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 for Directed I/O

Intel® Virtualization Technology for Directed I/O

Darren Abramson, Mobility Group, Intel Corporation
Jeff Jackson, Corporate Technology Group, Intel Corporation
Sridhar Muthrasanallur, Digital Enterprise Group, Intel Corporation
Gil Neiger, Corporate Technology Group, Intel Corporation
Greg Regnier, Corporate Technology Group, Intel Corporation
Rajesh Sankaran, Corporate Technology Group, Intel Corporation
Ioannis Schoinas, Corporate Technology Group, Intel Corporation
Rich Uhlig, Corporate Technology Group, Intel Corporation
Balaji Vembu, Digital Enterprise Group, Intel Corporation
John Wiegert, Corporate Technology Group, Intel Corporation

Index words: Virtualization, I/O, VMM, DMA, Interrupts

ABSTRACT

Intel® Virtualization Technology$^{\Delta}$ for Directed I/O (VT-d) is the next important step toward comprehensive hardware support for the virtualization of Intel® platforms. VT-d extends Intel's Virtualization Technology (VT) roadmap from existing support for IA-32 (VT-x) [1] and Itanium® processor (VT-i) [2] virtualization to include new support for I/O-device virtualization. This paper surveys a variety of established and emerging techniques for I/O virtualization and outlines their associated problems and challenges. We then detail the architecture of VT-d and describe how it enables the industry to meet the future challenges of I/O virtualization.

INTRODUCTION

There are a number of existing and emerging usage models where support for I/O virtualization is, or will become, increasingly important. Performance, scalability, cost, trust, reliability, and availability are all important considerations, and their relative importance can vary depending upon usage models and the market segment in which they are deployed.

There are two key requirements that are common across market segments and usage models. The first requirement is protected access to I/O resources from a given virtual machine (VM), such that it cannot interfere with the operation of another VM on the same platform. This isolation between VMs is essential for achieving availability, reliability, and trust. The second major requirement is the ability to share I/O resources among multiple VMs. In many cases, it is not practical or cost-effective to replicate I/O resources (such as storage or network controllers) for each VM on a given platform.

First we consider the importance of I/O virtualization in the data center. Many server applications are I/O intensive, especially for networking and storage. Key requirements within the data center include scalability and performance to enable server consolidation. Reliability and availability are important as mission-critical applications move onto virtualized data center servers and infrastructures.

In the case of server consolidation, virtualization is used to deploy multiple VMs (each containing an operating system (OS) and associated services and applications) onto a single server. This consolidation is done primarily to utilize the underlying server hardware more effectively. Many server applications require a significant amount of I/O performance, and so it follows that the consolidation of multiple server applications will need a scalable and high-performance solution for I/O virtualization. The scalability requirement comes from the fact that the total network and storage I/O required from a given server platform is the aggregate of the I/O requirements of the multiple consolidated applications. I/O performance is needed by each VM to satisfy a wide range of server applications with varied and demanding I/O performance requirements.

Next we look at the importance of I/O virtualization in client platforms. For most client platforms, I/O scalability and performance are relatively modest as compared to servers, but tend to be more sensitive to cost and trust issues.

In the case of the enterprise client, virtualization can be used to create a self-contained operating environment, or "virtual appliance," that is dedicated to capabilities such as manageability or security. These capabilities generally need protected and secure access to a network device to communicate with down-the-wire management agents and to monitor network traffic for security threats. For example, a security agent within a VM requires protected access to the actual network controller hardware. This agent can then intelligently examine network traffic for malicious payloads or suspected intrusion attempts before the network packets are passed to the guest OS, where user applications might be affected.

This virtual-appliance model can be applied beyond the enterprise client. Workstations and home computers can use this technique for management, security, content protection, and a wide variety of other dedicated services. The type of service deployed may dictate that various types of I/O resources, graphics, network, and storage devices, be isolated from the OS where the user's applications are running.

In this paper we survey a variety of existing and emerging techniques for addressing the above requirements of I/O virtualization. We begin in the next section by studying different options for Virtual Machine Monitor (VMM) structuring and software architecture, and then we discuss various techniques for sharing I/O resources among multiple guest OSs. Our survey highlights various challenges faced by today's I/O-virtualization techniques, and it underscores the need for new forms of hardware support to facilitate I/O-resource assignment, protection, and sharing. We then detail the architecture of Intel's VT-d and explain how it helps to establish a new platform infrastructure for addressing the challenges of I/O virtualization in future platforms based on Intel® technology.

VMM SOFTWARE ARCHITECTURE OPTIONS

As background, we identify and compare three distinct types of virtualization layer (or VMM) software architectures in this section (see Figure 1):

- OS-hosted VMMs
- Stand-alone hypervisor VMMs
- Hybrid VMMs

Each of these styles of VMM software architecture has its pros and cons, and the choice often depends on the particular requirements of a given usage model or market segment.

OS-Hosted VMMs

One approach to VMM software architecture is to build on the infrastructure of an existing OS [3] [15]. Such *OS-hosted VMMs* consist of a privileged ring-0 component (shown as the "VMM kernel" in Figure 1) that runs alongside the kernel of the hosting OS, and that obtains control of system resources–such as CPUs and system memory – to create an execution environment for one or more guest OSs. The VMM kernel context switches between host-OS and guest-OS state at periodic intervals as dictated by scheduling policy, or whenever host-OS support is required (e.g., to service hardware interrupts from a physical I/O device that is programmed by a host-OS device driver). Although the guest OS is allowed to directly execute on a physical CPU and to directly access certain portions of host physical memory subject to the control of the VMM kernel, any accesses to I/O devices are typically intercepted by the VMM kernel and proxied to a second, user-level component of the VMM (shown in Figure 1 as a *User-Level Monitor* or *ULM*). The ULM runs as an ordinary process of the host OS, and it contains virtual I/O-device models that service I/O requests from guest OSs. Device models in the ULM call the facilities of the underlying host OS via its file system and networking and graphics APIs to handle I/O requests from guest OSs.

An OS-hosted VMM architecture offers several advantages: the VMM can leverage any I/O device drivers that have been developed for the hosting OS, which can significantly ease porting of the VMM to a range of different physical host platforms. Further, the VMM can leverage other facilities of the host OS, such as code for scanning I/O busses, to perform I/O resource discovery and to manage host platform power-management functions.

A disadvantage of an OS-hosted VMM is that it is only as reliable, available, and secure as the host OS upon which it depends: If the host OS fails or must be rebooted (e.g., to install a software security patch), then all other guest OSs must be taken out of service as well. An OS-hosted VMM is also subject to the CPU scheduling policies of the host OS, which serves not only the VMM and its guest OSs, but also other applications running above the host OS. Depending on the security, availability, or real-time quality-of-service requirements of a given usage model, these disadvantages may not be acceptable, and alternative VMM software architectures may be warranted.

Stand-Alone Hypervisor VMMs

One such alternative approach is to structure the VMM as a stand-alone *hypervisor* that does not depend on a hosting

OS [4, 10, 11]. A hypervisor-style VMM incorporates its own I/O device drivers, device models, and scheduler.

A hypervisor-style VMM can fully control provisioning of physical platform resources, enabling it to provide scheduling and quality-of-service guarantees to its guest OSs. An additional advantage of a hypervisor-based VMM is that the code paths from guest OSs requests for I/O services to the actual physical I/O device drivers are typically shorter than in an OS-hosted VMM, which requires I/O requests to traverse two I/O stacks, first that of the guest OS, and then that of the host OS. Further, by controlling and limiting the size of the hypervisor kernel, the VMM can provide enhanced security and reliability through a smaller trusted computing base (TCB) [5, 9].

The advantages of a hypervisor-style VMM come at the expense of limited portability, because the necessary I/O-device drivers for any given physical platform must be developed to run within the hypervisor. More advanced system functions, such as ACPI-based system power management–which are inherited from the host OS in a hosted VMM–must also be reimplemented in a hypervisor-based VMM. While not as complex as a full modern OS, a mature hypervisor-based VMM can grow to a significant size over time, gradually compromising some of the benefits noted earlier (e.g., improved security through limiting the size of the TCB).

Hybrid VMMs

In an effort to retain some of the security and reliability benefits of hypervisor-style VMM architecture, while at the same time leveraging the facilities of an existing OS and its associated device drivers as in an OS-hosted VMM, some VMMs adopt a *hybrid* approach [6, 7, 9].

In a hybrid VMM architecture, a small hypervisor kernel (shown in Figure 1 as a *μ-hypervisor*) controls CPU and memory resources, but I/O resources are programmed by device drivers that run in a deprivileged *service OS*. The service OS functions in a manner similar to that of a host OS in that the VMM is able to leverage its existing device drivers. However, because the service OS is deprivileged by the μ-hypervisor, and because it operates solely on behalf of the VMM (i.e., it does not support other, arbitrary user applications), it is possible to improve the overall security and reliability of the system.

While a hybrid VMM architecture offers the promise of retaining the best characteristics of hosted- and hypervisor-style VMMs, it does introduce new challenges, including new performance overheads, due to frequent privilege-level transitions between guest OS and service OS through the μ-hypervisor. Further, the full benefits of deprivileging a service OS are only possible with new hardware support for controlling device Direct Memory Access (DMA) via the μ-hypervisor. As we will see later, such hardware support is provided by VT-d.

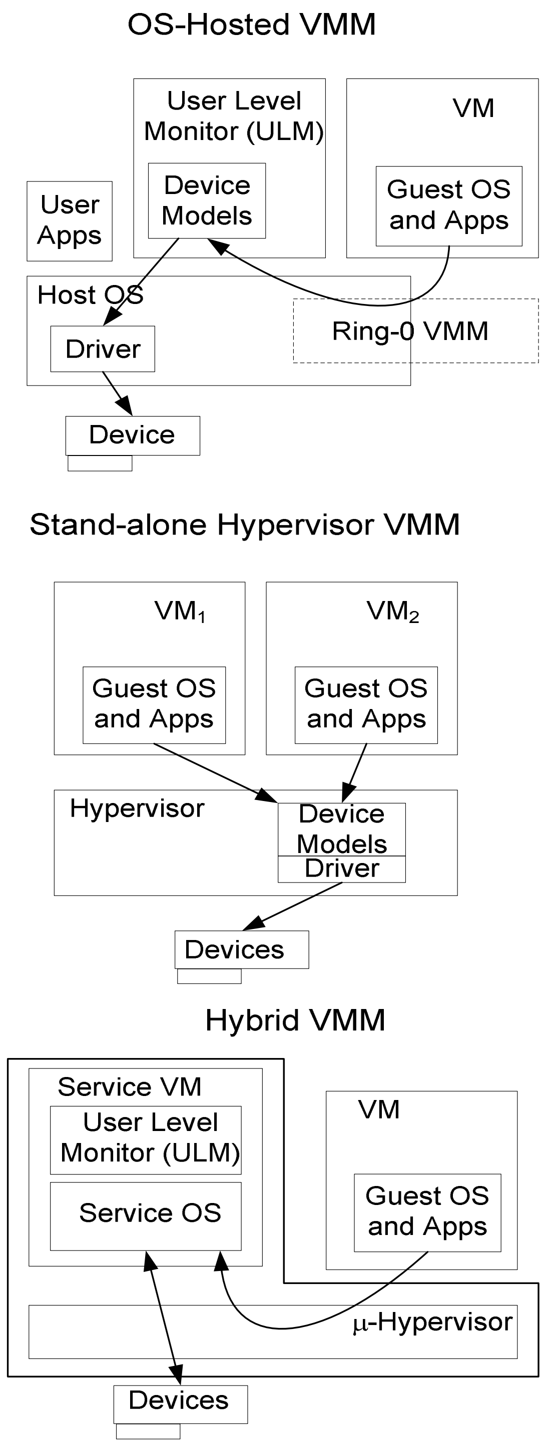

Figure 1: VMM software architectures

CURRENT I/O VIRTUALIZATION TECHNIQUES

When virtualizing an I/O device, it is necessary for the underlying virtualization software to service several types of operations for that device. Interactions between software and physical devices include the following:

- *Device discovery*: a mechanism for software to discover, query, and configure devices in the platform.
- *Device control*: a mechanism for software to communicate with the device and initiate I/O operations.
- *Data transfers*: a mechanism for the device to transfer data to and from system memory. Most devices support DMA in order to transfer data.
- *I/O interrupts*: a mechanism for hardware to be able to notify the software of events and state changes.

Each of these interactions is discussed, covering implementation, challenges, advantages, and disadvantages of each of the common virtualization techniques. The VMM could be a single monolithic software stack or could be a combination of a hypervisor and specialized guests (as shown in Figure 1). The type of VMM architecture used is independent of the concepts discussed in this section, but will become relevant later in our discussion.

Emulation

I/O mechanisms on native (non-virtualized) platforms are usually performed on some type of hardware device. The software stack, commonly a driver in an OS, will interface with the hardware through some type of memory-mapped (MMIO) mechanism, whereby the processor issues instructions to read and write specific memory (or port) address ranges. The values read and written correspond to direct functions in hardware.

Emulation refers to the implementation of real hardware completely in software. Its greatest advantage is that it does not require any changes to existing guest software. The software runs as it did in the native case, interacting with the VMM emulator just as though it would with real hardware. The software is unaware that it is really talking to a *virtualized* device. In order for emulation to work, several mechanisms are required.

The VMM must expose a device in a manner that it can be *discovered* by the guest software. An example is to present a device in a PCI configuration space so that the guest software can "*see*" the device and discover the memory addresses that it can use to interact with the device.

The VMM must also have some method for capturing reads and writes to the device's address range, as well as capturing accesses to the device-discovery space. This enables the VMM to *emulate* the real hardware with which the guest software believes it is interfacing.

The device (usually called a device model) is implemented by the VMM completely in software (see Figure 2). It may be accessing a real piece of hardware in the platform in some manner to service some I/O, but that hardware is independent of the device model. For example, a guest might see an Integrated Drive Electronics (IDE) hard disk model exposed by the VMM, while the real platform actually contains a Serial ATA (SATA) drive.

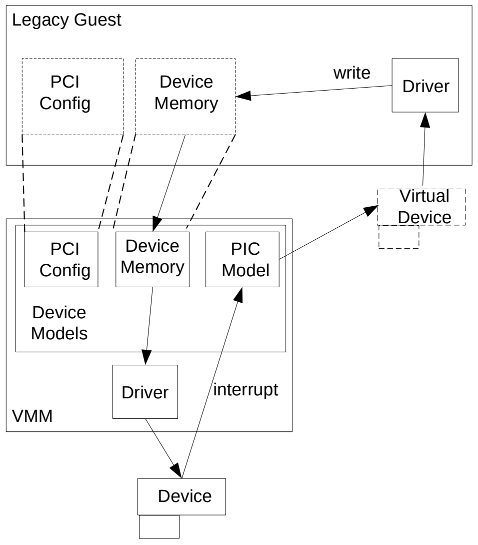

Figure 2: Device emulation model

The VMM must also have a mechanism for injecting interrupts into the guest at appropriate times on behalf of the emulated device. This is usually accomplished by emulating a Programmable Interrupt Controller (PIC). Once again, when the guest software exercises the PIC, these accesses must be trapped and the PIC device modeled appropriately by the VMM. While the PIC can be thought of as just another I/O device, it has to be there for any other interrupt-driven I/O devices to be emulated properly.

Emulation facilitates migration of VMs from one platform to another. Since the devices are purely emulated and have no ties to physical devices in the platform, it is easy to move a VM to another platform where the VMM can support the exact same emulated devices. If the guest VM did have some tie to any platform physical devices, those same physical devices would need to be present on any platform to which the VM was migrated.

Emulation also facilitates the sharing of platform physical devices of the same type, because there are instances of an emulation model exposed to potentially many guests. The VMM can use some type of sharing mechanism to allow all guest's emulation models access to the services of a single physical device. For example, the traffic from many guests with emulated network adapters could be bridged onto the platform's physical network adapter.

Since emulation presents to the guest software the exact interface of some existing physical hardware device, it can support a number of different guest OSs in an OS-independent manner. For example, if a particular storage device is emulated completely, then it will work with any software written for that device, independent of the guest OS, whether it be Windows*, Linux*, or some other IA-based OS. Since most modern OSs ship with drivers for many well-known devices, a particular device make and model can be selected for emulation such that it will be supported by these existing legacy environments.

While emulation's greatest advantage is that there are no requirements to modify guest device drivers, its largest detractor is low performance. Each interaction of the guest device driver with the emulated device hardware requires a transition to the VMM, where the device model performs the necessary emulation, and then a transition back to the guest with the appropriate results. Depending upon the type of I/O device that is being emulated, many of these transactions may be required to actually retrieve data from the device. These activities add considerable overhead compared to normal software-hardware interactions in a non-virtualized system. Most of this new overhead is compute-bound in nature and increases CPU utilization. The timing involved in each interaction can also accumulate to increase overall latency.

Another disadvantage of emulation is that the device model needs to emulate the hardware device very accurately, sometimes to the revision of the hardware, and must cover all corner cases. This can result in the need for "bug emulation" and problems arising with new revisions of hardware.

Paravirtualization

Another technique for virtualizing I/O is to modify the software within the guest, an approach that is commonly referred to as paravirtualization [4, 8]. The advantage of I/O paravirtualization is better performance. A disadvantage is that it requires modification of the guest software, in particular device drivers, which limits its applicability to legacy OS and device-driver binaries.

With paravirtualization (see Figure 3) the altered guest software interacts directly with the VMM, usually at a higher abstraction level than the normal hardware/software interface. The VMM exposes an I/O type-specific API, for example, to send and receive network packets–in the case of a network adaptor. The altered software in the guest then uses this VMM API instead of interacting directly with a hardware device interface.

Paravirtualization reduces the number of interactions between the guest OS and VMM, resulting in better performance (higher throughput, lower latency, reduced CPU utilization), compared to device emulation.

Instead of using an emulated interrupt mechanism, paravirtualization uses an eventing or callback mechanism. This again has the potential to deliver better performance, because interactions with a PIC hardware interface are eliminated, and because most OS's handle interrupts in a staged manner, adding overhead and latency. First, interrupts are fielded by a small Interrupt Service Routine (ISR). An ISR usually acknowledges the interrupt and schedules a corresponding worker task. The worker task is then run in a different context to handle the bulk of the work associated with the interrupt. With an event or callback being initiated directly in the guest software by the VMM, the work can be handled directly in the same context. With some implementations, when the VMM wishes to introduce an interrupt into the guest, it must force the running guest to exit to the VMM, where any pending interrupts can be picked up when the guest is reentered. To force a running guest to exit, a mechanism like IPI can be used. But this again adds overhead compared to a direct callback or event. Again, the largest detractor to this approach is that the interrupt handling mechanisms of the guest OS kernel must also be altered.

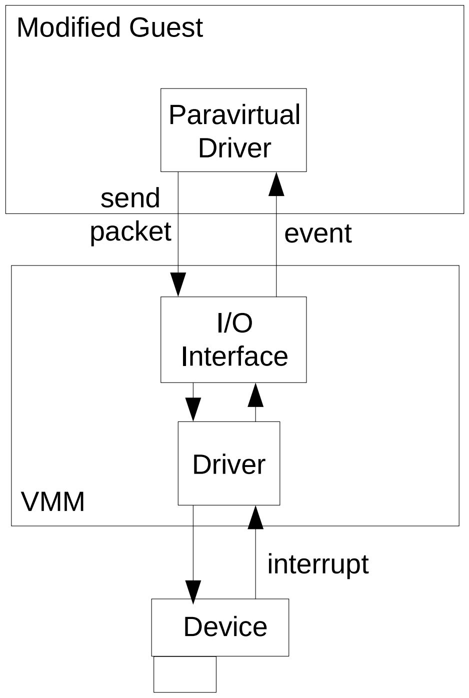

Figure 3: Device paravirtualization

Since paravirtualization involves changing guest software, usually the changed components are specific to the guest environment. For instance, a paravirtualized storage driver for Windows XP* will not work in a Linux environment. Therefore, a separate paravirtualized component must be developed and supported for each targeted guest environment. These changes require apriori knowledge of which guest environments will be supported by a particular VMM.

As with device emulation, paravirtualization is supportive of VM migration, provided that the VM is migrated to a platform that supports the same VMM APIs required by the guest software stack.

Sharing of any platform physical devices of the same type is supported in the same manner as emulation. For example, guests using a paravirtualized storage driver to read and write data could be backed by stores on the same physical storage device managed by the VMM.

Paravirtualization is increasingly deployed to satisfy the performance requirements of I/O-intensive applications. Paravirtualization of I/O classes that are performance sensitive, such as networking, storage, and high-performance graphics, appears to be the method of choice in modern VMM architecture. As described, para-virtualization of I/O decreases the number of transitions between the client VM and the VMM, as well as eliminates most of the processing associated with device emulation.

Paravirtualization leads to a higher level of abstraction for I/O interfaces within the guest OS. I/O-buffer allocation and management policies that are aware of the fact that they are virtualized can be used for more efficient use of the VT-d protection and translation facilities than would be possible with an unmodified driver that relies on full device emulation.

At least three of the major VMM vendors have adopted the capability to paravirtualize I/O in order to accomplish greater scaling and performance. Xen* and VMware already have the ability to run paravirtualized I/O drivers and Microsoft's plans include I/O paravirtualization in its next-generation VMM.

Direct Assignment

There are cases where it is desirable for a physical I/O device in the platform to be directly *owned* by a particular guest VM. Like emulation, direct assignment allows the owning guest VM to interface directly to a standard device hardware interface. Therefore, direct device assignment provides a native experience for the guest VM, because it can reuse existing drivers or other software to talk directly to the device.

Direct assignment improves performance over emulation because it allows the guest VM device driver to talk to the device in its native hardware command format eliminating the overhead of translating from the device command format of the virtual emulated device. More importantly, direct assignment increases VMM reliability and decreases VMM complexity since complex device drivers can be moved from the VMM to the guest.

Direct assignment, however, is not appropriate for all usages. First, a VMM can only allocate as many devices as are physically present in the platform. Second, direct assignment complicates VM migration in a number of ways. In order to migrate a VM between platforms, a similar device type, make, and model must be present and available on each platform. The VMM must also develop methods to extract any physical device state from the source platform, and to restore that state at the destination platform.

Moreover, in the absence of hardware support for direct assignment, direct assignment fails to reach its full potential in improving performance and enhancing reliability. First, platform interrupts may still need to be fielded by the VMM since it owns the rest of the physical platform. These interrupts must be routed to the appropriate guest–in this case the one that owns the physical device. Therefore, there is still some overhead in this relaying of interrupts. Second, existing platforms do not provide a mechanism for a device to directly perform data transfers to and from the system memory that belongs to the guest VM in an efficient *and* secure manner. A guest VM is typically operating in a subset of the real physical address space. What the guest VM believes is its physical memory really is not; it is a subset of the system memory virtualized by the VMM for the guest. This addressing mismatch causes a problem for DMA-capable devices. Such devices place data directly into system memory without involving the CPU. When the guest device driver instructs the device to perform a transfer it is using guest physical addresses, while the hardware is accessing system memory using host physical addresses.

In order to deal with the address space mismatch, VMMs that support direct assignment may employ a pass-through driver that intercepts all communication between the guest VM device driver and the hardware device. The pass-through driver performs the translation between the guest physical and real physical address spaces of all command arguments that refer to physical addresses. Pass-through drivers are device-specific since they must decode the command format for a specific device to perform the necessary translations. Such drivers perform a simpler task than traditional device drivers; therefore, performance is improved over emulation. However, VMM complexity remains high, thereby impacting VMM reliability. Still, the performance benefits have proven sufficient to employ this method in VMMs targeted to the server space, where it is acceptable to support direct assignment for only a relatively small number of common devices.

VMM Software Architecture Implications

Different I/O virtualization methods are not equally applicable to all VMM software architecture options.

Emulation is the most general I/O virtualization method, able to expose standard I/O devices to an unmodified guest OS. Accordingly, it is widely employed in existing OS-hosted, stand-alone hypervisor or hybrid VMM implementations.

As already mentioned, paravirtualization is increasingly being deployed in many VMMs to improve performance for common guests. It is readily applicable to stand-alone hypervisor VMMs. It can also be used in the interaction between the guest OS and the ULM in an OS-hosted VMM or can be used in the guest OS and the service VM in a hybrid VMM.

Direct assignment is used in cases where the guest OS cannot be modified either because it is difficult to do so or the paravirtualized guest device drivers are not qualified for a specific application. However, it is difficult to introduce direct assignment in an OS-hosted VMM since in general, such VMMs do not own real platform devices and do not maintain device drivers for such devices. On the other hand, direct assignment naturally reduces complexity in stand-alone hypervisor and hybrid VMMs since device drivers can be moved to the guest OS or service OSs, respectively. This reduced complexity is not possible with either emulation or paravirtualization.

As our discussion suggests, it is quite likely that a VMM can employ many different techniques for I/O virtualization concurrently. For instance, in the context of hybrid VMM, direct assignment might be used to assign a platform physical device to a particular guest VM, whose responsibility it is to share that device with many guests. Depending upon the needs and requirements of the guest, it may offer both emulated device models, as well as paravirtualized solutions to the different guests. A common configuration is to provide paravirtualized solutions for the most common guest environments, while an emulation solution is offered to support all other legacy environments.

IOVM Architecture

A major emerging trend among developers of virtualization software, in particular for I/O processing and sharing, is the VMM system decomposition.

The trend for the software architecture of VMMs is to move from a monolithic hypervisor model towards a software architecture that decomposes the VMM into a very thin privileged "micro-hypervisor" that resides just above the physical hardware, and one or more special-purpose VMs that are de-privileged relative to the hypervisor, and are responsible for services and policy. With regard to I/O virtualization, these deprivileged components of the VMM can be responsible for I/O processing and I/O resource sharing. We call this general architecture the "IOVM" model (see Figure 4). The IOVM model is a generalization of the hybrid VMM architecture in that I/O devices can be allocated to different service VMs specialized for the specific I/O function (e.g., network VM, storage VM, etc.).

Two major benefits of the IOVM model are the ability to use unmodified device drivers within the IOVM and the isolation of the physical device and its driver(s) from the other guest OSs, applications, and hypervisor. The use of unmodified drivers is possible because these drivers can run in a separate OS environment, in contrast to a monolithic hypervisor where new drivers are often written for the VMM environment. The isolation of the device and its driver protect the guest VMs from driver crashes, that is, the IOVM may crash due to a driver failure without severely affecting the guest OSs. A disadvantage of the IOVM model is that there is additional overhead incurred, due to additional communication and data movement between the guest OS and the IOVM. This performance penalty can be offset by paravirtualizing the interface of the IOVM, thus minimizing the number of interactions. The Xen VMM has implemented this architecture as "Isolated Driver Domains" [6], and Microsoft is in the process of developing a version of this architecture in their next generation of VMMs [7].

Direct assignment of I/O devices to IOVMs directly facilitates this usage model and is becoming increasingly important as VMMs are transitioning to this architecture. As we have seen, however, software by itself is not capable of fully protecting the system from errant DMA traffic between the I/O device and system memory while at the same time eliminating all device-specific functionality in the VMM. Hardware support on the platform closes this gap, by allowing the device to be safely assigned to an IOVM, thus allowing full protection from errant DMA transfers.

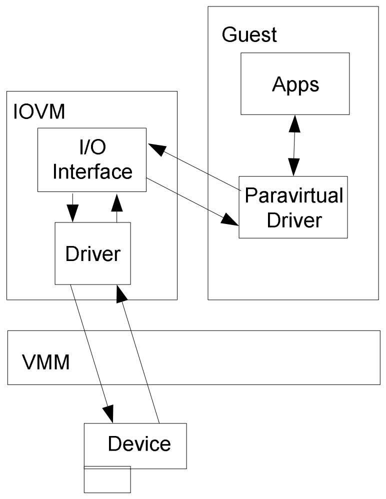

Figure 4: IOVM software architecture

PLATFORM HARDWARE SUPPORT FOR I/O VIRTUALIZATION

To enforce the isolation, security, reliability, and performance benefits of direct assignment, we need efficient hardware mechanisms to constrain the operation of I/O devices. The primary I/O device accesses that require this isolation are device transfers (DMAs) and interrupts. CPU virtualization mechanisms are sufficient to efficiently perform device discovery and schedule device operations.

Accordingly, VT-d [12] provides the platform hardware support for DMA and interrupt virtualization.

DMA Remapping

DMA remapping facilities have been implemented in a variety of contexts in the past to facilitate different usages. In workstations and server platforms, traditional I/O memory management units (IOMMUs) have been implemented in PCI root bridges to efficiently support scatter/gather operations or I/O devices with limited DMA addressability [17]. Other well-known examples of DMA remapping facilities include the AGP Graphics Aperture Remapping Table (GART) [18], the Translation and Protection Table (TPT) defined in the Virtual Interface Architecture [14], and subsequently influencing a similar capability in the InfiniBand Architecture [16] and Remote DMA (RDMA) over TCP/IP specifications [19]. DMA remapping facilities have also been explored in the context of NICs designed for low latency cluster interconnects [15].

Traditional IOMMUs typically support an aperture-based architecture. All DMA requests that target a programmed aperture address range in the system physical address space are translated irrespective of the source of the request. While this is useful for handling legacy device limitations (such as limited DMA addressability or scatter/gather capabilities), they are not adequate for I/O virtualization usages that require full DMA isolation.

The VT-d architecture is a generalized IOMMU architecture that enables system software to create multiple DMA protection domains. A protection domain is abstractly defined as an isolated environment to which a subset of the host physical memory is allocated. Depending on the software usage model, a DMA protection domain may represent memory allocated to a VM, or the DMA memory allocated by a guest-OS driver running in a VM or as part of the VMM itself. The VT-d architecture enables system software to assign one or more I/O devices to a protection domain. DMA isolation is achieved by restricting access to a protection domain's physical memory from I/O devices not assigned to it, through address-translation tables.

The I/O devices assigned to a protection domain can be provided a view of memory that may be different than the host view of physical memory. VT-d hardware treats the address specified in a DMA request as a DMA virtual address (DVA). Depending on the software usage model, a DVA may be the Guest Physical Address (GPA) of the VM to which the I/O device is assigned, or some software-abstracted virtual I/O address (similar to CPU linear addresses). VT-d hardware transforms the address in a DMA request issued by an I/O device to its corresponding Host Physical Address (HPA).

Figure 5 illustrates DMA address translation in a multi-domain usage. I/O devices 1 and 2 are assigned to protection domains 1 and 2, respectively, each with its on view of the DMA address space.

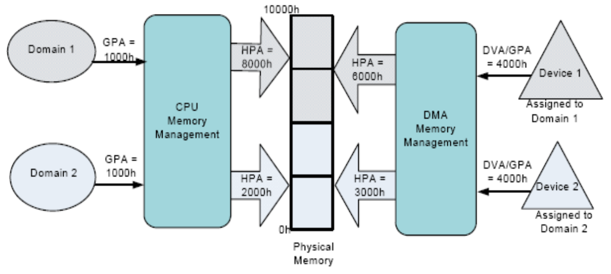

Figure 5: DMA remapping

Figure 6 illustrates a PC platform configuration with VT-d hardware implemented in the north-bridge component.

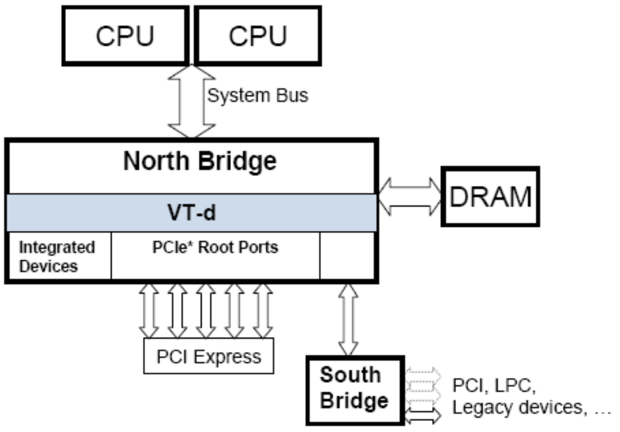

Figure 6: Platform configuration with VT-d

Mapping Devices to Protection Domains

To support multiple protection domains, the DMA remapping hardware must identify the device originating each DMA request. The requester identifier of a device is composed of its PCI Bus/Device/Function number assigned by PCI configuration software and uniquely identifies the hardware function that initiated the request. Figure 7 illustrates the requester-id as defined by the PCI specifications [20].

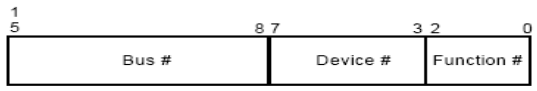

Figure 7: PCI requester identifier format

VT-d architecture defines the following data structures for mapping I/O devices to protection domains (see Figure 8):

- *Root-Entry Table*: Each entry in the root-entry table functions as the top-level structure to map devices for a specific PCI bus. The bus-number portion of the requester-id in DMA requests is used to index into the root-entry table. Each present root entry includes a pointer to a context-entry table.
- *Context-Entry Table*: Each entry in the context-entry table maps a specific I/O device on a bus to the protection domain to which it is assigned. The device and function-number portion of the requester-id is used to index into the context-entry table. Each present context entry includes a pointer to the address translation structures used to translate the address in the DMA request.

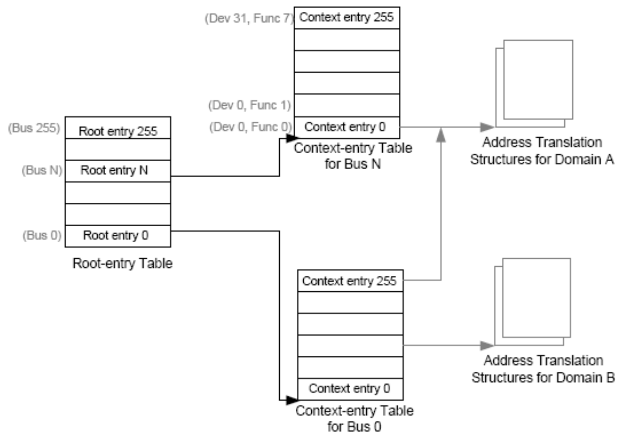

Figure 8: Device mapping structures

Address Translation

VT-d architecture defines a multi-level page-table structure for DMA address translation (see Figure 9). The multi-level page tables are similar to IA-32 processor page-tables, enabling software to manage memory at 4 KB or larger page granularity. Hardware implements the page-walk logic and traverses these structures using the address from the DMA request. The number of page-table levels that must be traversed is specified through the context-entry referencing the root of the page table. The page directory and page-table entries specify independent read and write permissions, and hardware computes the cumulative read and write permissions encountered in a page walk as the effective permissions for a DMA request. The page-table and page-directory structures are always 4 KB in size, and larger page sizes (2 MB, 1 GB, etc.) are enabled through super-page support.

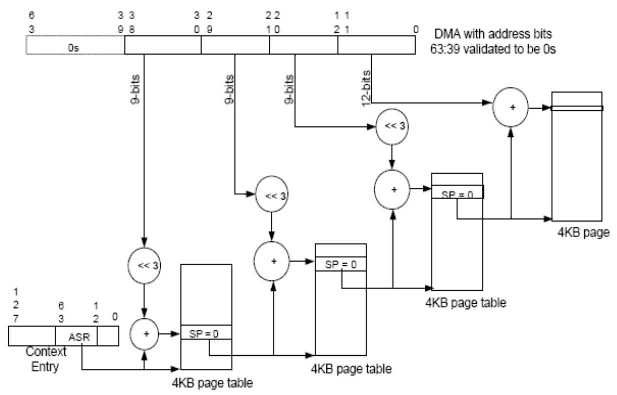

Figure 9: Example 3-level page table

Interrupt Remapping

For proper device isolation in a virtualized system, the interrupt requests generated by I/O devices must be controlled by the VMM. In the existing interrupt architecture for Intel platforms, a device may generate either a legacy interrupt (which is routed through I/O interrupt controllers) or may directly issue message signaled interrupts (MSIs) [20]. MSIs are issued as DMA write transactions to a pre-defined architectural address range, and the interrupt attributes (such as vector, destination processor, delivery mode, etc.) are encoded in the address and data of the write request. Since the interrupt attributes are encoded in the request issued by devices, the existing interrupt architecture does not offer interrupt isolation across protection domains.

The VT-d interrupt-remapping architecture addresses this problem by redefining the interrupt-message format. The new interrupt message continues to be a DMA write request, but the write request itself contains only a "message identifier" and not the actual interrupt attributes. The write request, like any DMA request, specifies the requester-id of the hardware function generating the interrupt.

DMA write requests identified as interrupt requests by the hardware are subject to interrupt remapping. The requestor-id of interrupt requests is remapped through the table structure. Each entry in the interrupt-remapping table corresponds to a unique interrupt message identifier from a device and includes all the necessary interrupt attributes (such as destination processor, vector, delivery mode, etc.). The architecture supports remapping interrupt messages from all sources including I/O interrupt controllers (IOAPICs), and all flavors of MSI and MSI-X interrupts defined in the PCI specifications.

Software Usages of DMA and Interrupt Remapping

The VT-d architecture enables DMA and interrupt requests from an I/O device to be isolated to its assigned protection domain. This capability makes possible a number of usages:

- *Remapping for legacy guests*: In this usage an I/O device is assigned directly to a VM running a legacy (virtualization unaware) environment. Since the guest OS has the guest-physical view of memory in this usage, the VMM programs the DMA remapping structures for the I/O device to support appropriate GPA to HPA mappings. Similarly, the VMM may program the interrupt-remapping structures to enable the interrupt requests from the I/O device to target the physical CPUs running the appropriate virtual CPUs of the legacy VM.
- *Remapping for IOMMU-aware guests*: An OS may be capable of using DMA and interrupt remapping hardware to improve its OS reliability or for handling specific I/O-device limitations. When such an OS is running within a VM, the VMM may expose virtual (emulated or paravirtualized) remapping hardware to the VM. The OS may create one or more protection domains each with its own DMA Virtual Address (DVA) space and program the virtual remapping hardware structures to support DVA to Guest Physical Address (GPA) mappings. The VMM must virtualize the remapping hardware by intercepting guest accesses to the virtual hardware and shadowing the virtual remapping structures to provide the physical hardware with structures for DVA to HPA mappings. Similar page table shadowing techniques are commonly used by the VMM for CPU MMU virtualization.

Hardware Caching and Invalidation Architecture

To improve DMA and interrupt-remapping performance, the VT-d architecture allows hardware implementations to cache frequently used remapping-structure entries. Specifically, the following architectural caching constructs are defined:

- *Context Cache*: Caches frequently used context entries that map devices to protection domains.
- *PDE (Page Directory Entry) Cache*: Caches frequently used page-directory entries encountered by hardware during page walks.
- *IOTLB (I/O Translation Look-aside Buffer)*: Caches frequently used effective translations (results of the page walk).

- *Interrupt Entry Cache*: Caches frequently used interrupt-remapping table entries.

These caching structures are fully managed by the hardware. When updating the remapping structures, the software is responsible for maintaining the consistency of these caches by invalidating any stale entries in the caches. VT-d architecture defines the following invalidation options:

- *Synchronous Invalidation*: The synchronous invalidation interface uses a set of memory-mapped registers for software to request invalidations and to poll for invalidation completions.
- *Queued Invalidation*: The queued-invalidation interface uses a memory-resident command queue for software to queue-invalidation requests. Software synchronizes invalidation completions with hardware by submitting an invalidation-wait command to the command queue. Hardware guarantees that all invalidation requests received before an invalidation-wait command are completed before completing the invalidation-wait command. Hardware signals the invalidation-wait command completion either through an interrupt or by coherently writing a software-specified memory location. The queued-invalidation interface enables usages where software can batch invalidation requests.

Scaling Address Translation Caches

Caching of the remapping structures enables hardware to minimize the DMA translation overhead that may otherwise be incurred when accessing the memory-resident translation structures. One of the challenges for DMA-remapping hardware implementations is to efficiently scale its hardware caching structures. Unlike CPU TLBs that support accesses from a CPU that is typically running one thread at a time, the DMA-remapping caches handle simultaneous DMA accesses from multiple devices, and often multiple DMA streams from a device.

This difference makes sizing the IOTLBs in DMA-remapping hardware implementations challenging, especially when the hardware design is re-used across a wide range of platform configurations. An approach to scaling the IOTLBs is to enable I/O devices to participate in DMA remapping by requesting translations for its own memory accesses from the DMA-remapping hardware and caching these translations locally on the I/O device in a Device-IOTLB.

To facilitate scaling of address translation caches, PCI Express* protocol extensions (referred to as Address Translation Services (ATS)) [22] are being standardized by the PCI Special Interest Group (PCI-SIG) [21]. ATS consist of a set of PCI transactions that allow the optimization of VT-d address translations. These extensions enable I/O devices to request translations from the root complex and for the root complex to return responses for each translation request. I/O devices may cache the returned translations in its local Device-IOTLBs and indicate if a DMA request is using un-translated address or translated address from its Device-IOTLB. To support usages where software may dynamically modify the translations, the ATS protocol extensions enable the root complex to request invalidations of translations cached in the Device-IOTLB of an I/O device, and for the I/O devices to return responses indicating when an invalidation request is completed.

VT-d architecture supports ATS protocol extensions and enables software to control (through the device-mapping structures) if an I/O device can issue these transactions. For DMA requests indicating translated addresses from allowed devices, VT-d hardware bypasses the DMA-address translation.

I/O devices may implement Device-IOTLBs and support these protocol extensions to minimize performance dependencies on the DMA-remapping caching resources in the platform. However, to preserve the security, isolation, and reliability benefits of DMA remapping, device implementations must ensure that only translation responses from the root complex cause entries to be inserted into the Device IOTLB.

Handling Remapping Errors

Any errors or permission violations detected as part of remapping a DMA request are treated as a remapping fault. Unlike CPU page faults, which are restart-able at instruction boundaries, DMA-remapping faults are not restart-able due to the posted nature of PCI transactions. Any DMA write request that generates a fault is blocked by the remapping hardware, and the DMA read requests return an error to the device in the read response. Hardware logs detail DMA requests that cause remapping faults and use a fault event (interrupt) to inform software about such faults. For devices that explicitly request translations, an error detected while processing the translation request is not treated as a DMA-remapping fault, but is merely conveyed to the device in the translation response. This enables such devices to support device-specific demand page faulting. Demand page faulting is beneficial for devices (such as graphics adapters) with large DMA footprints, enabling software to demand pin the DMA buffers.

FUTURE HARDWARE SUPPORT

While VT-d enables the direct assignment of devices to guest VMs, it does not directly facilitate the efficient sharing of devices across multiple guest VMs. Such efficient sharing is not feasible without fundamental changes in the way that devices present their resources to the platform. Further work is being done in the PCI-SIG [21] [22] to enhance the PCI Express* specifications to enable devices to be shared.

Briefly, these extensions enable PCI Express devices to support multiple virtual functions, each of which can be discovered, configured, and managed. This allows the direct assignment of a virtual function to a VM using VT-d, thus allowing a single physical device to be sharable among multiple VMs.

The importance and applicability of these sharable PCI Express devices may be largely dependent upon the performance requirements, usage model, and market segment in which they may be deployed.

CONCLUSION

The virtualization of I/O resources is an important step toward enabling a significant set of emerging usage models in the data center, the enterprise, and the home. VT-d support on Intel platforms provides the capability to ensure improved isolation of I/O resources for greater reliability, security, and availability.

Specifically, VT-d supports the remapping of I/O DMA transfers and device-generated interrupts. The architecture of VT-d provides the flexibility to support multiple usage models that may run un-modified, special-purpose, or "virtualization aware" guest OSs. The VT-d hardware capabilities for I/O virtualization complement the existing Intel VT capability to virtualize processor and memory resources. Together, this roadmap of VT technologies offers a complete solution to provide full hardware support for the virtualization of Intel platforms.

Ongoing and future developments within the virtualization hardware and software communities will build upon VT-d to ensure that the requirements for sharing, security, performance, and scalability are being met. I/O devices will become more aware of the existence of VT-d to ensure efficient caching and consistency mechanisms to enhance their performance. Given the protection provided by VT-d, future I/O devices will emerge that are sharable among multiple guest OSs. With VT-d, software developers can develop and evolve their architectures that provide fully protected sharing of I/O resources that are highly available, provide high performance, and scale to increasing I/O demands.

REFERENCES

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

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

[3] Sugarman, J., Venkitachalam, G., Lim, B., "Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor," in *Proceedings of 2002 USENIX Annual Technical Conference*, pp. 1–14, June 2001.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pp. 164–177, October 2003.

[5] T. Garfinkel, B. Pfaff, J. Chow, M., Rosenblum, and D. Boneh, "Terra: A virtual machine-based platform for trusted computing," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pp. 193–206, 2003.

[6] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williams, "Safe hardware access with the Xen virtual machine monitor," in *Proceedings of the First Workshop on Operating System and Architectural Support for the on-demand IT Infrastructure (OASIS-2004)*, October 2004.

[7] M. Kieffer, "Windows Virtualization Architecture," *WinHEC 2005*, at http://download.microsoft.com/download/9/8/f/98f3fe47-dfc3-4e74-92a3-088782200fe7/TWAR05013_WinHEC05.ppt*.

[8] A. Whitaker, M. Shaw, and S. Gribble, "Scale and Performance in the Denali Isolation Kernel," in *System Design and Implementation (OSDI),* Boston, MA, December 2002.

[9] P. England, B. Lampson, J. Manferdelli, M. Peinado, B. Willman, "A Trusted Open Platform," *IEEE Computer*, pp. 55–62, July 2003.

[10] R. Goldberg, "Survey of Virtual Machine Research," *IEEE Computer*, pp. 34–45, June 1974.

[11] R. Creasy, "The Origin of the VM/370 Time-Sharing System," *IBM Journal of Research and Development*, pp. 483–490, September 1981.

[12] Intel Corp., "Intel Virtualization Technology Specification for Directed I/O Specification," at www.intel.com/technology/vt/.

[13] 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*.

[14] Dave Dunning, Greg Regnier, Don Cameron, Gary McAlpine, et al., "The Virtual Interface Architecture," *IEEE Micro, Volume 18, Issue 2*, pp. 66–76, March-April 1998.

[15] Ioannis Schoinas and Mark D. Hill, "Address Translation Mechanisms in Network Interfaces," in *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, February 1998.

[16] InfiniBand Trade Association, http://www.infinibandta.org/specs*.

[17] Grant Grundler, "Porting Drivers to HP ZX1," in *Proceedings of the Ottawa Linux Symposium,* June 2002.

[18] Intel Corp., AGP V3.0 Interface Specification.

[19] RDMA Consortium, http://www.rdmaconsortium.org*

[20] PCI Express Base Specification 1.1, http://www.pcisig.com/specifications/pciexpress/base*

[21] http://www.pcisig.com*

[22] PCI Express Address Translation Services and I/O Virtualization, WinHEC 2006, http://www.microsoft.com/whdc/winhec/pres06.mspx*

AUTHORS BIOGRAPHIES

Darren Abramson is a principal engineer in Intel's Chipset Group. He received his B.S. degree from the University of Kansas. Darren joined Intel in 1991 and has worked primarily on client chipset development and architecture, bringing to market I/O initiatives such as PCI, USB, and more recently PCI Express. His e-mail is darren.abramson at intel.com.

Jeff Jackson is a senior architect in Intel's Corporate Technology Group. Recently his areas of interest have been around networking-related technologies in virtualized environments. Jeff received an M.S. degree in Computer Science from Purdue University in 1994. His e-mail is jeff.jackson at intel.com.

Sridhar Muthrasanallur is a senior I/O architect in Intel's Digital Enterprise Group. He has eight years of experience in architecting I/O solutions for the Intel® Server Chipsets. His e-mail is sridhar.muthrasanallur at intel.com.

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.

Greg Regnier is a principal engineer in Intel's Corporate Technology Group. He joined Intel in 1988 and his experiences include massively parallel supercomputers, cluster communications, and high-performance network architecture. Regnier has a B.S. degree in Computer Science from St. Cloud State University in Minnesota. His e-mail is greg.j.regnier at intel.com.

Rajesh Sankaran is a principal engineer in Intel's Corporate Technology Group and is involved in development of CPU and I/O virtualization architecture. He received his M.S. degree in Electrical Engineering from Portland State University. His e-mail is rajesh.sankaran at intel.com.

Ioannis (Yannis) Schoinas is a principal engineer in Intel's Corporate Technology Group. He received his B.S. and M.S. degrees from the University of Crete-Heraclion and his Ph.D. degree from the University of Wisconsin-Madison. Yannis joined Intel's Server Architecture Lab in 1998 and worked on coherence protocols for the i870 chipset and future Intel® platforms. He also worked on a wide range of platform architecture topics including memory RAS, system partitioning, configuration management, system security and VT-d architecture. He is currently focusing on Tera-Scale Computing architecture challenges. His e-mail is ioannis.t.schoinas at intel.com.

Rich Uhlig is a senior principal engineer in Intel's Corporate Technology Group and leads various aspects of Intel's Virtualization Technology program, including architecture definition, research prototyping, performance analysis and characterization of VMM software usage. Rich received a Ph.D. degree in Computer Science and Engineering from the University of Michigan in 1995.

Balaji Vembu is a client ingredient architect in Intel's DEG Architecture and Planning group. He received his Bachelor's degree in EE from Regional Engg College, Bhopal, India. He received his M.S. degree in Computer Science from the University of California, Santa Barbara. He joined Intel in 1993 and worked on graphics and video acceleration in the chipset group. He is currently focused on virtualization and security architecture definition for client platforms. His e-mail is balaji.vembu at intel.com.

John Wiegert is a senior software engineer in Intel's Corporate Technology Group. John received his B.S. degree in Computer Science from the Rochester Institute of Technology. His current research interests involve I/O virtualization. His e-mail is john.a.wiegert at intel.com.

For further information visit:

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