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

Intel® Technology Journal

Intel® Virtualization Technology

Intel® Virtualization Technology: Hardware Support for Efficient Processor Virtualization

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

Intel® Virtualization Technology: Hardware Support for Efficient Processor Virtualization

Gil Neiger, Corporate Technology Group, Intel Corporation
Amy Santoni, Digital Enterprise Group, Intel Corporation
Felix Leung, Digital Enterprise Group, Intel Corporation
Dion Rodgers, Digital Enterprise Group, Intel Corporation
Rich Uhlig, Corporate Technology Group, Intel Corporation

Index words: virtualization, processor, VT-x, VT-i

ABSTRACT

Virtualizing the physical resources of a computing system to improve sharing and utilization has been done for decades. Virtualization had once been confined to specialized server and mainframe systems, but improvements in the performance of platforms based on Intel® technology now allow those platforms to efficiently support virtualization. However, the IA-32 and Itanium® processor architectures pose a number of significant challenges to virtualization.

The first generation of Intel® Virtualization Technology$^{\Delta}$ (VT) for IA-32 and Itanium processors provides hardware support that simplifies processor virtualization, enabling reductions in virtual machine monitor (VMM) software size and complexity. Resulting VMMs can support a wider range of legacy and future operating systems (OSs) on the same physical platform while maintaining high performance.

In this paper, we provide details of the virtualization challenges posed by IA-32 and Itanium processors; present an overview and furnish details of VT-x (Intel Virtualization Technology for the IA-32 architecture) and VT-i (Intel Virtualization Technology for the Itanium architecture); show how VT-x and VT-i address virtualization challenges; and finally provide examples of usage of the VT-x and VT-i architecture.

INTRODUCTION

Virtualizing the physical resources of a computing system to achieve improved degrees of sharing and utilization is a well-established concept that goes back decades [1]. Full virtualization of all system resources (including processors, memory and I/O devices) makes it possible to run multiple operating systems (OSs) on a single physical platform. In contrast to a non-virtualized system, in which a single OS is solely in control of all hardware platform resources, a virtualized system includes a new layer of software, called a virtual-machine monitor (VMM). The principal role of the VMM is to arbitrate access to the underlying physical host platform resources so that these resources can be shared among multiple OSs that are "guests" of the VMM. The VMM presents to each guest OS a set of virtual platform interfaces that constitute a virtual machine (VM).

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

In the sections that follow, we examine some of the technical difficulties with bringing virtualization to IA-based systems and present an overview of Intel Virtualization Technology (VT), which provides hardware assists for overcoming these difficulties. The first generation of VT focuses on a set of hardware assists that facilitates the virtualization of IA processors. VT-x refers to new architectural extensions that aid in IA-32 processor virtualization, while VT-i refers to a set of assists for virtualizing Itanium processors. VT-x and VT-i eliminate many of the problems that make writing a VMM for IA-based systems a challenge and hence make possible the

broader availability of virtualization technology in both server and client systems.

SOFTWARE-ONLY VIRTUALIZATION WITH THE IA-32 AND ITANIUM® ARCHITECTURES

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

Challenges to Virtualizing Intel Architectures

Intel microprocessors (both IA-32 and Itanium architecture) 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. 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.

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, VMMs running on either IA-32 or Itanium processors must use *ring deprivileging*, a technique that runs all guest software at a privilege level greater than 0. A guest OS could be deprivileged in two distinct ways: it could run either at privilege level 1 (the 0/1/3 model) or at privilege level 3 (the 0/3/3 model).

Although the 0/1/3 model supports simpler VMMs, it cannot be used for guests on IA-32 processors in 64-bit mode (more details in "ring compression" section). (64-bit mode is part of Intel® Extended Memory 64 Technology$^{\Phi}$—Intel® EM64T—the 64-bit extensions to IA-32.)

Ring Aliasing

Ring aliasing refers to problems that arise when software is run at a privilege level other than the privilege level for which it was written.

An example in IA-32 involves the CS segment register, which points to the code segment. If the *PUSH* instruction is executed with the CS segment register, the contents of that register (which include the current privilege level) is pushed on the stack. Similarly, the Itanium instruction *br.call* saves the current privilege level into the *ppl* field of the Previous Function State (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

OSs 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. It could run entirely within the guest's virtual-address space, which allows it easy access to guest data, but the VMM's instructions and data structures would use a substantial amount of the guest's virtual-address space.

Alternatively, the VMM can 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 interrupt-descriptor table (IDT) and the global-descriptor table (GDT), which reside in the linear-address space. For the Itanium architecture, the structures include the interruption vector table (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 VM (if it can read those portions). 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.

Non-Faulting 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 page-table address (PTA) register of the Itanium architecture, a field that references

the base address of the virtual hash page table (VHPT). The instruction *mov to cr.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 privilege 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 Impact on Guest System Calls

Ring deprivileging can interfere with the effectiveness of facilities in the IA-32 architecture 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* faults if executed outside that ring. Ring deprivileging thus has the following implications:

- Executions of *SYSENTER* by a guest application cause transitions to the VMM and not to the guest OS. The VMM must emulate every guest execution of *SYSENTER*.
- Executions of *SYSEXIT* by a guest OS cause faults to the VMM. The VMM must emulate every guest execution of *SYSEXIT*.

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 thus 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 processor status register (PSR) to provide this function. In both cases, a value of 0 indicates that interrupts are masked.

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 fault in the context of ring deprivileging. Such faulting can cause problems because some OSs frequently mask and unmask interrupts. Intercepting every guest attempt to do so could significantly affect system performance.

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.

Access to Hidden State

Some components of IA-32 and Itanium processor state are not represented in any software-accessible register. Examples for IA-32 include the hidden descriptor caches for the segment registers. A segment-register load copies the 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 a mechanism for saving and restoring hidden components of a guest context when changing VMs or for preserving them while the VMM is running.

In the Itanium architecture, there is a field in the Register Stack Engine (RSE) called the current frame load enable (CFLE). There is no direct way to write this value. There are cases where the VMM may take an external interrupt and wants to return to the guest OS with this value equal to zero. The return from interrupt (*rfi*) instruction forces this value to a one.

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 (the 0/3/3 model). Thus, the guest OS runs at the same privilege level as guest applications and is not protected from them. This problem is called *ring compression*.

Frequent Access to Privileged Resources

A VMM may prevent guest access to privileged resources by forcing attempts at such accesses to fault. Even when this ensures correct behavior, performance may be compromised if the frequency of such faults is excessive.

In the IA-32 and Itanium architectures, an example involves the task-priority register (TPR). For the IA-32 architecture, this register is located in the advanced programmable interrupt controller (APIC), and for the Itanium architecture, it is one of the control registers. Because it controls interrupt prioritization, a VMM must not allow a guest OS access to the TPR. However, some OSs perform such accesses with very high frequency. These accesses require VMM intervention only if they cause the TPR to drop below a value determined by the VMM.

The Itanium architecture supports efficient interruption handlers by providing them with information about the interruption and the interrupted context. These data are

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.

ADDRESSING VIRTUALIZATION CHALLENGES IN SOFTWARE

To address the virtualization challenges that the IA-32 and Itanium architecture present, VMM designers have developed creative techniques for modifying guest software (source or binary). Denali [5] and Xen* [2] are examples of VMMs that use source-level modifications in a technique called *paravirtualization*. Developers of these VMMs modify the source code of a guest OS to create an interface that is easier to virtualize. Paravirtualization offers high performance and does not require changes to guest applications. A disadvantage of paravirtualization is that it limits the range of supported OSs; VMMs that rely on paravirtualization cannot support an OS whose source code the VMM's developers have not modified.

A VMM can support unmodified OSs by transforming guest-OS binaries on-the-fly to handle virtualization-sensitive operations. 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 OSs than VMMs that use paravirtualization.

A central design goal for Intel VT has been to eliminate the need for CPU paravirtualization and binary translation techniques, to simplify the implementation of robust VMMs that can support a broad range of unmodified guest OSs, and to maintain high levels of performance.

INTEL® VIRTUALIZATION ARCHITECTURE OVERVIEW

In this section, we discuss some of the details of Intel VT architecture. We first describe the VT-x support for IA-32 processor virtualization [6], and then we describe the VT-i support for Itanium processor virtualization [7].

VT-x Architecture Overview

VT-x augments IA-32 with two new forms of CPU operation: *VMX root operation* and *VMX non-root operation*. VMX root operation is intended for use by a VMM, and its behavior is very similar to that of IA-32 without VT-x. VMX non-root operation provides an alternative IA-32 environment controlled by a VMM and designed to support a VM. Both forms of operation support all four privilege levels, allowing guest software to run at its intended privilege level, and providing a VMM with the flexibility to use multiple privilege levels.

VT-x defines two new transitions: a transition from VMX root operation to VMX non-root operation is called a *VM entry*, and a transition from VMX non-root operation to VMX root operation is called a *VM exit*. VM entries and VM exits are managed by a new data structure called the virtual-machine control structure (*VMCS*). The VMCS includes a *guest-state area* and a *host-state area*, each of which contains 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 operation is changed substantially in VMX non-root operation. The most important change is that many instructions and events cause VM exits. Some instructions (e.g., INVD) cause VM exits unconditionally and thus can never be executed in VMX non-root operation. Other instructions (e.g., INVLPG) and all events can be configured to do so conditionally using *VM-execution control fields* in the VMCS.

Guest-State Area

The guest-state area of the VMCS is used to contain elements of the state of virtual CPU associated with that VMCS.

For proper VMM operation, certain registers must be loaded by every VM exit. These include those IA-32 registers that manage operation of the processor, such as the segment registers (to map from logical to linear addresses), CR3 (to map from linear to physical addresses), IDTR (for event delivery), and many others. The guest-state area contains fields for these registers so that their values can be saved as part of each VM exit.

In addition, the guest-state area contains fields corresponding to elements of processor state that are not held in any software-accessible register. One of these elements is the processor's *interruptibility state*, which indicates whether external interrupts are temporarily masked (e.g., due to execution of the MOV-SS instruction) and whether non-maskable interrupts (NMIs) are masked because software is handling an earlier NMI.

The guest-state area does not contain fields corresponding to registers that can be saved and loaded by the VMM itself (e.g., the general-purpose registers). Exclusion of such registers improves the performance of VM entries and VM exits. Software can manage these additional registers more efficiently as it knows better than the CPU when they need to be saved and loaded.

VM-Execution Control Fields

The VMCS contains a number of fields that control VMX non-root operation by specifying the instructions and events that cause VM exits. In this section, we present some of these controls.

The VMCS includes controls that support interrupt virtualization:

- *External-interrupt exiting*. When this control is set, all external interrupts cause VM exits; in addition, the guest is not able to mask these interrupts (e.g., interrupts are not masked if EFLAGS.IF=0).
- *Interrupt-window exiting*. When this control is set, a VM exit occurs whenever guest software is ready to receive interrupts (e.g., when EFLAGS.IF=1).
- *Use TPR shadow*. When this control is set, accesses to the APIC's TPR through control register CR8 (available only in 64-bit mode) are handled in a special way: executions of MOV CR8 access a *TPR shadow* referenced by a pointer in the VMCS. The VMCS also includes a *TPR threshold*; a VM exit occurs after any instruction that reduces the TPR shadow below the TPR threshold.

There are also VM-execution control fields that support efficient virtualization of the IA-32 control registers CR0 and CR4. These registers each comprise a set of bits controlling processor operation. A VMM may wish to retain control of some of these bits (e.g., those that manage paging) but not others (e.g., those that control floating-point instructions). The VMCS includes, for each of these registers, a *guest/host mask* that a VMM can use to indicate which bits it wants to protect. Guest writes can freely modify the unmasked bits, but an attempt to modify a masked bit causes a VM exit. The VMCS also includes, for each of these registers, a *read shadow* whose value is returned to guest reads of the register.

To support VMM flexibility, the VMCS includes bitmaps that allow a VMM selectivity regarding the causes of some VM exits. The following items detail three of these:

- *Exception bitmap*: This field contains 32 entries for the IA-32 exceptions. It allows a VMM to specify which exceptions should cause VM exits and which should not. For page faults, further selectivity is supported based on a fault's error code.
- *I/O bitmaps*: These bitmaps contain one entry for each port in the 16-bit I/O space. An I/O instruction (e.g., IN) causes a VM exit if it attempts to access a port whose entry is set in the I/O bitmaps.
- *MSR bitmaps*: These bitmaps contain two entries (one for read, one for write) for each model-specific register (*MSR*) currently in use. An execution of RDMSR (or WRMSR) causes a VM exit if it attempts to read (or write) an MSR whose read bit (or write bit) is set in the MSR bitmaps.

In addition to the controls mentioned above, there are VM-execution controls that support flexible VM exiting for a number of privileged instructions.

VMCS Details

Like the IA-32 page tables, each VMCS is referenced with a physical (not linear) address. This eliminates the need to locate the VMCS in the guest's linear-address space (which, as noted below, may be different from that of the VMM). The format and layout of the VMCS in memory is not architecturally defined, allowing implementation-specific optimizations to improve performance in VMX non-root operation and to reduce the latency of VM entries and VM exits. VT-x defines a set of new instructions that allows software to access the VMCS in an implementation-independent manner.

Details of VM Entries and VM Exits

As noted earlier, VM entries load processor state from the guest-state area of the VMCS. (Note that, because the state loaded includes CR3, the guest may run in a different linear-address space than the VMM.) In addition to loading guest state, VM entry can be optionally configured for *event injection*. The CPU effects this injection using the guest IDT to deliver an event (exception or interrupt) specified by the VMM, just as if it had actually occurred immediately after VM entry. This feature removes the need for a VMM to emulate delivery of these events.

As noted above, VM exits save processor state into the guest-state area and then load processor state from the host-state area. (Again, because the state loaded includes CR3, the VMM may run in a different linear-address space than the guest.) This implies that all VM exits use a common entry point in the VMM. To simplify the design of a VMM, VT-x specifies that each VM exit save into the VMCS detailed information on the cause of the VM exit. Every VM exit records an exit reason (specifying, for example, which instruction caused the VM exit); many also record an exit qualification, which provides further details. For example, if a VM exit is caused by the MOV CR instruction, the exit reason would indicate "control-register access" and the exit qualification would identify the following: (1) the specific control register (e.g., CR0); (2) whether the MOV was to or from the register; and (3) which other register was the source or destination of the instruction.

Each VM exit due to an IA-32 exception saves, in addition to information about the exception, information about any event (e.g., an external interrupt) that was being

delivered at the time the exception occurred. This allows a VMM to virtualize nested exceptions properly.

VT-i Architecture Overview

VT-i expands the Itanium architecture with extensions to the processor hardware and the Processor Abstraction Layer (*PAL*) firmware.

VT-i adds a new PSR bit (PSR.vm) that allows guest OSs to be run at the privilege level for which they were designed and creates *interceptions* to a VMM necessary for the creation of a complete VM. The VMM runs with this bit equal to zero and runs guest software with this bit equal to one.

The PSR.vm bit modifies the behavior of all privileged instructions as well as that of some non-privileged instructions that access state that a VMM may want to control (including the *thash*, *ttag*, and *mov cpuid* instructions). When a guest OS executes one of these instructions a virtualization intercept is caused which transfers control to the VMM with the PSR.vm bit set to zero.

PSR.vm is orthogonal to the privilege level. This fact allows guest software to run at its designated privilege level; if desired, a VMM can span multiple privilege levels.

PSR.vm also controls the number of virtual-address bits available to software. When a VMM is running (PSR.vm = 0), all implemented virtual-address bits are available. When a guest is running (PSR.vm = 1) the uppermost implemented virtual-address bit is not available and unimplemented data/instruction address faults or unimplemented instruction address traps are created if this bit is used. This provides a VMM a dedicated address space that guest software cannot access.

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

The PAL procedures are used for setting up and tearing down a VM environment; for setting global VMM configuration options; for initializing and terminating virtual processors; and for saving and restoring a subset of state of a virtual processor. These procedures follow the same calling convention as existing PAL procedures. In addition, a new PAL interface called a PAL service has been introduced for virtualization. PAL services reduce overhead through use of a new calling convention specifically targeted for use by a VMM. *PAL services* provide functionality to synchronize guest hardware registers and the VPD; to save and restore a subset of the state of a virtual processor; to resume execution of the guest software after a virtualization intercept; to calculate guest VHPT hashes and tags; and to set up pending interrupts for the guest.

The VPD table is located in memory selected by the VMM. It is usually located in the VMM's virtual-address space and is accessed by both the PAL firmware and the VMM. The VPD contains configuration settings for the virtual processor and a 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 but not its general registers. The layout of the VPD is architected to be 64K in size and includes reserved space for future usage.

The VPD contains two configuration fields that allow the VMM to customize the virtualization environment:

- *Virtualization-acceleration field*. This field allows the VMM to customize the virtualization of a particular resource or instruction, leading to a reduction in the number of virtualization intercepts that the VMM has to handle. It provides accelerations for external-interrupt handling as well as intercept control for reads and writes to interruption control registers (cr16-cr25), reads of the PSR, reads of CPUID, the *cover* instruction, and the bank-switch instruction (*bsw*).

  For example, a VMM could enable the bank-switch optimization. Guest execution of bsw would use values that the VMM had set up in the VPD for the guest OS and would never cause a virtualization intercept to the VMM.

- *Virtualization-disable field*. This field allows the VMM to disable virtualization of a particular resource or instruction, leading to a reduction in the number of virtualization intercepts the VMM handles. This field provides disables for virtualization of the external interrupt control registers (cr65–71), the performance monitoring registers, the debug registers, the PSR.i bit, and the interval timer match register.

To provide efficient handling of virtualization intercepts for a VMM, the architecture has added two new vectors into the IVT:

- *Virtualization vector*. This vector is used for all virtualization-related intercepts. To reduce decoding complexity, a VMM can configure the processor to provide the cause of the virtualization intercept (a bitmap field of intercepting instructions) as well as the faulting opcode in two of the processor banked registers. A VMM can relocate this handler to a

memory location outside the IVT as well through a PAL interface.

- *Virtual external interrupt vector*. The processor uses this vector when the guest unmasks a pending external interrupt. It would be used when the VMM has a virtual interrupt for the guest that it cannot deliver due to guest masking. When the guest performs an operation to unmask the highest pending interrupt, the guest state is updated and control is transferred to this new vector. This streamlines delivery of guest external interrupts for the VMM.

VT-i also provides global configuration options that a VMM can set that apply to all virtual processors activated by the VMM. These global configuration options determine whether the cause of a virtualization intercept is provided, if the opcode of the instruction causing the virtualization intercept is provided, if the performance counters are frozen for all virtualization intercepts, and the byte order (or endianness) of the date located in the VPD.

VT-i also includes the *vmsw* instruction. This instruction transitions the PSR.vm bit with minimum overhead. This can reduce transition overhead between guest software and a VMM in cooperative virtualization environments.

SOLVING VIRTUALIZATION CHALLENGES WITH VT-X AND VT-I

VT-x and VT-i allow 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. These facts allow VMMs to avoid the virtualization challenges identified earlier.

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 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, and hardware will not allow it to do so, thus providing the VMM exclusive use of half of the virtual-address space.

Ring Aliasing and Ring Compression

VT-x and VT-i allow a VMM to run guest software at its intended privilege level. This fact eliminates ring aliasing problems because instructions such as *PUSH* (of CS) and *br.call* cannot reveal that software is running in a VM. It 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 the problem of providing nonfaulting access 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 to become unimportant to a VMM.

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

Problems occur with the IA-32 instructions SYSENTER and SYSEXIT when a guest OS runs outside privilege level 0. With VT-x, a guest OS can run at privilege level 0, which eliminates problems associated with guest transitions.

Interrupt Virtualization

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

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 of 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 the

vector of the pending virtual interrupt. When guest software executes instructions to unmask the pending interrupt, control is transferred to the VMM via the new virtual external interrupt vector.

Access to Hidden State

VT-x and VT-i use different techniques to allow a VMM to manipulate components of guest state that are not represented in any software-accessible register.

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.

VT-i provides a way for the VMM to set the RSE CFLE bit to the desired value via an argument value in the PAL service used to return to guest interruption handlers.

Frequent Access to Privileged Resources

VT-x and VT-i allow a VMM to avoid the overhead of high-frequency guest accesses to the TPR register. A VMM can configure the VMCS (for VT-x) or use an acceleration (for VT-i) so that the VMM is invoked only when required: For VT-x this occurs when the value of the TPR shadow associated with the VMCS drops below that of a TPR threshold in the VMCS. For VT-i this occurs only when the writing of the TPR unmasks a virtual pending external interrupt for the guest.

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.

USAGE OF THE INTEL VIRTUALIZATION ARCHITECTURE

We have described the basic architecture for VT-x and VT-i, and in the next section, we provide some usage examples of the architecture by a VMM. This is intended to highlight some usage models, but it is not a comprehensive set of all usage models.

VMM Usage of VT-x Architecture Features

Exception Handling

VT-x allows a VMM to configure any IA-32 exception to cause a VM exit based on its vector (for page faults, further selectivity is supported based on a fault's error code). When handling such VM exits, a VMM has access to complete information about the exception, including its error code and any other fault-specific information (e.g., the faulting linear address for a page fault).

The VMM may determine that the exception causing the VM exit should be handled by the guest OS. In these cases, the VMM can perform a VM entry to guest using event injection to deliver the exception.

Alternatively, a VMM may respond to such a VM exit by eliminating the cause of the exception (e.g., by modifying the page tables to mark present a page that had not been present). In these cases, the VMM can then perform a VM entry to the guest, which will resume execution at the point at which the exception occurred. If the VM exit was due to a nested fault, the VMM can use event injection to deliver to the guest that event whose delivery encountered that nested fault.

Interrupt Virtualization

When a VMM has an interrupt to deliver to a guest OS, it can do so using event injection with the next VM entry. If guest software is not ready for an interrupt (e.g., because EFLAGS.IF = 0), the VMM can instead re-enter the guest having set the interrupt-window exiting VM-execution control. A VM exit will occur the next time the guest is ready for an interrupt. A VMM can then use event injection as part of the next VM entry.

Lazy Floating-Point State Processing

The IA-32 architecture includes features by which an OS can avoid the time-consuming restoring the floating-point state when activating a user process that does not use the floating-point unit. It does this by setting the TS bit in control register CR0. If a user process then tries to use the floating-point unit, a device-not-available fault (exception 7 = #NM) occurs. The OS can respond to this by restoring the floating-point state and by clearing CR0.TS, which prevents the fault from recurring.

VT-x includes features by which a VMM can process floating-point state lazily, even when supporting a guest OS that does so also. We outline how this may be done.

Before entering a guest whose floating-point state has not been restored, a VMM can do the following:

- Set the TS bit in the CR0 field in the guest-state area; this ensures that any guest floating-point access causes a #NM.
- Set bit 7 (corresponding to #NM) in the exception bitmap; this ensures that any #NM causes a VM exit.
- Set the TS bit in the CR0 guest/host mask; this ensures that any guest attempt to modify CR0.TS causes a VM exit.

- Set the TS bit in the CR0 read shadow to the value expected by guest software (determined on VM exits caused by guest attempts to modify CR0.TS).

In response to a VM exit caused by a #NM, a VMM can check the value of the TS bit in the CR0 read shadow. If it is set, the guest would have incurred its own #NM; the VMM can use event injection to deliver it to the guest. Otherwise, the VMM can do the following:

- Restore the guest's floating-point state.
- Set the TS bit in the CR0 field in the guest-state area to the value expected by guest software.
- Clear bit 7 in the exception bitmap; this ensures that the guest OS will handle any subsequent #NM.
- Clear the TS bit in the CR0 guest/host mask; this allows the guest to modify CR0.TS freely.

VMM Usage of VT-i Architecture Features

Instruction Emulation

The VMM virtualization intercept handler is responsible for emulating certain instructions for a guest OS including side effects of successful emulation. One example of instruction emulation is the `MOV-from-PTA` instruction. The VMM emulates this instruction by placing the guest PTA value in the target register of the instruction. Since the VMM has successfully implemented the `MOV-from-PTA` instruction, it needs to implement the side effects of the instruction execution required by the Itanium architecture. In this example the VMM must also update the value in the cr.iipa register, which records the last successfully executed instruction with PSR.ic equal to 1.

Virtualization Configuration

VT-i is capable of providing a virtualization intercept on every access to privileged resources that may be required or desired for certain VMM implementations. VT-i also provides a way for a VMM to specify virtualization policies on certain resources in advance such that interceptions to the VMM can be reduced for high frequency operations. This functionality is provided through virtualization-accelerations, virtualization-disables, and new synchronization services. One example is the interruption control register reads. Guest OS interruption handlers read interruption control registers frequently and cause a lot of interceptions into the VMM. The *interruption control register read acceleration* allows VMM software to provide preset values for all interruption control registers in the VPD and invoke the PAL write synchronization service before returning to a guest handler. When this acceleration is enabled, guest reads of the interruption control registers are not intercepted to the VMM; instead the value preset by the VMM is returned to the guest. Similarly, the *interruption control register write acceleration* allows the guest to write to interruption control registers without VMM interceptions. VMM can invoke the PAL read synchronization service to obtain the latest values written by the guest and perform any virtualization functions required before emulating the return from interrupt (*rfi*) instruction of the guest handler. All other accelerations and disables in VT-i have the same goal—to allow the VMM to specify the virtualization policies of the privileged resources ahead of time such that guest instructions can execute without interceptions to the VMM.

External and PAL-Based Interruption Handling

In addition to implementing policies to virtualize accesses to privileged resources on the processor, VMM software also needs to virtualize external interruptions as well as accesses to platform resources that are considered privileged. For example, VMM software will continue to handle external interruptions or PAL-based interruptions even if the guest OS had masked these interruptions.

VMM software delivers guest external interrupts only when they are unmasked. When unmasked, the VMM delivers the interruption to the guest handler required by the architecture. For example, the VMM needs to set up the values of the guest interruption control registers, PSR fields, and register stack engine (RSE) state. Since some of the RSE state is not accessible by VMM software, VT-i provides PAL service to allow VMMs to invoke guest handlers correctly.

VMM software registers the corresponding handlers for PAL-based interruptions (e.g., initialization and machine check events) and provides the virtualization policies for these events. VT-i makes no changes to the handling of PAL-based interruptions. The handling and propagation of these events from the VMM to the guest OS is VMM design specific.

FUTURE OF INTEL VIRTUALIZATION ARCHITECTURE

The following features are anticipated for future processors supporting VT-x:

- *NMI-window exiting*. The interrupt-window exiting VM-execution control (described earlier) causes a VM exit when a guest is ready for maskable external interrupts, allowing a VMM to deliver such interrupts in a timely way. NMI-window exiting provides corresponding support for non-maskable interrupts (NMIs), which are blocked by other conditions than those that block maskable external interrupts.

- *Virtual-processor identifiers (VPIDs)*. This feature allows a VMM to assign a different non-zero VPID to each virtual processor (the zero VPID is reserved for the VMM). The CPU can use VPIDs to tag translations in the TLBs. This feature eliminates the need for TLB flushes on every VM entry and VM exit and eliminates the adverse impact of those flushes on performance.
- *Extended page tables (EPT)*. When this feature is active, the ordinary IA-32 page tables (referenced by control register CR3) translate from linear addresses to *guest-physical* addresses. A separate set of page tables (the *EPT tables*) translate form guest-physical addresses to the *host-physical* addresses that are used to access memory. As a result, guest software can be allowed to modify its own IA-32 page tables and directly handle page faults. This allows a VMM to avoid the VM exits associated with page-table virtualization, which are a major source of virtualization overhead without EPT.

CONCLUSION

While the use of virtualization was once confined to proprietary server and mainframe computing systems, established and emerging applications for virtualization 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.

VT-x and VT-i are the first components of Intel VT, a series of processor 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 and thus enable the development of simpler VMM software that supports a wider range of legacy and future OS's 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: Andrew V. Anderson, Steven M. Bennett, Jason Brandt, Stephen Fischer, Gideon Gerzon, Gary Hammond, Stalinselvaraj Jeyasingh, Alain Kägi, Mike Kozuch, Tariq Masood, Sanjoy Mondal, Rajesh Parthasarathy, Rajesh Sankaran, Sebastian Schönberg, and Larry Smith. We also thank Fernando C. M. Martins for his many contributions to the development of this paper.

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," in *Proceedings 19th ACM Symp. Operating Systems Principles*, ACM Press, 2003, pp. 164–177.

[3] Microsoft Corp., "Microsoft Virtual Server 2005 Technical Overview," 2004,; at 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," in *Proceedings 5th Symp. Operating Systems Design and Implementation*, The Usenix Association, 2002, pp. 181–194.

[5] A. Whitaker, M. Shaw, and S. Gribble, "Scale and Performance in the Denali Isolation Kernel," in *Proceedings 5th Symp. Operating Systems Design and Implementation*, The Usenix Association, 2002, pp. 195–210.

[6] Intel Corp., "IA-32 Intel® Architecture Software Developer's Manuals;" at http://www.intel.com/design/pentium4/manuals/index_new.htm

[7] Intel Corp., "Intel® Itanium® Architecture Software Developer's Manual-Volume 2: System Architecture, Revision 2.2," document number 1805, 2006; at ftp://download.intel.com/design/Itanium/manuals/24531805.pdf

AUTHORS' BIOGRAPHIES

Gil Neiger is a principal engineer in Intel's Corporate Technology Group and leads development of the VT-x architecture. He received his Ph.D. degree in Computer Science from Cornell University.

Amy Santoni is a principal engineer in Intel's Digital Enterprise Group and leads the Itanium Architecture Team. She is one of the principal architects of VT-i. Amy has been with Intel for 13 years working in design, validation, firmware, and architecture positions. She received her B.S. degree in Computer Engineering from the University of Michigan.

Felix Leung is a staff engineer in Intel's Digital Enterprise Group and is responsible for the definition and development of VT-i architecture. He has been with Intel for 11 years and has held verification, design, and architecture positions in various IA-32 and Itanium processor design projects. He received his B.S. degree in Computer Science from the University of Wisconsin-Madison.

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 his M.S. degree in Computer Engineering from Clemson University.

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 his Ph.D. degree in Computer Science and Engineering from the University of Michigan.

[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