IBR/DBR are intercepted as virtualization faults.
- Instructions such as COVER, BSW are optimized to execute without virtualization faults.
- Exceptions/faults are intercepted by the VMM, and virtualized exceptions/faults are injected to the guest on a VM resume.
- External interrupts are intercepted by the VMM, and virtualized external interrupts are injected to the guest using the virtual external interrupt optimization.
- Read shadows are created for the guest interruption control registers, PSR, CPUID. Read accesses to such registers will not cause virtualization fault, but will return the shadow values.
- Write shadows are created for the guest interruption control registers. Write accesses to such registers will not cause virtualization fault, but will write to the shadow values.

An interesting question when designing Xen concerns the processor features that are exposed to HVM guests. Some VMMs present only a generic, minimally featured processor to the guest. This allows the guest to migrate easily to arbitrary platforms, but precludes the guest from using new instructions or processor features that may exist in the processor. For Xen, we are exporting most CPUID bits to the guest. We clearly need to clear the VMX bit [Leaf 1, ECX:5], or else the guest may bring up another level of virtualization. Other bits to be cleared include machine check architecture (MCA), because MCA issues are handled by the hypervisor. Today's OSs also use model-specific registers to detect the microcode version on the processor and to decide whether they need to perform a microcode update. For Xen, we decided to fake the update request, i.e., bump the microcode version number without changing the microcode itself.

Memory Virtualization

The virtual Memory Management Unit (MMU) module in the Xen hypervisor presents the abstraction of a hardware MMU to the HVM domain. HVM guests see guest physical addresses (GPAs), and this module translates GPAs to the appropriate machine physical addresses (MPAs).

IA-32 Memory Virtualization

The virtual MMU module supports all page table formats that can be used by the guest OS.

- For IA-32
    a. it supports 2-level page tables with 4 KB page size for 32-bit guests.
- For IA-32 Physical Address Extension (PAE)
    a. it supports 2-level page tables with 4 KB page sizes for 32-bit guests.
    b. it supports 3-level page tables with 4 KB and 2 MB page sizes for 32-bit PAE guests.
- For Intel EM64T
    a. it supports 2-level page tables with 4 KB page size for 32-bit guests.

b. it supports 3-level page tables with 4 KB and 2 MB page sizes for 32-bit PAE guests.

c. it supports 4-level page tables with 4 KB and 2 MB page sizes for 64-bit guests.

For the IA-32 architecture, this module maintains a shadow page table for the guest (Figure 3). This is the actual page table used by the processor during VMX operation, containing page table entries (PTEs) with machine page-frame numbers. Every time the guest modifies its page mapping, either by changing the content of a translation, creating a new translation, or removing an existing translation, the virtual MMU module will capture the modification and adjust the shadow page tables accordingly. Since Xen already has shadow page table code for paravirtualized guests, we extended the code to support fully virtualization guests. The resultant code handles paravirtualized and unmodified guests in a unified fashion.

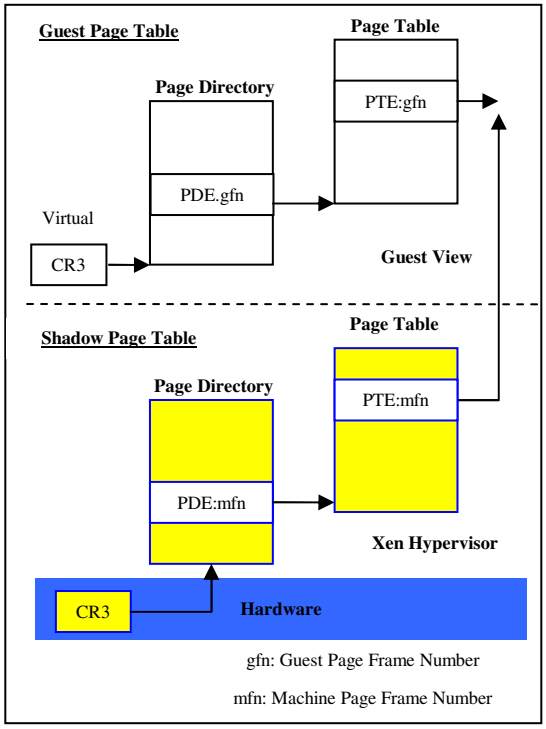

Figure 3: Shadow page table

From a performance point of view, the shadow page table code is the most critical for overall performance. The most rudimentary implementation includes the construction of shadow page tables from scratch every time the guest updates CR3 to request a TLB flush. This, however, will incur significant overhead. If we can tell which guest page table entries have been modified, we just need to clean up the affected shadow entries, allowing the existing shadow page tables to be reused.

The following algorithm is used to optimize shadow page table management:

- When allocating a shadow page upon page fault from the guest, write protect the corresponding *guest page table page*. This allows you to detect any attempt to modify the guest page table. For this to work, you need to find all translations that map the guest page table page. There are several optimizations for this as discussed below.
- Upon page fault against a guest page table page, save a "snapshot" of the page and give write permission to the page. The page is then added to an "out of sync" list with the information on such an attempt (i.e., which address, etc.). Now the guest can continue to update the page.
- When the guest executes an operation that results in the flush TLB operation, reflect all the entries on the "out of sync list" to the shadow page table. By comparing the snapshot and the current page in the guest page table, you can update the shadow page table efficiently by checking if the page frame numbers in the guest page tables are valid (i.e., contained in the domain).

Itanium Processor Architecture Memory Virtualization

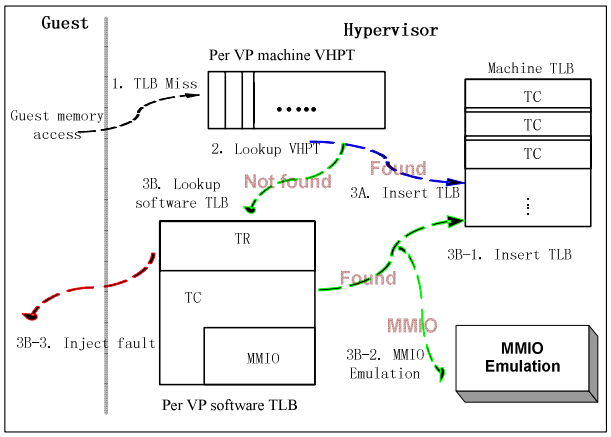

Figure 4: IPF TLB virtualization

The Itanium processor architecture defines Translation Register (TR) entries that can be used to statically map a range of virtual addresses to physical addresses. Translation Cache (TC) entries are used for dynamic mappings. Address translation entries can reside in either the TLB or in a Virtual Hash Page Table (VHPT). On a TLB miss, a hardware engine will walk the VHPT to extract the translation entry for the referenced address and insert the translation into the TLB.

Figure 4 illustrates the TLB virtualization logic in Xen. We extended the Xen hypervisor to capture all TLB

insertions and deletions initiated by a guest OS. This information is used to maintain the address translation for the guest. Two new data structures are added to Xen:

- The Machine VHPT is a per virtual CPU data structure. It is maintained by the hypervisor and tracks the translations for guest TR and TC entries mapping normal memory. It is walked by the hardware VHPT walker on a TLB miss.

  The Itanium processor architecture defines two formats for the VHPT. The short-format VHPT is meant to be used by an OS to implement linear page tables. The long-form VHPT has a larger foot print but supports protection keys and collision chains. We have extended the Xen hypervisor to use the long-form VHPT.

- The guest software TLB structure is used to track guest TRs and TCs mapping memory mapped I/O addresses or less than preferred page table entries. Access to these addresses must be intercepted and forwarded to the device model.

Region Identifier (RID) is an important component of the Itanium architecture virtual memory management system. It is used to uniquely identify a region of virtual address. Per Itanium architecture specifications, RID should have at least 18 bits and at most 24 bits. The exact number of RID bits implemented by a processor can be found by using the PAL_VM_SUMMARY call. An address lookup will require matching the RID as well as the virtual address.

Each IPF guest OS thinks it has unique ownership of the RIDs. If you allow two VT-i domains to run on the same processor with the same RID, you need to flush the machine TLB whenever a domain is switched out. This will have a significant negative impact on system performance.

The solution we used for Xen is to partition the RIDs between the domains. Specifically, we reserved several high-order bits from the RID as the guest identifier. The machine RID used for the guest is then a concatenation of the guest ID and the RID managed by the guest itself.

Machine_rid=guest_rid + (guest_id << 18)

As an illustration, if we have a CPU that support a 24-bit RID, the guest firmware inside the VT-i guest will report only 18-bit RID to the guest. The actual 24-bit RID installed into the machine will have the guest identifier in the upper 6-bit.

We also need two more RIDs per domain for guest physical mode emulation. The guest physical mode accesses are emulated by using a virtual address with special RIDs. This restricts the total number of IPF guests to 63.

This is a reasonable solution when the number of concurrent guests is limited and the guests are not running millions of processes concurrently. A more elaborate scheme is needed if this assumption is not true.

Device Virtualization

Figure 5 illustrates the device virtualization logic in Xen. The Virtual I/O devices (device models) in Dom0 provide the abstraction of a PC platform to the HVM domain. Each HVM domain sees an abstraction of a PC platform with a keyboard, mouse, real-time clock, 8259 programmable interrupt controller, 8254 programmable interval timer, CMOS, IDE disk, floppy, CDROM, and VGA/graphics.

To reduce the development effort, we reuse the device emulation module from the open source QEMU project [8]. Our basic design is to run an instance of the device models in Dom0 per HVM domain. Performance critical models like the Programmable Interrupt Timer (PIT) and the Programmable Interrupt Controller (PIC), are moved into the hypervisor.

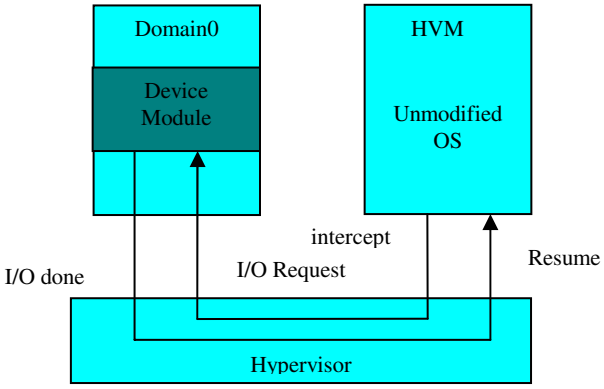

Figure 5: I/O Device virtualization

The primary function of the device model is to wait for an I/O event from the HVM guest and dispatch it to the appropriate device emulation model. Once the device emulation model completes the I/O request, it will respond back with the result. A shared memory between the device model and the Xen hypervisor is used for communication of I/O request and response.

The device model utilizes Xen's event channel mechanism and waits for events coming from the HVM domain via an event channel, with appropriate timeouts to support the internal timer mechanisms within these emulators.

I/O Port Accesses
We set up the I/O bitmap to intercept I/O port accesses by the guest. At each such VM exit, we collect exit qualification information such as port number, access size, direction, string or not, REP prefixed or not, etc. This information is packaged as an I/O request packet and sent to the device model in Dom0.

Following is an example of an I/O request handling from a HVM guest:

1. VM exit due to an I/O access.
2. Decode the instruction.
3. Make an I/O request packet (ioreq_t) describing the event.
4. Send the event to the device model in Dom0.
5. Wait for response for the I/O port and MMIO operation from the device model.
6. Unblock the HVM domain.
7. VMRESUME back to the guest OS.

Although this design significantly reduced our development efforts, almost all I/O operations require domain switches to Dom0 to run the device model, resulting in high CPU overhead and I/O latencies. To give HVM domains better I/O performance, we also ported Xen's Virtual Block Device (VBD) and Virtual Network Interface (VNIF) to HVM domains.

Memory-Mapped I/O Handling
Most devices require memory-mapped I/O to access the device registers. Critical interrupt controllers, such as I/O APIC, also require memory-mapped I/O access. We intercept these MMIO accesses as page faults.

On each VM exit due to page fault, you need to do the following:

- Check the PTE to see if the guest page-frame belongs to the MMIO range.
- If so, decode the instruction and send an I/O request packet to the device model in Dom0.
- Otherwise, hand the event to the shadow page code for handling.

The Itanium processor family supports memory-mapped I/O only. It implements the above logic in the page fault handler.

Interrupts Handling
The real local APICs and I/O APICs are owned and controlled by the Xen hypervisor. All external interrupts will cause VM exits. Interrupts owned by the hypervisor (e.g., the local APIC timer) are handled inside the hypervisor. Otherwise the handler in Dom0 is used if the interrupt is not used by the hypervisor. This way the HVM domain does not handle real external interrupts.

The HVM guests only see virtualized external interrupts. The device models can trigger a virtual external interrupt by sending an event to the interrupt controller (PIC or APIC) device model. The interrupt controller device model then injects a virtual external interrupt to the HVM guest on the next VM entry.

Virtual Device Drivers
The VBD and VNIF are based on a split driver pair where the front-end driver runs inside a guest domain while the backend driver runs inside Dom0 or an I/O VM. To port these drivers to HVM domains, we have to solve two major challenges:

1. Define a way to allow the hypervisor to access data inside the guest, based on a guest virtual address.

   We solved this problem by defining a copy_from_guest() hypercall that will walk the guest's page table and map the resulting physical pages into the hypervisor address space.

2. Define a way to signal Xen events to the virtual drivers. This must be done in a way that is consistent with the guest OSs device driver infrastructure.

   We solved this problem by implementing the driver as a fake PCI device driver with its own interrupt vector. This vector is communicated to the hypervisor via a hypercall. Subsequently, the hypervisor will use this vector to signal an event to the virtual device driver.

The send performance of the VNIF ported this way approximates that of the VNIF running in paravirtualized DomU. The receive throughput is lower. We are continuing our investigation.

PERFORMANCE TUNING VT-X GUESTS

In this section we describe the performance tuning exercise done to date for VT-x guests. The classic approach is to run a synthetic workload inside an HVM domain and compare the performance against the same workload running inside an identically configured paravirtualized domain. But to understand why the domain operates the way it does, we have to extend tools like Xentrace and Xenoprof to support HVM domains also.

Xentrace is a tool that can be used to trace events in the hypervisor. It can be used to count the occurrence of key

events and their handling time. We extended this tool to trace VT-x specific information such as VM exits, recording the exit cause and the handling time.

Xenoprof is a port of OProfile to the Xen environment. It is a tool that uses hardware performance counters to track clock cycle count, instruction retirements, TLB misses, and cache misses. Each time a counter fires, Xenoprof samples the program counter, thus allowing a profile to be built for the program hotspots. The original Xenoprof supports paravirtualized guests only. We extended this tool to support HVM domains.

A typical tuning experiment proceeds as follows:

1. Run a workload and use Xentrace to track the VM exit events occurring during the run.
2. Run a workload and use Xenoprof to profile the hotspots in the hypervisor.

We observed the bulk of the exits is caused by I/O instruction or shadow page table operations. I/O instructions have the longest handling time, requiring a context switch to Dom0. At one stage of our tuning experiment, 40% of the hypervisor time was spent in the shadow code.

Based on the above findings, we focused on tuning the I/O handler code and improving the shadow page handling.

- From the Xentrace result, we observed that the majority of the guest I/O accesses are to the PIC ports. This is because the guest timer handler needs regular access to PIC ports. By moving the PIC model to the hypervisor, we dramatically reduced the PIC handling time. Kernel build performance improved 14% and the CPU2k benchmark improved by 7%.
- The original QEMU IDE model handles IDE DMA operations in a synchronous fashion. When a guest starts an IDE DMA operation, the QEMU model will wait for the host to complete the DMA request. We added a new thread to handle DMA operations in an asynchronous fashion. This change increased guest kernel build performance by 8%.
- The original QEMU NIC model is implemented using a polling loop. We changed the code to an event driven design that will wait on the packet file descriptors. This change improved SCP performance by 10–40 times.
- The original QEMU VGA model emulated a graphics card. When the guest updates the screen, each video memory write causes a VM exit, and pixel data have to be forwarded to a VGA model in Dom0. To speed up graphics performance, we implemented a shared memory area between the QEMU model and the HVM guest. Guest video memory write will no longer cause a VM exit. The VGA model will update the screen periodically using data in the shared memory area. This improved XWindow performance dramatically by 5–1000 times.

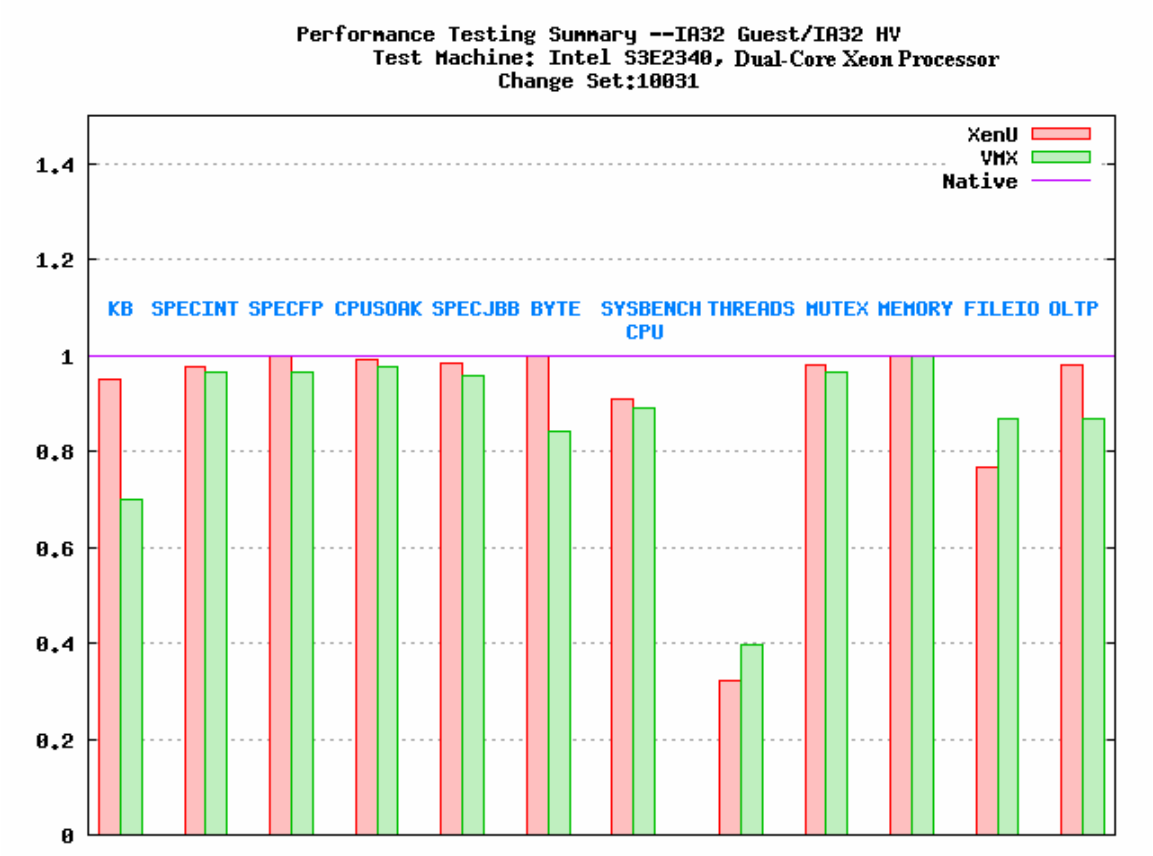

Figure 6: Performance comparison of paravirtualized vs. VT-x domain

BENCHMARK PERFORMANCE

Figure 6 compares the system performance results reported by various benchmarks when running in an identically configured paravirtualized domain and a VT-x domain. The performance of the same benchmark in a native environment is used as a reference. The data are collected on an Intel® S3E2340 platform, with 2.3 GHz/800 MHz FSB dual-core Intel® Xeon® processor, 4 GB of DDR2 533 MHz memory, a 160 GB Seagate SATA disk and an Intel® E100 Ethernet controller. RHEL4U1 is used as the OS in Dom0, DomU, and VT-x domains. Dom0 is configured with two virtual CPUs and 512 MB of memory. DomU and the VT-x domains are configured with a single virtual CPU with 512 M of memory and a 20 GB physical partition as its virtual disk.

CURRENT STATUS

As of this writing, Xen is under active development by Intel and various partners in the community. Readers interested in the latest status should consult the xen-devel* or xen-user* mailing list. Novell and RedHat are incorporating Xen into their upcoming releases. Virtual Iron and XenSource are developing products that will leverage Xen and Intel Virtualization Technology.

ACKNOWLEDGMENTS

The work described in this paper was made possible by many people. We thank our management for supporting this work. We acknowledge the many past and present members of the Xen team in the Intel® Open Source Technology Center for the many hours they spent developing and testing this code. Felix Leung, Alberto Munoz, and Mary Xie have provided immeasurable help for the VT-i project. A special thanks goes to Ian Pratt and Keir Fraser for working the full virtualization issues with us. Leendert van Doorn is the creator of the VMXAssist logic to execute real mode code. Their guidance and assistance throughout the course of this project has been invaluable.

REFERENCES

[1] Pratt, Ian; Fraser, Keir; Hand, Steve; Limpach, Christian; Warfield, Andrew; Magenheimer, Dan; Nakajima, Jun; Mallick, Asit; "Xen 3.0 and the Art of Virtualization," in *Proceedings of Linux Symposium, Volume Two*, 2005.

[2] Uhlig, R.; Neiger, G.; Rodgers, D.; Santoni, A.L.; Martins, F.C.M.; Anderson, A.V.; Bennett, S.M.; Kagi, A.; Leung, F.H.; Smith, L.; "Intel Virtualization Technology," *IEEE Computer Volume 38, Issue 5*, pp. 48–56, May 2005.

[3] Intel Virtualization Technology Specification for the IA-32 Architecture, at www.intel.com/technology/vt/, Intel Corp.

[4] Intel Virtualization Technology Specification for the Intel Itanium Architecture, at www.intel.com/technology/vt/, Intel Corp.

[5] Bochs IA-32 Emulator project, at http://bochs.sourceforge.net/*

[6] MultiProcessor Specification, at http://developer.intel.com/design/pentium/datashts/24201606.pdf, Intel Corp., Version 1.4, May 1997.

[7] Hewlett-Packard Corporation; Intel Corporation; Microsoft Corporation; Phoenix Technologies Ltd.; Toshiba Corporation; *Advanced Configuration and Power Interface Specification, Revision 3.0*, September 2, 2004.

[8] QEMU at http://fabrice.bellard.free.fr/qemu*

[9] Nakajima, Jun; Mallick, Asit; Pratt, Ian; Fraser, Keir, "X86-64 XenLinux: Architecture, Implementation, and Optimizations," *Ottawa Linux Symposium*, 2006.

AUTHORS' BIOGRAPHIES

Yaozu Dong is a technical lead in the Open Source Technology Center in Shanghai, PRC. He joined Intel in 1998 and had been involved in various embedded system projects from PalmOS* to Windows CE* to Linux, and several virtualization projects. He received his Bachelors and Masters degrees in Engineering from Shanghai Jiao Tong University, PRC. His e-mail address is eddie.dong at intel.com.

Shaofan Li is an engineering manager in the Open Source Technology Center in Shanghai, PRC. She joined Intel in 1999 and had been involved in IPMI, EFI, and several virtualization projects. She currently manages the Xen development team of PRC in the Intel Open Source Technology Center. Her team is focusing on enabling Intel Virtualization Technology in Xen for both IA-32 and Itanium architectures. She received her Bachelors and Masters degrees in Engineering from Shanghai Jiao Tong University, PRC. Her e-mail address is susie.li at intel.com.

Asit Mallick is a senior principal engineer leading the system software architecture in the Intel Open Source Technology Center. He joined Intel in 1992 and has worked on the development and porting of numerous operating systems to Intel architecture. Prior to joining Intel, he worked in Wipro Infotech, India on the development of networking software. Asit earned his Masters degree in Engineering from the Indian Institute of Science, India. His e-mail address is asit.k.mallick at intel.com.

Jun Nakajima is a principal engineer leading Linux and Xen projects at the Intel Open Source Technology Center. He is recognized as one of the key contributors to Xen, including Xen/XenLinux port to Intel EM64T, the major VT-x support codes, and the architecture. He has over 15 years of experience with operating system internals and an extensive background in processor architectures. Prior to joining Intel, he worked on various projects in the industry such as AT&T/USL Unix System V Releases (SVR) like the SVR4.2, and Chorus microkernel based fault-tolerant distributed SVR4. Jun earned his Bachelors of Engineering degree from the University of Tokyo in Japan. His e-mail is jun.nakajima at intel.com.

Kun Tian is a software engineer in the Open Source Technology Center in Shanghai, PRC. He joined Intel in 2003 and has been involved in Linux kernel development and virtualization-related projects. He is currently working on adding Intel Virtualization Technology to Xen for Itanium processor. He received his Masters degree in Engineering from the University of Electronic Science and Technology of China. His e-mail is kevin.tian at intel.com.

Xuefei Xu is a software engineer in the Open Source Technology Center in Shanghai, PRC. He joined Intel in 2003 and had been involved in several virtualization projects. He currently is working on adding Intel Virtualization Technology to Xen for Itanium. He received his Masters degree in Engineering from the Huazhong University of Science and Technology, PRC. His e-mail is anthony.xu at intel.com.

Fred Yang is a project lead in the Intel Open Source Technology Center in Santa Clara, California. He joined Intel in 1989 and had been involved in a series of operating system projects for Intel processors. He currently leads the team that is adding Intel Virtualization Technology to Xen for Itanium. He received his M.S. degree in Computer Science from the University of Texas at Arlington. His e-mail is fred.yang at intel.com.

Wilfred Yu is an engineering manager in the Intel Open Source Technology Center in Santa Clara, California. He joined Intel in 1983 and had been involved in a series of operating system projects for Intel processors. He currently manages the team that is adding Intel

Virtualization Technology to Xen. He received his Bachelors degree in Engineering from McGill University and his Masters of Applied Science and PhD degrees from the University of Waterloo, Canada. His e-mail is wilfred.yu at intel.com.

[Δ] Intel® Virtualization Technology requires a computer system with an enabled Intel® processor, BIOS, virtual machine monitor (VMM) and, for some uses, certain platform software enabled for it. Functionality, performance or other benefits will vary depending on hardware and software configurations and may require a BIOS update. Software applications may not be compatible with all operating systems. Please check with your application vendor.

[Φ] Intel® EM64T requires a computer system with a processor, chipset, BIOS, operating system, device drivers and applications enabled for Intel EM64T. Processor will not operate (including 32-bit operation) without an Intel EM64T-enabled BIOS. Performance will vary depending on your hardware and software configurations. See www.intel.com/info/em64t for more information including details on which processors support Intel EM64T or consult with your system vendor for more information.

THIS PAGE INTENTIONALLY LEFT BLANK

For further information visit:

developer.intel.com/technology/itj/index.htm