Hardware Support for Efficient Virtualization

John Fisher-Ogden
University of California, San Diego

Abstract

Virtual machines have been used since the 1960's in creative ways. From multiplexing expensive mainframes to providing backwards compatibility for customers migrating to new hardware, virtualization has allowed users to maximize their usage of limited hardware resources. Despite virtual machines falling by the way-side in the 1980's with the rise of the minicomputer, we are now seeing a revival of virtualization with virtual machines being used for security, isolation, and testing among others.

With so many creative uses for virtualization, ensuring high performance for applications running in a virtual machine becomes critical. In this paper, we survey current research towards this end, focusing on the hardware support which enables efficient virtualization. Both Intel and AMD have incorporated explicit support for virtualization into their CPU designs. While this can simplify the design of a stand alone virtual machine monitor (VMM), techniques such as *paravirtualization* and hosted VMM's are still quite effective in supporting virtual machines.

We compare and contrast current approaches to efficient virtualization, drawing parallels to techniques developed by IBM over thirty years ago. In addition to virtualizing the CPU, we also examine techniques focused on virtualizing I/O and the memory management unit (MMU). Where relevant, we identify shortcomings in current research and provide our own thoughts on the future direction of the virtualization field.

1 Introduction

The current virtualization renaissance has spurred exciting new research with virtual machines on both the software and the hardware side. Both Intel and AMD have incorporated explicit support for virtualization into their CPU designs. While this can simplify the design of a stand alone virtual machine monitor (VMM), techniques such as *paravirtualization* and hosted VMM's are still quite effective in supporting virtual machines.

This revival in virtual machine usage is driven by many motivating factors. Untrusted applications can be safely sandboxed in a virtual machine providing added security and reliability to a system. Data and performance isolation can be provided through virtualization as well. Security, reliability, and isolation are all critical components for data centers trying to maximize the usage of their hardware resources by coalescing multiple servers to run on a single physical server. Virtual machines can further increase reliability and robustness by supporting live migration from one server to another upon hardware failure.

Software developers can also take advantage of virtual machines in many ways. Writing code that is portable across multiple architectures requires extensive testing on each target platform. Rather than maintaining multiple physical machines for each platform, testing can be done within a virtual machine for each platform, all from a single workstation. Virtualization can also be exploited for debugging purposes. Post-mortem forensics of a crashed or compromised server can be expedited if the server was running in a virtual machine [9]. Virtualization can also be used to support techniques such as bidirectional debugging [12] which aid both software developers and system administrators.

One final factor in the revival of virtual machines is they can provide simplified application deployment by packaging an *entire* environment together to avoid complications with dependencies and versioning.

With so many creative uses for virtualization, ensuring high performance for applications running in a virtual machine becomes critical. In this paper, we survey current research towards this end, focusing on the hardware support which enables efficient virtualization.

We compare and contrast current approaches to efficient virtualization, drawing parallels to techniques developed by IBM over thirty years ago. In addition to virtualizing the CPU, we also examine techniques focused on virtualizing I/O and the memory management unit (MMU). Where relevant, we identify shortcomings in current research and provide our own thoughts on the future direction of the virtualization field.

In the remainder of this paper, we present and evaluate multiple techniques aimed at providing efficient virtualization. In Section 2, we provide some historical background to put current Intel and AMD proposals in context. Section 3 then details the current approach from Intel. We next turn to the virtualization of the MMU in Section 4 and I/O in Section 5. Finally, Section 6 provides some discussion and comparisons before considering future directions for this field. Section 7 concludes our analysis.

2 Background

In this section, we will highlight relevant approaches to virtualization from the past few decades before discussing the current techniques from Intel and AMD.

2.1 Classical Virtualization

Popek and Goldberg's 1974 paper define requirements for what is termed *classical virtualization* [15]. By their standards, a piece of software can be considered a VMM if it meets the following three requirements:

- *Equivalent execution*. Programs running in a virtual environment run identically to running natively, barring differences in resource availability and timing.
- *Performance*. A "statistically dominant" subset of instructions must be executed directly on the CPU.
- *Safety*. A VMM must completely control system resources.

An early technique for virtualization was *trap and emulate*. While this approach was effective at providing an equivalent execution environment, its performance was severely lacking as each instruction could require tens of native instructions to emulate. The performance requirement for a VMM does not rule out trap and emulate, but rather, limits its application.

Popek and Goldberg also define *sensitive* instructions which can violate the safety and encapsulation that a VMM provides. For example, an instruction which changes the amount of system resources available would be considered sensitive. A VMM can be constructed for an architecture if the sensitive instructions are a subset of the privileged instructions. This ensures that the VMM can step in on all sensitive instructions and handle them safely since they are guaranteed to trap.

However, even if an architecture fails this, as the x86 architecture does, software techniques can be employed to achieve a similar execution environment despite not being *classically virtualizable*.

2.2 IBM Virtualizable Architectures

Now that we have established a baseline for virtualizable architectures, we examine a few IBM systems which pioneered the field of virtualization.

2.2.1 VM/370

The Virtual Machine Facility/370 (VM/370) [8] provides multiple virtual machines to users, each having the same architecture as the underlying IBM System/370 hardware they run on.

The VM/370 is comprised of three distinct modules: the Control Program (CP), Conversational Monitor System (CMS), and Remote Spooling and Communications Subsystem (RSCS). The Control Program handles the duties of a VMM and creates virtual machines, while CMS is the guest operating system which runs in each virtual machine. CMS was originally written for the IBM System/360 and transitioned to the virtual environment once CP came on-line. The final module of VM/370, RSCS, handles the networking and communication between virtual machines and also remote workstations.

A major goal of IBM was maintaining compatibility across a *family* of computers. While the VM/370 ran on the System/370 and exported that architecture through its virtual machines, programs written for the System/360 could still be run with degraded performance, despite the underlying architecture not supporting certain features.

An important design goal for CP and CMS was to make the virtual machine environment appear identical to its native counterpart. However, IBM did not make efficiency and performance an explicit design goal. While efficiency was not eschewed outright, these pioneering efforts rightly focused on functionality and correctness.

When running multiple guests in virtual machines, each guest believes that all of memory is at its disposal. Since a VMM must provide an *equivalent* environment for guests, dynamic address translation (DAT) must be performed to translate guest physical addresses to host physical addresses. VM/370 uses *shadow page tables* to achieve this translation.

Shadow page tables are a fairly simple mechanism for providing DAT but have been used quite heavily over the years. A guest OS manages its own page tables to map guest virtual addresses to guest physical addresses. Since guest physical addresses are actually *host virtual* addresses, the VMM must then use its own page tables to map to a host physical address. Once a host physical address is obtained, a mapping from guest virtual address to host physical address can be inserted into the hardware translation lookaside buffers (TLB).

2.2.2 370-XA

The System/370 Extended Architecture (370-XA) [11] continues the evolution of virtual machines beyond the VM/370. Given that performance was not an explicit goal for the VM/370, the 370-XA was able to increase the efficiency of virtualization in a variety of ways.

Since the trap and emulate technique was used so heavily, the 370-XA incorporated μ-code extensions called *assists* to the CPU to replace common functions that were expensive to emulate. As not all the available assists were targeted at virtualization support, we restrict our discussion to the assists that did target virtualization.

In previous systems, assists had proved themselves to be quite indispensable for running virtual machines. This caused the 370-XA to coalesce a large number of these assists into a new execution mode for the CPU, *interpretive execution*, which recognizes special instructions and enables most privileged instructions to execute directly in the virtual environment.

To enter interpretive execution mode, the privileged SIE instruction is used (Start Interpretive Execution). The operand given to SIE is the *state description* which describes the current state of the guest. Upon exiting interpretive execution, the state description is updated, including the guest program status word (PSW). The state description also details the reason for the exit to expedite any necessary handling by the host program.

Potential causes for exiting interpretive execution include interrupts, exceptions, instructions that require simulation, or even any instruction that the host program chooses via a mask.

Interpretive execution on the 370-XA can provide virtual environments for both the System/370 and the 370-XA architectures. However, the 370-XA does not use shadow page tables like VM/370. Since the 370-XA supports a larger 2GB address space, there were concerns over a possible sparseness of address references leading to a poor TLB cache hit rate. Maintaining the shadow page tables can be costly as well.

To avoid these issues, the 370-XA performs both levels of translation in hardware rather than relying on the shadow page tables to map guest physical addresses to host physical addresses. In Section 4, we see that both Intel and AMD have adopted similar approaches.

While guests can execute many privileged instructions in interpretive execution, guest I/O instructions do cause a trap to the VMM. The 370-XA does support a checking mode on a sub-channel basis that limits references to guest's storage only. This checking mode provides some protection against malicious or buggy guests

However, the 370-XA *preferred-machine assist* allows trusted guests to run directly in the host address space to avoid the overhead of an extra level of translation. These trusted guests can execute most privileged instructions, including those for I/O. Guests also handle their own interrupts in this mode, reducing the need to trap to the VMM.

On a final note, the 370-XA supports segment protection for limiting access among guests for isolation and security. This is not an assist per se, but rather an extension of the base architecture.

2.2.3 VM/ESA

Building upon the 370-XA, the Virtual Machine/Enterprise Systems Architecture (VM/ESA) [14] also uses interpretive execution to efficiently support virtual machine guests. While the 370-XA supported two architectures as virtual environments, the VM/ESA supports *five* different architecture modes: System/370, ESA/390, VM Data Spaces mode, 370-XA, and ESA/370, with the latter two being architectural subsets of ESA/390.

The VM Data Spaces mode enables memory sharing communication among guests that do not use DAT and also removes the 2 GB address space limit. While supporting five environments for virtual machines may seem unnecessary with today's personal workstations, one must remember that the VM/ESA was designed to run on large mainframes for *enterprises*. Providing compatibility during migration to a newer platfrom as well as enabling testing of the new platform was critical to IBM's business since the hardware was quite expensive.

Like the 370-XA, the VM/ESA also supports preferred storage mode via the preferred-machine assist. The 370-XA could only support a single guest in this mode since the guest did not use paging. However, the VM/ESA includes Multiple Domain Facility (MDF) which adds *zones* to support multiple guests in preferred mode. A guest is assigned a contiguous block of host storage with a register set to the base of this block and another register with the size of the block. The VM/ESA can then support multiple preferred guests each in its own zone, using single register translation to efficiently map between a guest physical address and a host physical address.

The dominant reason for guests to run in preferred storage mode is to achieve high performance I/O without the need to perform multiple levels of address translation. The single register translation maintains the performance gains while enabling multiple preferred guests.

The VM/ESA does support running VM/ESA as a guest of itself, "Russian doll" style. Interpreted SIE enables another instance of interpretive execution when already interpretively executing, distinguishing between "virtual" guests and "real" guests. However, not all hardware models support interpreted SIE. In that case, interpreted SIE can be simulated through shadow page tables and other shadow structures in the "real" guest. Zone relocation replaces the lowest level of dynamic address translation to reduce the performance premium for running nested virtual machines.

To conclude our discussion of VM/ESA, we note that the hardware TLBs are not tagged and must be flushed when switching between guests.

The VM/370, 370-XA, and VM/ESA illustrate the progression of virtualization techniques, with increasing amounts of functionality and performance as the systems matured. Many ground-breaking ideas were formulated in these systems, and we can clearly see their influence on the current virtualization offerings from Intel and AMD.

2.3 x86 Virtualization

We now step forward in time and consider the widely used x86 architecture. Due to the rise of personal workstations and decline of mainframe computers, virtual machines were considered nothing more than an interesting footnote in the history of computing. Because of this, the x86 was designed without much consideration for virtualization. Thus, it is unsurprising that the x86 fails to meet Popek and Goldberg's requirements for being *classically* virtualizable.

However, techniques were developed to circumvent the shortcomings in x86 virtualization. We first present a few of the architectural challenges inherent in the x86 before discussing various solutions to these challenges.

2.3.1 Architectural Challenges

The x86 architecture supports 4 privilege levels, or *rings*, with ring 0 being the most privileged and ring 3 the least. Operating systems run in ring 0, user applications run in ring 3, and rings 1 and 2 are not typically used.

Ring Compression

To provide isolation among virtual machines, the VMM runs in ring 0 and the virtual machines run either in ring 1 (the 0/1/3 model) or ring 3 (the 0/3/3 model). While the 0/1/3 model is simpler, it can not be used when running in 64 bit mode on a CPU that supports the 64 bit extensions to the x86 architecture (AMD64 and EM64T).

To protect the VMM from guest OSes, either paging or segment limits can be used. However, segment limits are not supported in 64 bit mode and paging on the x86 does not distinguish between rings 0, 1, and 2. This results in *ring compression*, where a guest OS must run in ring 3, unprotected from user applications.

Ring Aliasing

A related problem is *ring aliasing* where the true privilege level of a guest OS is exposed, contrary to the guest's belief that it is running in ring 0. For example, executing a PUSH instruction on the CS register, which includes the current privilege level, and then subsequently examining the results would reveal the privilege discrepancy.

Address Space Compression

Address space compression provides another hurdle for virtualizing the x86 architecture. The VMM can either run its own address space which can be costly when switching between guests and the VMM, or it can run in part of the guest's address space. When the VMM runs in its own address space, some storage in the guest address space is still required for control structures like the interrupt-descriptor table (IDT) and the global-descriptor table (GDT) [13]. In either case, the VMM must protect the portions of the address space it uses from the guest. Otherwise, a guest could discover its running in a virtual machine or compromise the virtual machine's isolation by reading or writing those locations.

Non-Privileged Sensitive Instructions

Next, in clear violation of "classical" virtualization, the x86 supports sensitive instructions that are not privileged and therefore do not trap to the VMM for correct handling. For example, the SMSW instruction stores the machine status word in a register which can then be read by the guest [16], exposing privileged information.

Silent Privilege Failures

Another problem involving privileged state is that some privileged accesses, rather than trapping to the VMM, fail *silently* without faulting. This violates Popek and Goldberg's tenet that guest virtual machines must execute identically to native execution barring solely timing and resource availability.

Interrupt Virtualization

Finally, *interrupt virtualization* can be a challenge for x86 virtual machines. The VMM wants to manage external interrupt masking and unmasking itself to maintain control of the system. However, some guest OSes frequently mask and unmask interrupts, which would result in poor performance if a switch to the VMM was required on each masking instruction.

We have briefly presented some of the challenges to virtualization on the x86 architecture. We refer interested readers to Robin and Irvine's analysis [16] for a more thorough presentation.

2.3.2 Binary Translation

While emulation can provide transparency and compatibility for guest virtual machines, its performance can be poor. One technique to improve virtualization performance is *binary translation*.

Binary translation involves rewriting the instructions of an application and inserting traps before problem sections or converting instructions to an entirely different instruction set architecture (ISA). Binary translation can be done *statically* or *dynamically*. Dynamic binary translation is used in just-in-time compilation (JIT), for example when executing bytecode on a Java Virtual Machine (JVM).

Many of the x86 architectural challenges outlined previously can be solved by simply inserting a trap instruction that enables the VMM to gain control and correctly emulate any problematic instructions.

Static binary translation can have difficulty analyzing a binary to reconstruct basic block information and a control flow graph. Dynamic translation avoids this because it can translate instructions as needed. However, the online translation must be done quickly to maintain acceptable levels of performance.

A novel example of binary translation is the FX!32 profile-directed binary translator from DEC [7]. FX!32 emulates an application on its first run while profiling the application to determine the instructions that would most benefit from running natively. These instructions are then translated so the next time the application is run, its performance improves dramatically.

While FX!32 is a solution to running x86 applications on DEC's Alpha architecture, its hybrid approach combining emulation and dynamic binary translation illustrates an effective solution to executing unmodified binaries transparently, without sacrificing performance.

2.3.3 Paravirtualization

Binary translation enables virtualization when recompiling source code is not desirable or feasible. *Paravirtualization* eschews this restriction in the name of high performance virtual machines.

Rather than presenting an equivalent virtual environment to guests, paravirtualization exposes virtual machine information to guest operating systems, enabling the guests to make more informed decisions on things like page replacement. In addition, source-level modifications can be made to avoid the x86 challenges to virtualization. Whereas binary translation would trap on problematic instructions, paravirtualization can avoid the instructions entirely.

A leading paravirtualization system is Xen [6]. Xen achieves high performance for guest virtual machines while retaining the benefits of virtualization—resource utilization, isolation, etc.

Of course, Xen must sacrifice binary compatibility for guest operating systems. While one can easily recompile a Linux OS to run on Xen, the same can not be said for Microsoft's Windows OSes.

2.4 Co-designed Virtual Machines

While software tricks can often be played to support virtualization on an uncooperative architecture, an alternative is to design the architecture and VMM in tandem. These *co-designed* virtual machines blur the strict ISA boundary into a virtual ISA that enables increased communication between hardware and software.

For example, software can track the phases of an application and tune the branch prediction logic in the hardware to optimize for the current application phase.

While this technique has only seen limited use, the best example is Transmeta's Crusoe processor. The Crusoe externally supports an x86 ISA while internally using a very long instruction word (VLIW) architecture for power efficiency [17].

3 Current Approaches

Virtualization on the x86 architecture has required unnecessary complexity due to its inherent lack of support for virtual machines. However, extensions to the x86 remedy this problem and as a result, can support a much simpler VMM. Further, the extensions succeed in making the x86 architecture classically virtualizable.

Both leading chip manufacturers, Intel and AMD, have rolled out these virtualization extensions in current processors. Intel calls its virtualization technology VT-x, previously codenamed Vanderpool. AMD's extensions go by the name AMD-V, previously Secure Virtual Machine (SVM) and codenamed Pacifica.

While Intel VT-x and AMD-V are not entirely equivalent, they share the same basic structure. Therefore, we focus our discussion on Intel's offering, noting significant departures for AMD in Section 6.2.

3.1 Intel VT-x

Intel VT-x introduces new modes of CPU operation: VMX root operation and VMX non-root operation[13]. One can think of VMX root operation being similar to previous IA-32 operation before VT-x and is intended for VMMs ("host" mode), while VMX non-root operation is essentially a *guest* mode targeted at virtual machines. Both operating modes support execution in all four privilege rings.

The `VMRUN` instruction performs a VM entry, transferring from host to guest mode. Control transfers back to host mode on a VM exit which can be triggered by both conditional and unconditional events. For example, the `INVD` instruction unconditionally triggers a VM exit while a write to a register or memory location might depend on which bits are being modified.

Critical to the interaction between hosts and guests is the virtual machine control structure (VMCS) which contains both guest state and host state. On VM entry, the guest processor state is loaded from the VMCS after storing the host processor state. VM exit swaps these operations, saving the guest state and loading the host state.

The processor state includes segment registers, the CR3 register, and the interrupt descriptor table register (IDTR). The CR3 register (control register 3) holds the physical location of the page tables. By loading and storing this register on VM entry and exit, guest virtual machines can run in an entirely separate address space than the VMM.

However, the VMCS does *not* contain any general purpose registers as the VMM can do this as needed. This improves VM entry and VM exit performance. On a related note, a guest's VMCS is referenced with a physical address to avoid first translating a guest virtual address.

As alluded to above, the biggest difference between host and guest mode (VMX root and non-root operation) is that many instructions in guest mode will trigger a VM exit. The *VM-execution control fields* set the conditions for triggering a VM exit.

The control fields include:

- *External-interrupt exiting*. Sets whether external interrupts causes VM exits, regardless of guest interrupt masking.
- *Interrupt-window exiting*. Causes a VM exit when guest interrupts are unmasked.
- *Use TPR shadow*. Accesses to the task priority register (TPR) through register CR8 (64-bit mode only) can be set to use a shadow TPR register, available in the VMCS. This avoids a VM exit in the common case.
- *CR masks and shadows*. Bit masks for each control register enable guest modification of select bits while transferring to host mode on writes to other bits. Similar to the TPR register, the VMCS also includes shadow registers which a guest can freely read.

While the register masks provide fine-grained control over specific control registers, the VMCS also includes several bitmaps that provide added flexibility.

- *Exception bitmap*. Selects which exceptions cause a VM exit. Page faults can be further differentiated based on the fault's error code.
- *I/O bitmap*. Configures which ports in the 16-bit I/O port space cause VM exits when accessed.
- *MSR bitmaps*. Similar to CR bit masks, each model specific register (MSR) has a read bitmap and a write bitmap to control accesses.

With all of these possible events causing a VM exit, it becomes important for a VMM to quickly identify the problem and correct it so control can return to the guest virtual machine. To facilitate this, a VM exit also includes details on the reasons for the exit to aid the VMM in handling it.

While the VMM responds to events from a guest, this becomes a two-way communication channel with *event injection*. Event injection allows the VMM to introduce interrupts or exceptions to a guest using the IDT.

3.1.1 Architectural Challenges Addressed

In Section 2.3.1, we outlined several architectural challenges inherent in the x86 which created barriers to virtualization. Now that we have examined VT-x in more detail, we see that VT-x does in fact provide solutions to each challenge.

By introducing a new mode of execution with full access to all four privilege rings, both the ring compression and ring aliasing problems disappear. A guest OS executes in ring 0 while the VMM is still fully protected from any errant behavior.

Since each guest VMCS is referenced with a physical address and the VMCS stores the critical IDTR and CR3 registers, virtual machines have full access to their entire address space, eliminating the problem of address space compression.

The x86 contains both non-privileged sensitive instructions and privileged instructions that fail silently. However, given VT-x's extensive flexibility for triggering VM exits, fine-grained control over any potentially problematic instruction is available.

Lastly, the VMCS control fields also address the challenge of interrupt virtualization. External interrupts can be set to always cause a VM exit, and VM exits can be conditionally triggered upon guest masking and unmasking of interrupts.

With these solutions to the x86 virtualization challenges, the x86 can finally be termed *classically* virtualizable. With VT-x, the VMM can be much simpler compared to the previous techniques of paravirtualization and binary translation. A simpler VMM leaves less room for error and can provide a more secure virtual environment for guest virtual machines.

3.1.2 Performance

Intel VT-x provides the hardware support enabling a simpler VMM. However, simplicity and performance are often competing goals.

Adams and Agesen demonstrate that software techniques for virtualization, e.g. paravirtualization and binary translation, outperform a hardware-based VMM leveraging Intel's VT-x [4]. They experiment with several macro- and micro-benchmarks, as well as so-called "nanobenchmarks" which exercise individual virtualization-sensitive instructions.

The hardware VMM performs better in some of the experiments, but overall the software VMM provides a better high-performance virtualization solution. Reasons for this performance discrepancy include:

- *Maturity*. Hardware assisted virtualization on the x86 is still an emerging technology while software techniques have been around long enough to mature.
- *Page faults*. Maintaining integrity of shadow page tables can be expensive and cause many VM exits.
- *Statelessness*. VMM must reconstruct the cause for a VM exit from the VMCS.

Of these barriers to high performance, only the last is inherent to a hardware VMM. Maturity will come in due time and both Intel and AMD have proposed solutions providing hardware MMU support for servicing page faults.

Similar to the VM/370, VT-x strives for correctness and functionality before aggressively optimizing. A VM entry required 2409 cycles on Intel's P4 microarchitecture, the first to support VT-x. However, the next generation Core microarchitecture reduces this to 937 cycles, a 61% reduction [4]. We expect to see further improvements to VT-x's performance as it becomes a more established technology.

4 MMU Virtualization

Given the performance degradation caused by handling page faults via shadow page tables, virtualizing the memory management unit becomes an important next step. Intel's extended page tables (EPT) and AMD's Nested page tables (NPT) are proposals for doing exactly this.

Rather than have the VMM maintain the integrity of the shadow page table mappings, EPT adds a separate set of hardware-walked page tables which map the guest physical addresses to host physical addresses. Shadow page tables were managed in software by the VMM but EPT and Nested Paging add hardware support to avoid costly VM entries and exits.

Another feature that Intel and AMD will include is tagged TLBs. Intel assigns virtual-processor identifiers (VPIDs) to each virtual machine and tags the translation entries in the TLB with the appropriate VPID. This avoids the performance hit of flushing the TLB on every VM entry and VM exit. Without a tagged TLB, flushing the translations is required to avoid incorrect mappings from virtual to physical addresses.

While the extended page tables and VPID tagged TLBs are not currently available, we expect them to further close the performance gap by eliminating major sources of virtualization overhead.

5 I/O Virtualization

Efficient I/O virtualization is an important consideration for many uses of virtual machines. Here, we present current approaches to handling I/O in a virtual environment. We then examine AMD's current DEV proposal before moving on to Intel's VT-d.

5.1 Current Methods

Before describing current techniques for handling input and output in virtual machines, we review three distinct classes of VMMs. Each class provides a unique approach to I/O and virtualization, illustrated in Figure 1.

- *Hosted VMM*. The VMM executes in an existing OS, utilizing the device drivers and system support provided by the OS [18].
- *Stand-Alone VMM*. A hypervisor runs directly on the hardware and incorporates its own drivers and system services [6].
- *Hybrid VMMs*. Combines the control of a stand-alone VMM with the simplicity of hosted VMM by running a deprivileged *service OS* as an additional guest and routing I/O requests through it.

Hardware support via device drivers is paramount to providing an effective VMM for virtualizing I/O. Re-implementing device drivers for a stand-alone VMM can be expensive and therefore, limits the portability of the VMM to a small set of supported hardware. However, using an additional operating system to handle I/O resources imposes additional performance overhead. Reducing this performance overhead is one of the goals of Intel's VT-d proposal.

Given these three styles of VMMs, we now consider existing techniques for virtualizing I/O that can be applied with varying effectiveness to each VMM class. These techniques can be concurrently employed by a VMM, e.g. paravirtualizing a high-performance network card while emulating a legacy disk controller.

Emulation

Device *emulation* is the most general technique and requires the implementation of real hardware completely in software. This creates a virtual device that the guest interacts with.

There is usually a different physical hardware device underneath that performs the actual I/O. The emulated virtual device serve as an adapter, converting an unsupported interface to a supported one.

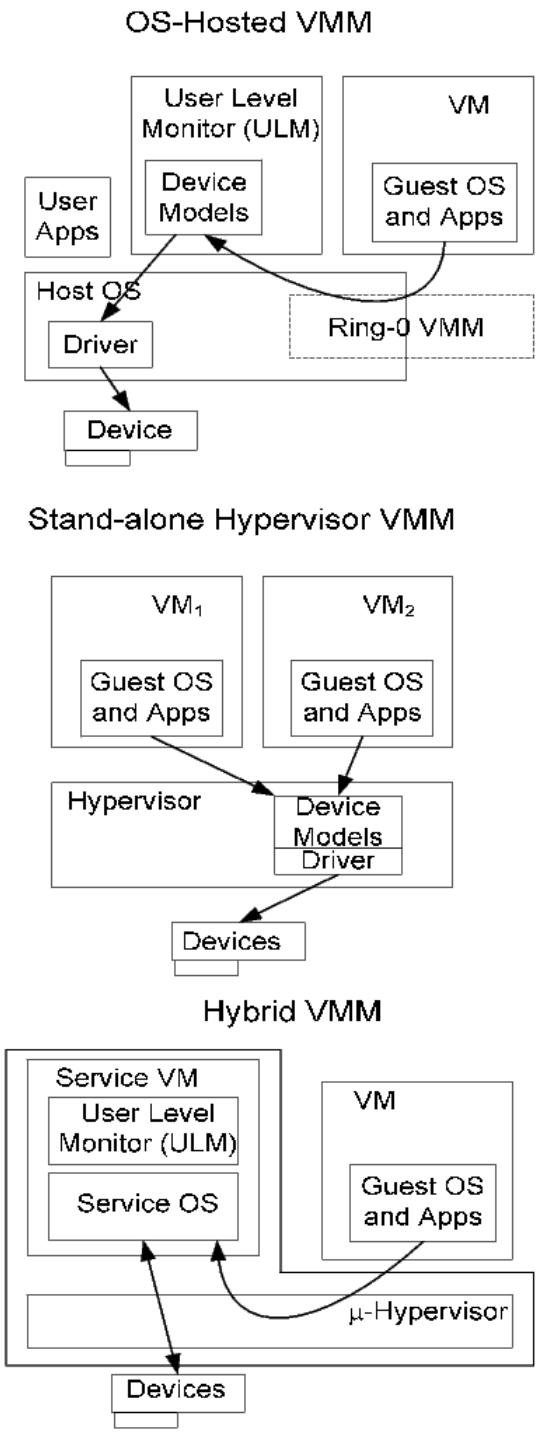

Figure 1: Current Methods for Virtualizing I/O. Credit [3]

In addition, a VMM must also be able to inject interrupts into the guest. This is often done via emulation of a programmable interrupt controller (PIC).

In a datacenter, migration of virtual machines is often important to maintain high availability in the face of unreliable hardware. Device emulation facilitates virtual machine migration since the virtual device state exists in memory and can be easily transferred. Further, the guest is not tied to a specific piece or version of hardware that might not be available on another machine.

Another consideration for virtualization is the ability to efficiently multiplex a device across multiple guest VMs. Device emulation simplifies sharing a physical device since the VMM can perform the multiplexing while presenting individual virtual devices to each guest.

While emulation does have the ease of migration and multiplexing advantages, it has the disadvantage of poor performance. Emulation requires the VMM to perform a significant amount of work to present the illusion of a virtual device. Further, when dealing with specific device driver binaries, "bug emulation" may be required to match the hardware expectations of the device driver.

Emulation can be effectively applied to all three VMM classes.

Paravirtualization

Rather than bending over backwards to match the expectations of a device driver or other guest software, *paravirtualization* modifies the guest software to cooperate directly with the VMM.

Of course, this is only possible when source-level modifications can be made and recompiled. Proprietary operating systems and device drivers can not be paravirtualized. This limits paravirtualization's applicability, but the performance boost achieved offers a worthwhile trade-off.

Paravirtualization uses an eventing or callback mechanism for better performance than an emulated PIC. However, performance comes at the cost of modifying a guest OS's interrupt handling mechanisms.

Also, necessary modifications for one guest OS might be entirely separate from modifications required for a different guest OS.

Similar to emulation, paravirtualization supports both VM migration and device sharing. VM migration is possible as the destination platform supports the same VMM APIs needed by the guest software stack [3].

While paravirtualization is usually applied to standalone VMMs, it is applicable to all classes.

Dedicated Devices

The final device virtualization technique we consider does not virtualize the device but rather assigns it directly to

a guest virtual machine. *Dedicated devices* utilize the guest's device drivers, ensuring full compatibility. They also simplify the VMM by removing much of the complexity required to securely and efficiently handle I/O requests.

Virtual devices can be easily replicated for additional guests but there are only limited physical resources that can be dedicated to guests. Further, directly assigning a device to a virtual machine can make the VM difficult to migrate, especially any hardware device state.

Dedicated devices eliminate most virtualization overhead and enable added simplicity in a VMM. However, the main disadvantage is that direct memory access (DMA) from hardware devices directly to a guest's virtual address is not currently supported due to isolation and address translation challenges.

Enabling DMA for this scenario is one of the key contributions of Intel's VT-d.

5.2 AMD DEV

Since Intel's VT-d provides further functionality beyond that of AMD's DEV, we first consider DEV, setting the stage for VT-d.

AMD's proposal for handling I/O virtualization adds a *device exclusion vector* (DEV) that permits or blocks DMA to specified memory pages. DEV is essentially a table that specifies access controls between devices and memory pages.

Since guests have exclusive access to memory pages, DEV can be used to provide exclusive access to a device by only permitting DMA between the device and the guest's address space.

The access check is made on the HyperTransport boundary at the CPU and either blocks or permits the DMA request. If access is permitted, the request is marked safe to avoid subsequent access control checks.

The end result of DEV is that it ensures that a dedicated hardware device writes only to the intended guest's assigned memory pages.

5.3 Intel VT-d

Intel's VT-d proposal surpasses AMD's DEV in terms of functionality. The main components of VT-d are DMA remapping and interrupt virtualization.

VT-d adds a generalized IOMMU architecture. Traditional IOMMU's have been used to efficiently support DMA scatter/gather operations and are implemented in PCI root bridges [3].

VT-d incorporates software specified protection domains which restrict access only to devices assigned to a domain. This provides similar isolation and exclusion as DEV. A protection domain is loosely defined as an "isolated environment to which a subset of the host physical memory is allocated."[3] This abstract definition enables protection domains to be defined for virtual machines as well as device drivers running in the VMM.

By using address translation tables, VT-d achieves the necessary DMA isolation, restricting access to physical memory only to assigned devices. DMA virtual addresses (DVA) are used in the translation tables. Depending on the software usage model, DVAs can be guest physical addresses, host linear addresses, or some other abstracted virtual I/O address.

VT-d also includes an IOTLB to cache address translation lookups, improving the performance further. The PCI Bus/device/function acts as the identifier for the DMA request. This ID is used when performing address translations to achieve the DMA remapping from DVA to physical memory.

Once the data has been transferred to memory, the appropriate guest must be notified. Previously, interrupts were routed through the VMM, adding additional overhead. VT-d adds interrupt virtualization support to boost performance.

Before VT-d, devices on the x86 could signal an interrupt using legacy I/O interrupt controllers or issue a message signaled interrupt (MSI) via DMA to a predefined address range. Current device issued MSIs encode interrupt attributes in the DMA requests, violating the isolation requirement across protection domains [3].

Intel's VT-d redefines the interrupt-message format for MSIs, providing the necessary isolation. The VT-d DMA writes contain only a message identifier and the hardware device's requester id. These ids are then used to index into a interrupt-remapping table which contains the expected interrupt attributes.

This interrupt-remapping table is validated by the VMM and opaque message ids are given to devices to provide the protection domain isolation.

We have simplified our discussion of VT-d for clarity, but VT-d also includes additional mapping structures, some of which are nested hierarchically. Repeatedly traversing these tables can become expensive. To avoid this in the common case, VT-d adds various hardware managed caching structures, including the abovementioned IOTLB and an interrupt entry cache, to improve performance.

5.4 Future Directions

VT-d enables virtual machines to utilize DMA with dedicated devices in a protected, efficient manner. However, VT-d does not support high-performance multiplexing of devices.

Intel has done much of the necessary work to achieve

efficient device sharing, and now, the device makers themselves as well as the PCI Express Special Interest Group [2] (PCISIG) must join in the efforts. Towards this end, the PCISIG has proposed a few extensions to PCI Express which we briefly discuss here.

Since the IOTLB must handle multiple concurrent requests at times, finding the right size for the TLB structure can be difficult. One solution to this involves moving these cached translation entries from the IOTLB to the physical devices themselves. This proposal by the PCISIG is termed *address translation services* [1] (ATS).

Of course, ATS would have to guarantee that installed translations only come from a valid source and do not violate protection domains.

Another proposal enables PCI Express devices to support multiple *virtual functions*, each of which gives the illusion of an entirely separate physical device. This enables the direct assignment of virtual functions on a device to a guest virtual machine, while efficiently multiplexing the device across multiple guests. Combining this proposal with Intel VT-d, I/O virtualization and direct assignment of devices will become an attractive feature for guest virtual machines.

An interesting case study for this virtual functions proposal is InfiniBand. InfiniBand has actually supported virtual functions at the hardware level for a few years now [10]. Separate logical communication links, termed virtual lanes, share a single physical link. Each lane individually performs its own flow control and buffer management, isolated from the other lanes. InfiniBand supports up to 15 general purpose virtual lanes plus an additional lane for control traffic.

Whether the PCISIG can build upon InfiniBand's virtual lanes and provide a general solution for all PCI Express devices remains to be seen. We look forward to PCISIG's proposals coming to fruition.

6 Discussion

In this section, we look back to IBM and see the roots of Intel's VT-x and AMD-V. As we have focused on Intel for the most part, we discuss AMD's alternative offering. We then turn forward to future directions and applications of hardware-supported virtualization.

6.1 IBM Comparisons

Now that we have presented both IBM's and Intel's techniques for virtualization, we see direct correlations between IBM's virtualization approaches and Intel's VT-x.

The following table lists IBM's concepts on the left with Intel's adaptations on the right.

IBM vs. Intel VT-x	
Interpretive execution	VMX non-root mode
State description	VMCS
Shadow page tables	Shadow page tables
2-level page tables	Extended page tables

The interpretive execution mode pioneered by the 370-XA enabled guests to execute most privileged instructions directly. Intel targets this efficiency with its VMX non-root mode which lets guests execute privileged instructions in ring 0.

The VMCS conceptually represents the same thing as IBM's guest state descriptions, albeit augmented to match the specifics of the x86.

As discussed in Section 4, VT-x currently utilizes shadow page tables to virtualize the MMU and handle guest paging. The 370-XA avoided shadow page tables and performed both levels of translation using hardware walked page tables. Intel's EPT and AMD's Nested Paging proposals both incorporate the additional level of indirection into the hardware to avoid the costs of managing the shadow page tables.

Exits from IBM's interpretive execution were caused by interrupts, exceptions, instructions that required simulation, or any instruction chosen via a mask. VM exits from Intel's VMX non-root mode also can be similarly configured although the control fields in the VMCS provide finer-grained policies through bitmasks on registers, exceptions, and I/O ports.

We note that VM/ESA supports "Russian doll" virtualization with multiple levels of interpretive execution. VT-x would need to use paravirtualization or binary translation to achieve the same effect. Of course, the motivations for supporting this use case are not clear so it is understandable that VT-x eschews this functionality.

Lastly, VM/ESA did not support tagged TLBs and therefore, a flush was required when switching between virtual machines. Again, current Intel and AMD offerings suffer from the same performance degradation. However, Intel proposes adding a virtual processor ID (VPID) to differentiate TLB entries between multiple guest virtual machines.

While Intel and AMD have certainly drawn upon the pioneering work of IBM, they have expanded the techniques and adapted them accordingly to fit the x86 architecture. In particular, high performance I/O has become critical to virtualization in data centers. Intel's VT-d offers some interesting solutions to this challenge, moving one step closer to the complete virtualization of all aspects of a machine.

However, we note that we have only examined a small, albeit important, fraction of IBM's virtualization research. Techniques and ideas from Intel may not be as novel as we realize since IBM performed a large amount of virtualiza-

tion research in the 1970's.

6.2 AMD-V

AMD-V [5] is functionally quite similar to Intel's VT-x. However, the two competing approaches provide incompatible ISAs. When AMD first announced its codenamed Pacifica virtualization initiative, it included proposals addressing both the MMU and DMA. This gave AMD an edge in functionality over Intel, but Intel quickly released their own proposals with Intel's DMA remapping going beyond the functionality provided by AMD.

One reason that AMD announced their proposals first is that AMD processors contain the MMU on-chip as well as the HyperTransport communication bus. AMD could not ignore these components as easily as Intel.

In addition to shadow paging and nested paging, AMD-V also supports *paged real mode* which is similar to the preferred machine assist zones of the VM/ESA. Paged real mode enables the virtualization of guests that run in real mode, using only segments and offsets to specify physical addresses. Shadow paging is used here as well to provide the illusion of real mode for appropriate guests.

One terminology difference we note is that AMD terms its shared host-guest control structure the virtual machine control block (VMCB) as opposed to Intel's VMCS.

6.3 Future Directions

With VT-x and AMD-V, the x86 architecture has become classically virtualizable. Moving beyond this barrier, interest turned to efficiently handling page faults and the memory management unit. While optimizations will no doubt be made to existing virtualization components, the last challenge involves I/O and DMA.

Current I/O solutions leave much to be desired and the upcoming proposal from Intel still only lays the foundation for I/O virtualization. VT-d enables the direct assignment of hardware devices to guest virtual machines, but it does not facilitate *sharing* a device across multiple guests [3].

Quality of service and priority scheduling are very real considerations in a datacenter environment. Efficiently multiplexing a device across guests is only the first step. Fine-grained control over *policy* is critical as well. Achieving device multiplexing both securely and efficiently is certainly a motivating goal for virtualization research.

Another direction for virtualization is eliminating all performance overheads for virtual machines. Executing at native speeds is the gold standard of virtual machine benchmarking. It will be interesting to see if hardware techniques can surpass software virtualization, or if the optimal solution might be a hybrid approach.

There will always be some work that a VMM must handle for guests, but the key will be to amortize this overhead across as many guest instructions as possible.

Finally, it is important to consider the market forces driving virtualization. We have sketched many motivating scenarios where efficient virtualization is critical. However, to what extent VT-d and DEV influence the market remains to be seen. Of course, academic institutions can pioneer research in many areas, it still falls to Intel and AMD to put the technology in the hands of consumers.

7 Conclusion

We have critically examined hardware support for efficient virtualization. IBM pioneered this area with μ-code assists and interpretive execution as well as shadow page tables and two-level hardware walked page tables. Current techniques and proposals from Intel and AMD build upon IBM's foundation, adding tagged TLBs, DMA remapping, and finer-grained control. These additions represent an important step towards the true goal of achieving native execution speeds in a virtual machine on the x86 architecture.

Acknowledgments

We would like to thank Geoff Voelker for his invaluable input during discussions forming this work.

References

[1] PCI Express Address Translation Services and I/O Virtualization, WinHEC 2006, `http://www.microsoft.com/whdc/winhec/pred06.mspx`.

[2] PCI Special Interests Group. `http://www.pcisig.com`.

[3] Darren Abramson, Jeff Jackson, Sridhar Muthrasanallur, Gil Neiger, Greg Regnier, Rajesh Sankaran, Ioannis Schoinas, Rich Uhlig, Balaji Vembu, and John Weigert. Intel Virtualization Technology for directed I/O. *Intel Technology Journal*, 10(3):179–192, August 2006.

[4] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. *Operating Systems Review*, 40(5):2–13, December 2006.

[5] AMD. Amd64 virtualization codenamed "pacifica" technology: Secure virtual machine architecture reference manual, May 2005.

[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, and R. Neugebauer. Xen and the art of virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles*, October 2003.

[7] Anton Chernoff, Mark Herdeg, Ray Hookway, Chris Reeve, Norman Rubin, Tony Tye, S. Bharadwaj Yadavalli, and John Yates. FX!32: A profile-directed binary translator. *IEEE Micro*, 18(2):56–64, Mar/Apr 1998.

[8] Robert J. Creasy. The origin of the vm/370 time-sharing system. *IBM Journal of Research and Development*, 25(5):483–490, 1981.

[9] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. Revirt: enabling intrusion analysis through virtual-machine logging and replay. *SIGOPS Oper. Syst. Rev.*, 36(SI):211–224, 2002.

[10] Chris Eddington. Infinibridge: An infiniband channel adapter with integrated switch. *IEEE Micro*, 22(2):48–56, 2002.

[11] Peter H. Gum. System/370 extended architecture: Facilities for virtual machines. *IBM Journal of Research and Development*, 27(6):530–544, November 1983.

[12] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. *Proceedings of the 2005 Annual USENIX Technical Conference*, April 2005.

[13] Gil Neiger, Amy Santoni, Felix Leung, Dion Rodgers, and Rich Uhlig. Intel Virtualization Technology: Hardware support for efficient processor virtualization. *Intel Technology Journal*, 10(3):167–177, August 2006.

[14] D. L. Osisek, K. M. Jackson, and P. H. Gum. ESA/390 interpretive-execution architecture, foundation for VM/ESA. *IBM Systems Journal*, 30(1):34–51, 1991.

[15] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, July 1974.

[16] John Scott Robin and Cynthia E. Irvine. Analysis of the Intel Pentium's ability to support a secure virtual machine monitor. In USENIX, editor, *Proceedings of the Ninth USENIX Security Symposium, August 14–17, 2000, Denver, Colorado*, page 275, San Francisco, CA, USA, 2000. USENIX.

[17] J. Smith, S. Sastry, T. Heil, and T. Bezenek. Achieving high performance via co-designed virtual machines. In *IWIA '98: Proceedings of the 1998 International Workshop on Innovative Architecture*, page 77, Washington, DC, USA, 1998. IEEE Computer Society.

[18] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O devices on vmware workstation's hosted virtual machine monitor. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 1–14, Berkeley, CA, USA, 2001. USENIX Association.