Intel Virtualization Technology

Once confined to specialized server and mainframe systems, virtualization is now supported in off-the-shelf systems based on Intel architecture hardware. Intel Virtualization Technology provides hardware support for processor virtualization, enabling simplifications of virtual machine monitor software. Resulting VMMs can support a wider range of legacy and future operating systems while maintaining high performance.

Rich Uhlig
Gil Neiger
Dion Rodgers
Amy L. Santoni
Fernando C.M. Martins
Andrew V. Anderson
Steven M. Bennett
Alain Kägi
Felix H. Leung
Larry Smith
Intel Corporation

Virtualizing a computing system's physical resources to achieve improved sharing and utilization has been well established for decades.[1] Full virtualization of all system resources–including processors, memory, and I/O devices–makes it possible to run multiple operating systems on a single physical platform.

In a nonvirtualized system, a single OS controls all hardware platform resources. A virtualized system includes a new layer of software, the *virtual machine monitor*. The VMM's principal role is to arbitrate accesses to the underlying physical host platform's resources so that multiple operating systems (which are guests of the VMM) can share them. The VMM presents to each guest OS a set of virtual platform interfaces that constitute a virtual machine (VM).

Once confined to specialized, proprietary, high-end server and mainframe systems, virtualization is now becoming more broadly available and is supported in off-the-shelf systems based on Intel architecture (IA) hardware. This development is due in part to the steady performance improvements of IA-based systems, which mitigates traditional virtualization performance overheads. Other factors include new creative software approaches that address the difficulties inherent to IA virtualization[2-4] and the emergence of novel applications for virtualization in both industry and academia.

VIRTUALIZATION USAGE MODELS

Classic benefits of virtualization include improved utilization, manageability, and reliability of mainframe systems.[5] Several users with differing OS requirements can more easily share a virtualized server, OS upgrades can be staged across VMs to minimize downtime, and failures in guest software can be isolated to the VMs in which they occur.

While these benefits have traditionally been considered valuable in high-end server systems, recent academic research and emerging new VMM-based products suggest that the benefits of virtualization have wider appeal across a broad range of both server and client systems. Figure 1 illustrates three categories of functional capabilities that encompass a broad range of virtualization usages.

Workload isolation

Virtualization can improve overall system security and reliability by isolating multiple software stacks in their own VMs. Security may be improved because intrusions can be confined to the VM in which they occur, while reliability can be enhanced because software failures in one VM do not affect the other VMs.

Thomas Bressoud and Fred Schneider examined the application of virtualization techniques to achieve system fault tolerance by running identical copies of the same workload in two separate VMs to recover from system failures.[6] The Terra[7] and ReVirt[8] projects are recent academic explorations into the use of virtualization for improved security. Principles of system-software isolation feature prominently in Microsoft's NGSCB (Next-Generation Secure Computing Base)[9] and in VMware's ACE (Assured Computing Environment).

Published by the IEEE Computer Society
0018-9162/05/$20.00 © 2005 IEEE

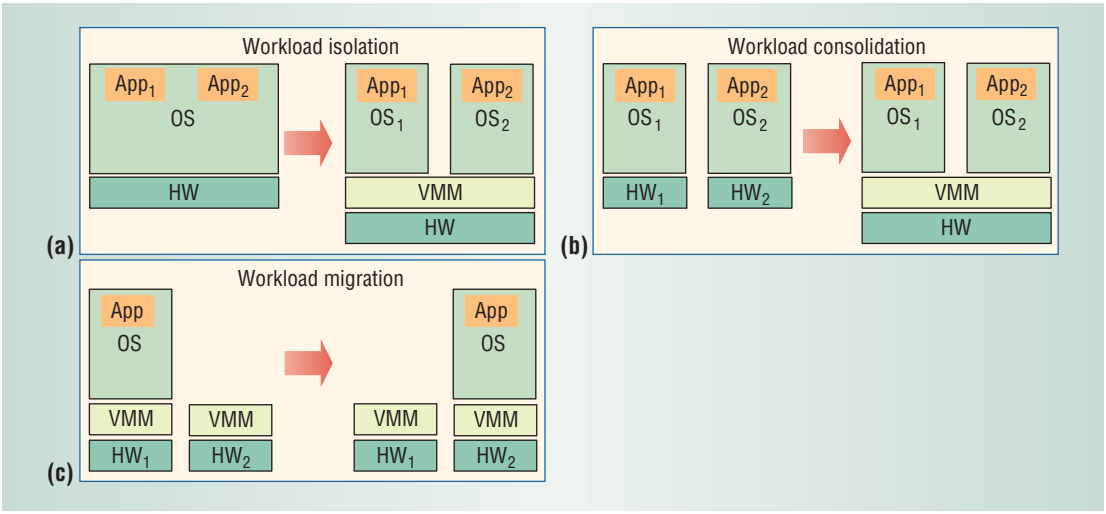

Figure 1. Virtualization capabilities. VMM-based products offer a broad range of virtualization usages in three categories: (a) workload isolation, (b) workload consolidation, and (c) workload migration.

Workload consolidation

Corporate data centers are challenged by the proliferation of large numbers of heterogeneous and underutilized servers that run single-OS and single-application workloads—for example, Web hosting or file serving. Virtualization makes it possible to consolidate individual workloads onto a single physical platform, reducing the total cost of ownership.

Management of upgrades presents another concern for information technology managers. When new hardware or a new OS release becomes available, the challenges of supporting incompatible legacy software often gate entire corporate upgrades. Virtualization mitigates this problem by allowing systems to run legacy and new operating systems concurrently.

Embedding certain system-management functions within a VM can improve client manageability. For example, routing all network traffic through a management VM can provide network "circuit breaker" capabilities that disconnect the client from a corporate intranet if it appears to be infected by a virus.

Workload migration

By encapsulating a guest's state within a VM, virtualization makes it possible to decouple the guest from the hardware on which it is currently running and to migrate it to a different platform.

In addition to facilitating hardware maintenance operations, VM migration can be triggered automatically by workload balancing or failure-prediction agents. This capability delivers improved quality of service at a lower operational cost. Xen[2] and the Internet Suspend-Resume Project[10] have demonstrated workload migration in both servers and clients, and the technology forms the basis of commercial products such as VMotion from VMware.[11]

SOFTWARE-ONLY INTEL ARCHITECTURE VIRTUALIZATION

Established and emerging applications motivate strong support for virtualization in both server and client computing systems. Unfortunately, the IA-32 and Itanium architectures impose many challenges to providing such support. Software techniques exist that address some of those challenges.

Challenges to virtualizing Intel architectures

Intel microprocessors provide protection based on the concept of a 2-bit privilege level, using 0 for most-privileged software and 3 for the least-privileged. The privilege level determines whether privileged instructions, which control basic CPU functionality, can execute without fault; it also controls address-space accessibility based on the configuration of the processor's page tables and, for IA-32, segment registers. Most IA software uses only privilege levels 0 and 3, as Figure 2a illustrates.

For an OS to control the CPU, some of its components must run with privilege level 0. Because a VMM cannot allow a guest OS such control, a guest OS cannot execute at privilege level 0. Thus, IA-based VMMs must use *ring deprivileging*, a technique that runs all guest software at a privilege level greater than 0. A VM could deprivilege a guest OS by running it either at privilege level 1 (the 0/1/3 model) or at privilege level 3 (the 0/3/3 model). Figures 2b and 2c illustrate these choices. Although the 0/1/3 model supports simpler VMMs, it cannot be used on IA-32 processors for guests in 64-bit

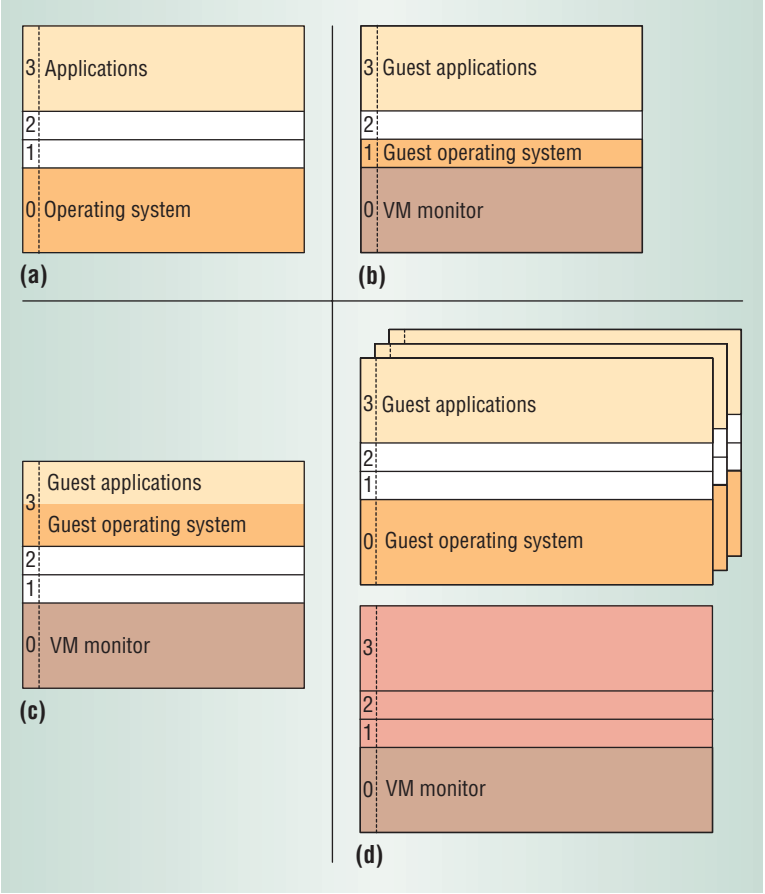

Figure 2. Virtualization methods. Intel microprocessors provide protection based on the concept of a 2-bit privilege level, using 0 for most-privileged software and 3 for least-privileged. The privilege level determines whether privileged instructions, which control basic CPU functionality, can execute without fault. (a) Nonvirtualized system with OS operating at level 0 and all software applications running at level 3; (b) the 0/1/3 model for ring deprivileging; (c) the 0/3/3 model for ring deprivileging; (d) system using VT-x or VT-i instead of deprivileging to constrain guest software.

mode. The 64-bit mode is part of Intel's EM64T (Extended Memory 64 Technology), the 64-bit extension to IA-32.

Ring deprivileging causes numerous virtualization challenges.

Ring aliasing. Ring aliasing refers to problems that arise when software is run at a privilege level other than the level for which it was written. An example in IA-32 is the PUSH instruction—which pushes its operand on the stack—when executed with the CS register (part of which is the current privilege level). Similarly, the Itanium instruction br.call saves the current privilege level into the ppl field in the PFS register, which can be read at any privilege level. In either case, a guest OS could easily determine that it is not running at privilege level 0.

Address-space compression. Operating systems expect to have access to the processor's full virtual-address space, known as the linear-address space in IA-32. A VMM must reserve for itself some portion of the guest's virtual-address space. The VMM could run entirely within the guest's virtual-address space, which allows it easy access to guest data, although the VMM's instructions and data structures might use a substantial amount of the guest's virtual-address space.

Alternatively, the VMM could run in a separate address space, but even in that case the VMM must use a minimal amount of the guest's virtual-address space for the control structures that manage transitions between guest software and the VMM. (For IA-32, these structures include the IDT and the GDT, which reside in the linear-address space. For the Itanium architecture, the structures include the IVT, which resides in the virtual-address space.)

The VMM must prevent guest access to those portions of the guest's virtual-address space that the VMM is using. Otherwise, the VMM's integrity could be compromised if the guest can write to those portions, or the guest could detect that it is running in a virtual machine if it can read them. Guest attempts to access these portions of the address space must generate transitions to the VMM, which can emulate or otherwise support them. The term *address-space compression* refers to the challenges of protecting these portions of the virtual-address space and supporting guest accesses to them.

Nonfaulting access to privileged state. Privilege-based protection prevents unprivileged software from accessing certain components of CPU state. In most cases, attempted accesses result in faults, allowing a VMM to emulate the desired guest instruction. However, the IA-32 and Itanium architectures both include instructions that access privileged state and do not fault when executed with insufficient privilege. For example, the IA-32 registers GDTR, IDTR, LDTR, and TR contain pointers to data structures that control CPU operation. Software can execute the instructions that write to, or *load*, these registers (LGDT, LIDT, LLDT, and LTR) only at privilege level 0. However, software can execute the instructions that read, or *store*, from these registers (SGDT, SIDT, SLDT, and STR) at any privilege level. If the VMM maintains these registers with unexpected values, a guest OS using the latter instructions could determine that it does not have full control of the CPU.

Another example pertains to the Itanium page-table address (PTA) register, which contains the base address of the virtual hash page table (VHPT). The instruction mov PTA is the normal way to access this register, and software can execute it only at privilege level 0. However, the thash instruction indirectly exposes all or part of the VHPT base address, and software can execute it at any privi-

lege level. If the VMM maintains the VHPT at a different address than the guest OS expects, a guest OS using the thash instruction could determine that it does not have full control of the CPU.

Adverse impacts on guest transitions. Ring deprivileging can interfere with the effectiveness of facilities in the IA-32 and Itanium architectures that accelerate the delivery and handling of transitions to OS software. The IA-32 SYSENTER and SYSEXIT instructions support low-latency system calls. SYSENTER always effects a transition to privilege level 0, and SYSEXIT will fault if executed outside that privilege level. Ring deprivileging thus has the following implications:

- Executions of SYSENTER by a guest application will cause a transition to the VMM and not to the guest OS. The VMM must thus emulate every guest execution of SYSENTER.
- Execution of SYSEXIT by a guest OS will cause a fault to the VMM. Thus, the VMM must emulate every guest execution of SYSEXIT.

The Itanium architecture supports efficient interruption handlers by providing them with information about an interruption and the interrupted context. This data is recorded, not in memory, but in a set of interruption-control registers. The processor protects system integrity by generating faults in response to accesses to those registers outside privilege level 0. Typically, every interruption handler reads these registers. If each such access generates a fault to the VMM, the performance of these handlers will be severely compromised.

Interrupt virtualization. Providing support for external interrupts, especially regarding interrupt masking, presents some specific challenges to VMM design. Both the IA-32 and Itanium architectures provide mechanisms for masking external interrupts, preventing their delivery when the OS is not ready for them. IA-32 uses the interrupt flag (IF) in the EFLAGS register to control interrupt masking; the Itanium architecture uses the i bit in the PSR to provide this function.

A VMM will likely manage external interrupts and deny guest software the ability to control interrupt masking. Existing protection mechanisms allow such denial of control by ensuring that guest attempts to control interrupt masking will fault in the context of ring deprivileging. Such faulting can cause problems because some operating systems frequently mask and unmask interrupts. Intercepting every guest attempt to do so could significantly affect system performance.

Intel Architecture Glossary

The IA-32 and Itanium architectures each include specific instructions, registers, and tables, some of which are listed below.

IA-32 terms

CPUID: CPU identification instruction
CR: control registers: CR0, CR3 (page-table base address, which controls translation from linear to physical addresses), CR4, and CR8 (current task priority)
CS: segment register for the current code segment; in some modes. its low 2 bits are the current privilege level
DR: debug register
EFLAGS: 32-bit version of the flags register; contains arithmetic flags as well as the interrupt flag (IF), used to mask interrupts
GDT: global descriptor table; contains descriptors that can be loaded into segment registers LDTR and TR
GDTR, IDTR, LDTR, TR: registers that reference the GDT, IDT, LDT, and TSS
HLT: halt instruction
IDT: interrupt descriptor table; controls the delivery of exceptions and interrupts to their software handlers
IF: bit in the EFLAGS register that controls interrupt masking
INVLPG: invalidate TLB entry instruction
LDT: local descriptor table; contains descriptors that can be loaded into segment registers
LGDT, LIDT, LLDT, LTR: instructions that write to GDTR, IDTR, and TR
MOV: move instruction; different versions allow read and write access to the control registers and debug registers
MWAIT: monitor wait instruction
PUSH: push instruction; pushes its operand on the stack
RDMSR, WRMSR: instructions to read from and write to model-specific registers
RDPMC: read performance-monitoring counters instruction
RDTSC: read time-stamp counter instruction
segment registers: registers that control translation from logical to linear addresses
SGDT, SIDT, SLDT, STR: instructions that read from GDTR, IDTR, and TR
SYSENTER, SYSEXIT: fast system call and fast return from fast system call instructions
TSS: task-state segment; among other things, the current TSS controls the ability of software to access I/O ports

Itanium terms

br.call: branch instruction used to effect a conditional procedure call
i: bit in the PSR that controls interrupt masking
IVT: interrupt vector table; controls delivery of exceptions and interrupts to their software handlers
mov: move instruction; different versions allow read and write access to the control registers (including PTA)
PFS: previous function state register
ppl: previous privilege level field in the PFS register
PAL: processor abstraction layer; provides a consistent firmware interface to processor implementation-specific features
PSR: processor status register
PTA: page table address register
rfi: return from interruption instruction
thash: translation hashed entry address instruction
VHPT: virtual hash page table; controls translation from virtual to physical addresses

A central design goal for Intel Virtualization Technology is to eliminate the need for CPU paravirtualization and binary translation techniques.

Even if it were possible to prevent guest modifications of interrupt masking without intercepting each attempt, challenges would remain when a VMM has a "virtual interrupt" to deliver to a guest. A virtual interrupt should be delivered only when the guest has unmasked interrupts. To deliver virtual interrupts in a timely way, a VMM should intercept some, but not all, attempts by a guest to modify interrupt masking. Doing so could significantly complicate the design of a VMM.

Ring compression. Ring deprivileging uses privilege-based mechanisms to protect the VMM from guest software. IA-32 includes two such mechanisms: segment limits and paging. Because segment limits do not apply in 64-bit mode, paging must be used in this mode. Because IA-32 paging does not distinguish privilege levels 0-2, the guest OS must run at privilege level 3. Thus, the guest OS will run at the same privilege level as guest applications and will not be protected from them. This problem is called ring compression.

Access to hidden state. Some components of IA-32 CPU state are not represented in any software-accessible register. Examples include the hidden descriptor caches for the segment registers. A segment-register load copies a referenced descriptor (from the GDT or LDT) into this cache, which is not modified if software later writes to the descriptor tables. IA-32 does not provide mechanisms for saving and restoring these hidden components of a guest context when changing VMs or for preserving them while the VMM is running.

Addressing virtualization challenges in software

To address the virtualization challenges that the IA-32 and Itanium architecture present, VMM designers have developed creative solutions that modify guest software (source or binary). Denali[12] and Xen[2] are examples of VMMs that use source-level modifications in a technique called *paravirtualization*. Developers of these VMMs modify a guest-OS kernel and its device drivers to create an interface that is easier to virtualize.

Paravirtualization offers high performance and does not require making changes to guest applications. A disadvantage of paravirtualization is that it limits the range of supported operating systems. For example, Xen cannot currently support an operating system that its developers have not modified, such as Microsoft Windows.

A VMM can support legacy operating systems by making modifications directly to guest-OS binaries. VMMs that use such binary translation techniques include those developed by VMware[4] as well as Virtual PC and Virtual Server from Microsoft.[3] Such VMMs support a broader range of operating systems, albeit with higher performance overheads, than VMMs that use paravirtualization.[2]

A central design goal for Intel Virtualization Technology is to eliminate the need for CPU paravirtualization and binary translation techniques and thereby enable the implementation of VMMs that can support a broad range of unmodified guest operating systems while maintaining high levels of performance.

INTEL VIRTUALIZATION TECHNOLOGY

Intel Virtualization Technology includes VT-x support for IA-32 processor virtualization and VT-i support for the Itanium architecture.[13,14]

VT-x architecture overview

VT-x augments IA-32 with two new forms of CPU operation: VMX root operation and VMX non-root operation. A VMM runs in VMX root operation; it runs its guests in VMX non-root operation. Both forms of operation support all four privilege levels, allowing a guest OS to run at its intended privilege level and providing a VMM with the flexibility to use multiple privilege levels. VMX root operation is similar to IA-32 without VT-x. Software running in VMX non-root operation is deprivileged in certain ways, regardless of privilege level.

VT-x defines two new transitions: a transition from VMX root operation to VMX non-root operation—that is, from VMM to guest—called a *VM entry*, and a transition from VMX non-root operation to VMX root operation—that is, from guest to VMM—called a *VM exit*.

The virtual-machine control structure (VMCS) is a new data structure that manages VM entries and VM exits and processor behavior in VMX non-root operations. 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.

Processor behavior changes substantially in VMX non-root operation. Most importantly, many instructions and events cause VM exits. Some instructions cannot be executed in VMX non-root operation because they cause VM exits uncondi-

tionally; these include CPUID, MOV from CR3, RDMSR, and WRMSR. Other instructions, interrupts, and exceptions can be configured to cause VM exits conditionally, using VM-execution control fields in the VMCS.

VM-execution control fields. The VM-execution control fields allow a VMM the flexibility to specify the instructions and events that cause VM exits. There are separate controls for each of the following instructions: HLT, INVLPG, MOV CR8, MOV DR, MWAIT, RDPMC, and RDTSC. These controls support a variety of virtualization strategies. Additional controls allow selective protection of CR0, CR3, and CR4.

VT-x includes two controls that support interrupt virtualization. When the external interrupt exiting control is set, all external interrupts cause VM exits; in addition, the guest cannot mask interrupts. When the interrupt-window exiting control is set, a VM exit occurs whenever guest software indicates that it is ready to receive interrupts.

To support VMM flexibility, VT-x includes bitmaps that allow a VMM selectivity regarding some causes of VM exits. One of these is the exception bitmap, which contains 32 entries for the IA-32 exceptions. It allows a VMM to specify which exceptions should cause VM exits and which should not. Another bitmap allows per-port control of I/O instructions.

VMCS details. The guest-state area contains the state of the virtual CPU associated with the VMCS. It includes fields corresponding to the IA-32 registers that manage processor operation, such as the segment registers, CR3, and IDTR.

In addition, the guest-state area includes fields corresponding to certain components of nonregister processor state—for example, the descriptor caches for the segment registers. Inclusion of these components allows the VMM to record their values when a VM is not running and to restore them when the VM is restarted.

A VMM references the VMCS with a physical—not linear—address. This eliminates the need to locate the VMCS in the guest's linear-address space, which can be different from the VMM's linear-address space.

VM entries and exits. VM entries load processor state from the guest-state area of the VMCS. A VMM can optionally configure VM entry to follow this loading by injecting an interrupt or exception. The CPU effects this injection using the guest IDT, just as if the injected event had occurred immediately after VM entry. This feature removes the need for a VMM to emulate delivery of these events.

VM exits save processor state into the guest-state area and load processor state from the host-state area. All VM exits use a common entry point to the VMM. To simplify the design of a VMM, every VM exit saves into the VMCS detailed information specifying the reason for the exit; many exits also record an exit qualification, which provides further details.

VT-x includes VM execution control fields that support a variety of virtualization strategies for IA-32.

For example, if the MOV CR instruction causes a VM exit, the exit reason would indicate "control-register access"; the exit qualification would indicate (1) the identity of the control register (for example, CR0); (2) whether the MOV was to or from the control register; and (3) which general-purpose register was the source or destination of the instruction.

Both VM entries and VM exits load CR3 (the base address of the page-table hierarchy). This implies that the VMM and the guest can run in different linear-address spaces.

VT-i architecture overview

VT-i consists of extensions to the Itanium processor hardware and the processor abstraction layer (PAL) firmware.

Processor status bit PSR.vm. The principal hardware extension is the addition of a new bit (vm) in the processor-status register (PSR). A VMM runs with PSR.vm = 0; it runs its guests with PSR.vm = 1. All four privilege levels can be used regardless of the value of PSR.vm; guest software can run at its intended privilege level, and a VMM has the flexibility to use multiple privilege levels. When PSR.vm = 0, processor operation is similar to operation without VT-i. When PSR.vm = 1, all privileged instructions and some nonprivileged instructions—for example, thash—cause a new virtualization fault.

PSR.vm is cleared to 0 on all interruptions delivered through the IVT; thus, the VMM or PAL handles all interruptions, even those belonging to guest software. The VMM or PAL can set PSR.vm to 1 by using the rfi instruction to return to guest software. VT-i adds a new instruction, vmsw (virtual machine switch), which modifies the PSR.vm bit with minimum overhead, reducing the latency of transitions between guest software and a VMM in cooperative virtualization environments.

PSR.vm also controls the number of virtual-address bits available to software. When a VMM is running—that is, PSR.vm = 0—all implemented virtual address bits are available. When a guest is running—that is, PSR.vm = 1—the uppermost implemented virtual-address bit is not available,

VT-i includes PAL firmware layer additions that provide a consistent programming interface to a VMM even if the hardware is not implemented identically across processor generations.

and an exception occurs if this bit is used. This reserves some dedicated address space for the VMM that guest software cannot access.

IVT vectors. To facilitate efficient handling of transitions to a VMM, VT-i adds two new vectors to the IVT. The VMM uses the virtualization vector to configure the processor to use two of the processor-banked registers to identify the cause of the virtualization fault and the faulting opcode. With the virtual external interrupt vector, the VMM can use a PAL service to register pending virtual interrupts. If the VMM has registered an interrupt and the guest performs an operation that would unmask it, control is transferred to the virtual external interrupt vector.

PAL firmware layer extensions. VT-i includes additions to the PAL firmware layer that provide a consistent programming interface to a VMM even if the hardware is not implemented identically across processor generations. The PAL extensions include a set of new procedures, the addition of PAL services for high-frequency VMM operations, and a virtual processor descriptor (VPD) table.

VT-i defines PAL procedures for setting up and tearing down a virtual machine environment, initializing and terminating virtual processors, and saving and restoring a virtual processor's state. These procedures follow the same calling convention as other PAL procedures.

VT-i introduces a PAL interface for virtualization called a service. To reduce overhead, PAL services use a new calling convention specifically targeted for VMMs. PAL services provide several functions including synchronizing guest shadow registers and the VPD, saving and restoring a subset of a virtual processor's state, and resuming execution of guest software after a transition to the VMM.

Both the PAL firmware and the VMM can access the VPD, which is located in memory. The VPD contains configuration settings for the virtual processor and the subset of the virtual processor's state that influences its execution characteristics. For example, the virtual processor's control-register values are located in the VPD. The VPD contains two configuration fields that allow the VMM to customize the virtualization environment:

- The virtualization-acceleration field optimizes virtualization for particular resources and instructions. For example, it allows optimizations that reduce the number of transitions a VMM must handle when the guest is accessing the interruption-control registers in a fault handler.
- The virtualization-disable field disables virtualization of a particular resource or instruction. For example, the VMM could allow a guest direct access to control registers dedicated to external interrupts.

SOLVING VIRTUALIZATION CHALLENGES WITH VT-X AND VT-I

VT-x and VT-i solve virtualization challenges in part by allowing guest software to run at its intended privilege level. Guest software is constrained, not by privilege level, but because—for VT-x—it runs in VMX non-root operation or—for VT-i—with PSR.vm = 1. Figure 2d illustrates this usage.

Address-space compression

VT-x and VT-i provide two different techniques for solving address-space compression problems. With VT-x, every transition between guest software and the VMM can change the linear-address space, allowing the guest software full use of its own address space. The VMX transitions are managed by the VMCS, which resides in the physical-address space, not the linear-address space.

With VT-i, the VMM has a virtual-address bit that guest software cannot use. A VMM can conceal hardware support for this bit by intercepting guest calls to the PAL procedure that reports the number of implemented virtual-address bits. As a result, the guest will not expect to use this uppermost bit, allowing the VMM exclusive use of half of the virtual-address space.

Ring aliasing and ring compression

VT-x and VT-i eliminate ring-aliasing problems because they allow a VMM to run guest software at its intended privilege level. Instructions such as PUSH (of CS) and br.call cannot reveal that software is running in a virtual machine. VT-x also eliminates ring compression problems that arise when a guest OS executes at the same privilege level as guest applications.

Nonfaulting access to privileged state

VT-x and VT-i avoid problems of nonfaulting accesses to privileged state in two ways: by adding support that causes such accesses to transition to a VMM and by adding support that causes the state accessed to become unimportant to a VMM.

A VMM based on VT-x does not require control of the guest privilege level, and the VMCS con-

trols the disposition of interrupts and exceptions. Thus, it can allow its guest access to the GDT, IDT, LDT, and TSS. VT-x allows guest software running at privilege level 0 to use the instructions LGDT, LIDT, LLDT, LTR, SGDT, SIDT, SLDT, and STR.

With VT-i, the thash instruction causes virtualization faults, giving a VMM the opportunity to conceal any modifications it may have made to the VHPT base address.

Guest transitions

Guest software cannot use the IA-32 instructions SYSENTER and SYSEXIT if the guest OS runs outside privilege level 0. With VT-x, a guest OS can run at privilege level 0, allowing use of these instructions.

With VT-i, a VMM can use the virtualization-acceleration field in the VPD to indicate that guest software can read or write the interruption-control registers without invoking the VMM on each access. The VMM can establish the values of these registers before any virtual interruption is delivered and can revise them before the guest interruption handler returns.

Interrupt virtualization

VT-x and VT-i both provide explicit support for the virtualization of interrupt masking.

VT-x includes an external-interrupt exiting VM-execution control. When this control is set to 1, a VMM prevents guest control of interrupt masking without gaining control on every guest attempt to modify EFLAGS.IF. Similarly, VT-i includes a virtualization-acceleration field that prevents guest software from affecting interrupt masking and avoids making transitions to the VMM on every access to the PSR.i bit.

VT-x also includes an interrupt-window exiting VM-execution control. When this control is set to 1, a VM exit occurs whenever guest software is ready to receive interrupts. A VMM can set this control when it has a virtual interrupt to deliver to a guest. Similarly, VT-i includes a PAL service that a VMM can use to register that it has a virtual interrupt pending. When guest software is ready to receive such an interrupt, the service transfers control to the VMM via the new virtual external interrupt vector.

Access to hidden state

VT-x includes in the guest-state area of the VMCS fields corresponding to CPU state not represented in any software-accessible register. The processor loads values from these VMCS fields on every VM entry and saves into them on every VM exit. This provides the support necessary for preserving this state while the VMM is running or when changing VMs.

While virtualization technology was once confined to proprietary server and mainframe computing systems, established and emerging applications in both server and client systems are moving it into the mainstream.

Despite the promise of new and existing virtualization usages, many challenges stand in the way of achieving efficient virtualization of today's IA-based systems. Creative software techniques such as binary translation and paravirtualization have addressed some of these problems, but the scope of these challenges has meant that these solutions are either highly complex—and potentially less robust—or are incomplete in their ability to run unmodified legacy operating systems.

VT-x and VT-i are the first components of Intel Virtualization Technology, a series of processor and chipset innovations soon to become available in IA-based client and server platforms. VT-x and VT-i offer solutions to the problems inherent in IA-32 and Itanium processor virtualization, enabling simpler, more robust, and more secure VMM software to support a wider range of legacy and future operating systems while maintaining high levels of performance. ■

Acknowledgments

The authors thank the following for their contributions to the development of the VT-x and VT-i architectures: Erik Cota-Robles, Gary Hammond, Stalinselvaraj Jeyasingh, Mike Kozuch, Tariq Masood, Dale Morris, Jonathan Ross, Rajesh Sankaran, Sebastian Schönberg, and Chris Zou.

References

1. R.P. Goldberg, "Survey of Virtual Machine Research," *Computer*, June 1974, pp. 34-45.
2. P. Barham et al., "Xen and the Art of Virtualization," *Proc. 19th ACM Symp. Operating Systems Principles*, ACM Press, 2003, pp. 164-177.
3. Microsoft Corp., "Microsoft Virtual Server 2005 Technical Overview," 2004; http://download.

microsoft.com/download/5/5/3/55321426-cb43-4672-9123-74ca3af6911d/VS2005TechWP.doc.
4. C.A. Waldspurger, "Memory Resource Management in VMware ESX Server," *Proc. 5th Symp. Operating Systems Design and Implementation*, Usenix, 2002, pp. 181-194.
5. R.J. Creasy, "The Origin of the VM/370 Time-Sharing System," *IBM J. Research and Development*, Sept. 1981, pp. 483-490.
6. T.C. Bressoud and F.B. Schneider, "Hypervisor-Based Fault Tolerance," *Proc. 15th ACM Symp. Operating Systems Principles*, ACM Press, 1995, pp. 1-11.
7. G.W. Dunlap et al., "ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay," *Proc. 5th Symp. Operating Systems Design and Implementation*, Usenix, 2002, pp. 211-224.
8. T. Garfinkel et al., "Terra: A Virtual Machine-Based Platform for Trusted Computing," *Proc. 19th ACM Symp. Operating Systems Principles,* ACM Press, 2003, pp. 193-206.
9. P.B. England et al., "A Trusted Open Platform," *Computer*, July 2003, pp. 55-62.
10. M. Kozuch and M. Satyanarayanan, "Internet Suspend/Resume," *Proc. 4th IEEE Workshop Mobile Computing Systems and Applications*, IEEE Press, 2002, p. 40.
11. VMware Inc., "Building Virtual Infrastructure with VMware VirtualCenter," white paper V00014-20001205, 2004; www.vmware.com/pdf/vi_wp.pdf.
12. A. Whitaker, M. Shaw, and S. Gribble, "Scale and Performance in the Denali Isolation Kernel," *Proc. 5th Symp. Operating Systems Design and Implementation*, Usenix, 2002, pp. 195-210.
13. Intel Corp., "Intel Virtualization Technology Specification for the IA-32 Architecture;" www.intel.com/technology/vt/.
14. Intel Corp., "Intel Virtualization Technology Specification for the Intel Itanium Architecture;" www.intel.com/technology/vt/.

Rich Uhlig *is a senior principal engineer in Intel's Corporate Technology Group and leads various aspects of Intel's overall virtualization effort including architecture definition, research prototyping, performance analysis, and software usage. He received a PhD in computer science and engineering from the University of Michigan.*

Gil Neiger *is a principal engineer in Intel's Corporate Technology Group and leads development of the VT-x architecture. He received a PhD in computer science from Cornell University.*

Dion Rodgers *is a senior principal engineer in Intel's Digital Enterprise Group and is responsible for bringing multiple advanced technology initiatives such as VT-x to the IA-32 product line. He received an MS in computer engineering from Clemson University.*

Amy L. Santoni *is a senior staff engineer in Intel's Digital Enterprise Group and is one of the principal architects of VT-i. She received a BS in computer engineering from the University of Michigan.*

Fernando C.M. Martins *is a senior technology strategist in Intel's Digital Enterprise Group and leads strategic planning for Intel Virtualization Technology. He received a PhD in electrical and computer engineering from Carnegie Mellon University. Contact him at fernando.martins@intel.com.*

Andrew V. Anderson *is a staff researcher in Intel's Corporate Technology Group and is working on performance characterization and techniques for efficient virtualization. He received an MS in electrical and computer engineering from Brigham Young University.*

Steven M. Bennett *is a senior staff researcher in Intel's Corporate Technology Group and is responsible for development of technologies to support virtualization. He received an MS in computer science from the University of Wisconsin-Madison.*

Alain Kägi *is a researcher in Intel's Corporate Technology Group and leads the development of research virtual machine monitors. He received a PhD in computer science from the University of Wisconsin-Madison.*

Felix H. Leung *is a staff engineer in Intel's Digital Enterprise Group and has developed hardware and PAL support for VT-i. He received a BS in computer sciences from the University of Wisconsin-Madison.*

Larry Smith *is a staff architect in Intel's Digital Enterprise Group and led the development and implementation of VT-x. He received a BS in industrial management from the Lawrence Institute of Technology.*