Bringing Virtualization to the x86 Architecture with the Original VMware Workstation

EDOUARD BUGNION, Stanford University
SCOTT DEVINE, VMware Inc.
MENDEL ROSENBLUM, Stanford University
JEREMY SUGERMAN, Talaria Technologies, Inc.
EDWARD Y. WANG, Cumulus Networks, Inc.

This article describes the historical context, technical challenges, and main implementation techniques used by VMware Workstation to bring virtualization to the x86 architecture in 1999. Although virtual machine monitors (VMMs) had been around for decades, they were traditionally designed as part of monolithic, single-vendor architectures with explicit support for virtualization. In contrast, the x86 architecture lacked virtualization support, and the industry around it had disaggregated into an ecosystem, with different vendors controlling the computers, CPUs, peripherals, operating systems, and applications, none of them asking for virtualization. We chose to build our solution independently of these vendors.

As a result, VMware Workstation had to deal with new challenges associated with (i) the lack of virtualization support in the x86 architecture, (ii) the daunting complexity of the architecture itself, (iii) the need to support a broad combination of peripherals, and (iv) the need to offer a simple user experience within existing environments. These new challenges led us to a novel combination of well-known virtualization techniques, techniques from other domains, and new techniques.

VMware Workstation combined a hosted architecture with a VMM. The hosted architecture enabled a simple user experience and offered broad hardware compatibility. Rather than exposing I/O diversity to the virtual machines, VMware Workstation also relied on software emulation of I/O devices. The VMM combined a trap-and-emulate direct execution engine with a system-level dynamic binary translator to efficiently virtualize the x86 architecture and support most commodity operating systems. By relying on x86 hardware segmentation as a protection mechanism, the binary translator could execute translated code at near hardware speeds. The binary translator also relied on partial evaluation and adaptive retranslation to reduce the overall overheads of virtualization.

Written with the benefit of hindsight, this article shares the key lessons we learned from building the original system and from its later evolution.

Categories and Subject Descriptors: C.0 [**General**]: *Hardware/software interface, virtualization*; C.1.0 [**Processor Architectures**]: General; D.4.6 [**Operating Systems**]: Security and Protection; D.4.7 [**Operating Systems**]: Organization and Design

General Terms: Algorithms, Design, Experimentation

Additional Key Words and Phrases: Virtualization, virtual machine monitors, VMM, hypervisors, dynamic binary translation, x86

Together with Diane Greene, the authors co-founded VMware, Inc. in 1998.

Authors' addresses: E. Bugnion, School of Computer and Communication Sciences, EPFL, CH-1015 Lausanne, Switzerland; S. Devine, VMware, Inc., 3401 Hillview Avenue, Palo Alto, CA 94304; M. Rosenblum, Computer Science Department, Stanford University, Stanford, CA 94305; J. Sugerman, Talaria Technologies, Inc.; E. Y. Wang, Cumulus Networks, Inc.

DOI 10.1145/2382553.2382554 http://doi.acm.org/10.1145/2382553.2382554

ACM Reference Format:
Bugnion, E., Devine, S., Rosenblum, M., Sugerman, J., and Wang, E. Y. 2012. Bringing virtualization to the x86 architecture with the original VMware Workstation. ACM Trans. Comput. Syst. 30, 4, Article 12 (November 2012), 51 pages.
DOI = 10.1145/2382553.2382554 http://doi.acm.org/10.1145/2382553.2382554

1. INTRODUCTION

We started VMware in 1998 with the goal of bringing virtualization to the x86 architecture and the personal computer industry. VMware's first product—VMware Workstation—was the first virtualization solution available for 32-bit, x86-based platforms. The subsequent adoption of virtualization had a profound impact on the industry. In 2009, the ACM awarded the authors the *ACM Software System Award* for VMware Workstation 1.0 for Linux. Receiving that award prompted us to step back and revisit, with the benefit of hindsight, the technical challenges in bringing virtualization to the x86 architecture.[1]

The concept of using virtual machines was popular in the 1960s and 1970s in both the computing industry and academic research. In these early days of computing, virtual machine monitors (VMMs) allowed multiple users, each running their own single-user operating system instance, to share the same costly mainframe hardware [Goldberg 1974]. Virtual machines lost popularity with the increased sophistication of multi-user operating systems, the rapid drop in hardware cost, and the corresponding proliferation of computers. By the 1980s, the industry had lost interest in virtualization and new computer architectures developed in the 1980s and 1990s did not include the necessary architectural support for virtualization.

In our research work on system software for scalable multiprocessors, we discovered that using virtual machine monitors could solve, simply and elegantly, a number of hard system software problems by innovating in a layer below existing operating systems. The key observation from our Disco work [Bugnion et al. 1997] was that, while the high complexity of modern operating systems made innovation difficult, the relative simplicity of a virtual machine monitor and its position in the software stack provided a powerful foothold to address limitations of operating systems.

In starting VMware, our vision was that a virtualization layer could be useful on commodity platforms built from x86 CPUs and primarily running the Microsoft Windows operating systems (a.k.a. the WinTel platform). The benefits of virtualization could help address some of the known limitations of the WinTel platform, such as application interoperability, operating system migration, reliability, and security. In addition, virtualization could easily enable the co-existence of operating system alternatives, in particular Linux.

Although there existed decades' worth of research and commercial development of virtualization technology on mainframes, the x86 computing environment was sufficiently different that new approaches were necessary. Unlike the vertical integration of mainframes where the processor, platform, VMM, operating systems, and often the key applications were all developed by the same vendor as part of a single architecture [Creasy 1981], the x86 industry had a disaggregated structure. Different companies independently developed x86 processors, computers, operating systems, and applications. For the x86 platform, virtualization would need to be inserted without changing either the existing hardware or the existing software of the platform.

[1] In this article, the term *x86* refers to the 32-bit architecture and corresponding products from Intel and AMD that existed in that era. It specifically does not encompass the later extensions that provided 64-bit support (Intel IA32-E and AMD x86-64) or hardware support for virtualization (Intel VT-x and AMD-v).

As a result, VMware Workstation differed from classic virtual machine monitors that were designed as part of monolithic, single-vendor architectures with explicit support for virtualization. Instead, VMware Workstation was designed for the x86 architecture and the industry built around it. VMware Workstation addressed these new challenges by combining well-known virtualization techniques, techniques from other domains, and new techniques into a single solution.

To allow virtualization to be inserted into existing systems, VMware Workstation combined a hosted architecture with a virtual machine monitor (VMM). The hosted architecture enabled a simple user experience and offered broad hardware compatibility. The architecture enabled, with minimal interference, the system-level co-residency of a host operating system and a VMM. Rather than exposing the x86 platform's I/O diversity to the virtual machines, VMware Workstation relied on software emulation of canonically chosen I/O devices, thereby also enabling the hardware-independent encapsulation of virtual machines.

The VMware VMM compensated for the lack of architectural support for virtualization by combining a trap-and-emulate direct execution engine with a system-level binary translator to efficiently virtualize the x86 architecture and support most commodity operating systems. The VMM used segmentation as a protection mechanism, allowing its binary translated code to execute at near hardware speeds. The VMM also employed adaptive binary translation to greatly reduce the overhead of virtualizing in-memory x86 data structures.

The rest of this article is organized as follows: we start with the statement of the problem and associated challenges in Section 2, followed by an overview of the solution and key contributions in Section 3. After a brief technical primer on the x86 architecture in Section 4, we then describe the key technical challenges of that architecture in Section 5. Section 6 covers the design and implementation of VMware Workstation, with a focus on the hosted architecture (Section 6.1), the VMM (Section 6.2), and its dynamic binary translator (Section 6.3). In Section 7, we evaluate the system, including its level of compatibility and performance. In Section 8, we discuss lessons learned during the development process. In Section 9, we describe briefly how the system has evolved from its original version, in particular as the result of hardware trends. We discuss related approaches and systems in Section 10 and conclude in Section 11.

2. CHALLENGES IN BRINGING VIRTUALIZATION TO THE X86 ARCHITECTURE

Virtual machine monitors (VMMs) apply the well-known principle of *adding a level of indirection* to the domain of computer hardware. VMMs operate directly on the real (physical) hardware, interposing between it and a *guest* operating system. They provide the abstraction of *virtual machines*: multiple copies of the underlying hardware, each running an independent operating system instance [Popek and Goldberg 1974].

> A virtual machine is taken to be an *efficient, isolated duplicate* of the real machine. We explain these notions through the idea of a virtual machine monitor (VMM). As a piece of software a VMM has three essential characteristics. First, the VMM provides an environment for programs that is essentially identical with the original machine; second, programs run in this environment show at worst only minor decreases in speed; and last, the VMM is in complete control of system resources.

At VMware, we adapted these three core attributes of a virtual machine to x86-based target platform as the following.

— *Compatibility*. The notion of an *essentially identical environment* meant that any x86 operating system, and all of its applications, would be able to run without modifications as a virtual machine. A VMM needed to provide sufficient compatibility at the hardware level such that users could run whichever operating system, down to the update and patch version, they wished to install within a particular virtual machine, without restrictions.

— *Performance*. We believed that *minor decreases in speed* meant sufficiently low VMM overheads that users could use a virtual machine as their primary work environment. As a design goal, we aimed to run relevant workloads at near native speeds, and in the worst case to run them on then-current processor with the same performance as if they were running natively on the immediately prior generation of processors. This was based on the observation that most x86 software wasn't designed to only run on the latest generation of CPUs.

— *Isolation*. A VMM had to guarantee the isolation of the virtual machine without making any assumptions about the software running inside. That is, a VMM needed to be in *complete control of resources*. Software running inside virtual machines had to be prevented from any access that would allow it to modify or subvert its VMM. Similarly, a VMM had to ensure the privacy of all data not belonging to the virtual machine. A VMM had to assume that the guest operating system could be infected with unknown, malicious code (a much bigger concern today than during the mainframe era).

There was an inevitable tension between these three requirements. For example, total compatibility in certain areas might lead to a prohibitive impact on performance, in which case we would compromise. However, we ruled out any tradeoffs that might compromise isolation or expose the VMM to attacks by a malicious guest. Overall, we identified four major challenges to building a VMM for the x86 architecture.

(1) *The x86 architecture was not virtualizable.* It contained virtualization-sensitive, unprivileged instructions, which violated the Popek and Goldberg [1974] criteria for strict virtualization. This ruled out the traditional trap-and-emulate approach to virtualization. Indeed, engineers from Intel Corporation were convinced their processors could not be virtualized in any practical sense [Gelsinger 1998].

(2) *The x86 architecture was of daunting complexity.* The x86 architecture was a notoriously big CISC architecture, including legacy support for multiple decades of backwards compatibility. Over the years, it had introduced four main modes of operations (real, protected, v8086, and system management), each of which enabled in different ways the hardware's segmentation model, paging mechanisms, protection rings, and security features (such as call gates).

(3) *x86 machines had diverse peripherals.* Although there were only two major x86 processor vendors, the personal computers of the time could contain an enormous variety of add-in cards and devices, each with their own vendor-specific device drivers. Virtualizing all these peripherals was intractable. This had dual implications: it applied to both the front-end (the virtual hardware exposed in the virtual machines) and the back-end (the real hardware the VMM needed to be able to control) of peripherals.

(4) *Need for a simple user experience.* Classic VMMs were installed in the factory. We needed to add our VMM to existing systems, which forced us to consider software delivery options and a user experience that encouraged simple user adoption.

3. SOLUTION OVERVIEW

This section describes at a high level how VMware Workstation addressed the challenges mentioned in the previous section. Section 3.1 covers the nonvirtualizability of the x86 architecture. Section 3.2 describes the guest operating system-strategy used throughout the development phase, which was instrumental in helping mitigate the deep complexity of the architecture. Section 3.3 describes the virtual hardware platform, which addresses one half of the peripheral diversity challenge. Section 3.4 describes the role of the host operating system in VMware Workstation, which addresses the second half of peripheral diversity, as well as the user experience challenge.

3.1. Virtualizing the x86 Architecture

A VMM built for a virtualizable architecture uses a technique known as trap-and-emulate to execute the virtual machine's instruction sequence directly, but safely, on the hardware. When this is not possible, one approach, which we used in Disco, is to specify a virtualizable subset of the processor architecture, and port the guest operating systems to that newly defined platform. This technique is known as paravirtualization [Barham et al. 2003; Whitaker et al. 2002] and requires source-code level modifications of the operating system. Paravirtualization was infeasible at VMware because of the compatibility requirement, and the need to run operating systems that we could not modify.

An alternative would have been to employ an all emulation approach. Our experience with the SimOS [Rosenblum et al. 1997] machine simulator showed that the use of techniques such as *dynamic binary translation* running in a user-level program could limit overheads of complete emulation to a factor of 5 slowdown. While that was fast for a machine simulation environment, it was clearly inadequate for our performance requirements.

Our solution to this problem combined two key insights. First, although trap-and-emulate direct execution could not be used to virtualize the entire x86 architecture, it could actually be used some of the time. And in particular, it could be used during the execution of application programs, which accounted for most of the execution time on relevant workloads. As a fallback, dynamic binary translation would be used to execute just the system software. The second key insight was that by properly configuring the hardware, particularly using the x86 segment protection mechanisms carefully, system code under dynamic binary translation could also run at near-native speeds.

Section 6.2 describes the detailed design and implementation of this hybrid direct execution and binary translation solution, including an algorithm that determines when dynamic binary translation is necessary, and when direct execution is possible. Section 6.3 describes the design and implementation of the dynamic binary translator.

3.2. A Guest Operating System-Centric Strategy

The idea behind virtualization is to make the virtual machine interface identical to the hardware interface so that all software that runs on the hardware will also run in a virtual machine. Unfortunately, the description of the x86 architecture, publicly available as the *Intel Architecture Manual* [Intel Corporation 2010], was at once baroquely detailed and woefully imprecise for our purpose. For example, the formal specification of a single instruction could easily exceed 8 pages of pseudocode while omitting crucial details necessary for correct virtualization. We quickly realized that attempting to implement the entire processor manual was not the appropriate bootstrapping strategy.

Instead, we chose a list of key guest operating systems to support and worked through them, initially one at a time. We started with a minimal set of features and progressively enhanced the completeness of the solution, while always preserving the correctness of the supported feature set. Practically speaking, we made very restrictive assumptions on how the processor's privileged state could be configured by the guest operating system. Any attempt to enter into an unsupported combination of privileged state would cause the virtual machine to stop executing.

We started with Linux for our first guest operating system. This turned out to be mostly straightforward as Linux was designed for portability across multiple processors, we were familiar with its internals, and of course had access to the source code. At that point, the system, although early in development, already could run Linux efficient in an isolated virtual machine. Of course, we had only encountered, and therefore implemented, a tiny fraction of the possible architectural combinations.

After Linux, we tackled Windows 95, the most popular desktop operating system of the time. That turned out to be *much* more difficult. Windows 95 makes extensive use of a combination of 16-bit and 32-bit protected mode, can run MS-DOS applications using the processor's v8086 mode, occasionally drops into real mode, makes numerous BIOS calls, and makes extensive and complex use of segmentation, and manipulates segment descriptor tables extensively [King 1995]. Unlike Linux, we had no access to the Windows 95 source code and, quite frankly, did not fully understand its overall design. Building support for Windows 95 forced us to greatly expand the scope of our efforts, including developing an extensive debugging and logging framework and diving into numerous arcane elements of technology. For the first time, we also had to deal with ambiguous or undocumented features of the x86 architecture, upon whose correct emulation Windows 95 depended. Along the way, we also ensured that MS-DOS and its various extensions could run effectively in a virtual machine.

Next, we focused on Windows NT [Custer 1993], an operating system aimed at enterprise customers. Once again, this new operating system configured the hardware differently, in particular in the layout of its linear address space. We had to further increase our coverage of the architecture to get Windows NT to boot, and develop new mechanisms to get acceptable performance.

This illustrates the critical importance of prioritizing the guest operating systems to support. Although our VMM did not depend on any internal semantics or interfaces of its guest operating systems, it depended heavily on understanding the ways they configured the hardware. Case-in-point: we considered supporting OS/2, a legacy operating system from IBM. However, OS/2 made extensive use of many features of the x86 architecture that we never encountered with other guest operating systems. Furthermore, the way in which these features were used made it particularly hard to virtualize. Ultimately, although we invested a significant amount of time in OS/2-specific optimizations, we ended up abandoning the effort.

3.3. The Virtual Hardware Platform

The diversity of I/O peripherals in x86 personal computers made it impossible to match the virtual hardware to the real, underlying hardware. Whereas there were only a handful of x86 processor models in the market, with only minor variations in instruction-set level capabilities, there were hundreds of I/O devices most of which had no publically available documentation of their interface or functionality. Our key insight was to *not* attempt to have the virtual hardware match the specific underlying hardware, but instead have it always match some configuration composed of selected, canonical I/O devices. Guest operating systems then used their own existing, built-in mechanisms to detect and operate these (virtual) devices.

The virtualization platform consisted of a combination of multiplexed and emulated components. Multiplexing meant configuring the hardware so it can be directly used by the virtual machine, and shared (in space or time) across multiple virtual machines. Emulation meant exporting a software simulation of the selected, canonical hardware

Table I. Virtual Hardware Configuration Options of VMware Workstation 2.0

	Virtual Hardware (front-end)	*Back-end*
Multiplexed	1 virtual x86 CPU, with the same instruction set extensions as the underlying hardware CPU	Scheduled by the host operating system on either a uniprocessor or multiprocessor host
	Up to 512MB of contiguous DRAM (user configurable)	Allocated and managed by the host OS (page-by-page)
Emulated	PCI Bus	Fully emulated compliant PCI bus with B/D/F addressing for all virtual motherboard and slot devices
	4x 4IDE disks 7x Buslogic SCSI Disks	Virtual disks (stored as files) or direct access to a given raw device
	1x IDE CD-ROM	ISO image or emulated access to the real CD-ROM
	2x 1.44MB floppy drives	Physical floppy or floppy image
	1x VMware graphics card with VGA and SVGA support	Ran in a window and in full-screen mode. SVGA required VMware SVGA guest driver
	2x serial ports COM1 and COM2	Connect to Host serial port or a file
	1x printer (LPT)	Can connect to host LPT port
	1x keyboard (104-key)	Fully emulated; keycode events are generated when they are received by the VMware application
	1x PS-2 mouse	Same as keyboard
	3x AMD PCnet Ethernet cards (Lance Am79C970A)	Bridge mode and host-only modes
	1x Soundblaster 16b	Fully emulated

component to the virtual machine. Table I shows that we used multiplexing for processor and memory and emulation for everything else.

For the multiplexed hardware, each virtual machine had the illusion of having one dedicated CPU[2] and a fixed amount of contiguous RAM starting at physical address 0. The amount was configurable, up to 512MB in VMware Workstation 2.

Architecturally, the emulation of each virtual device was split between a front-end component, which was visible to the virtual machine, and a backend component, which interacted with the host operating system [Waldspurger and Rosenblum 2012]. The front-end was essentially a software model of the hardware device that could be controlled by unmodified device drivers running inside the virtual machine. Regardless of the specific corresponding physical hardware on the host, the front-end always exposed the same device model.

This approach provided VMware virtual machines with an additional key attribute: *hardware-independent encapsulation*. Since devices were emulated and the processor was under the full control of the VMM, the VMM could at all times enumerate the entire state of the virtual machine, and resume execution even on a machine with a totally different set of hardware peripherals. This enabled subsequent innovations such as suspend/resume, checkpointing, and the transparent migration of live virtual machines across physical boundaries [Nelson et al. 2005].

For example, the first Ethernet device front-end was the AMD PCnet “lance”, once a popular 10Mbit NIC [AMD Corporation 1998], and the backend provided network connectivity at layer-2 to either the host’s physical network (with the host acting as a bridge), or to a software network connected to only the host and other VMs on the same host. Ironically, VMware kept supporting the PCnet device long after physical

[2] Throughout the article, the term CPU always refers to a (32-bit) hardware thread of control, and never to a distinct core or socket of silicon.

PCnet NICs had been obsoleted, and actually achieved I/O that was orders of magnitude faster than 10Mbit [Sugerman et al. 2001]. For storage devices, the original front-ends were an IDE controller and a Buslogic Controller, and the backend was typically either a file in the host filesystem, such as a virtual disk or an ISO 9660 image [International Standards Organization 1988], or a raw resource such as a drive partition or the physical CD-ROM.

We chose the various emulated devices for the platform as follows: first, many devices were mandatory to run the BIOS, and then install and boot the commercial operating systems of the time. This was the case for the various software-visible components on a computer motherboard of the time, such as PS/2 keyboard and mouse, VGA graphics card, IDE disks and PCI root complex, for which the operating systems of the era had built in, non-modular device drivers.

For more advanced devices, often found on PCI expansion cards such as network interfaces or disk controllers, we tried to chooses one representative device within its class that had broad operating system support with existing drivers, and for which we had access to acceptable documentation. We took that approach to initially support one SCSI disk controller (Buslogic), one Ethernet NIC (PCnet), and a sound card (Creative Labs' 16-bit Soundblaster card).

In a few cases, we invented our own devices when necessary, which also required us to write and ship the corresponding device driver to run within the guest. For example, graphics adapters tended to be very proprietary, lacking in any public programming documentation, and highly complex. Instead, we designed our own virtual SVGA card. Later, VMware engineers applied the same approach to improve networking and storage performance, and implemented higher-performance paravirtualized devices, plus custom drivers, as an alternative to the fully emulated devices (driven by drivers included in the guest operating system).

New device classes were added in subsequent versions of the product, most notably broad support for USB. With time, we also added new emulated devices of the existing classes as PC hardware evolved, for example, the Intel e1000 NIC, es1371 PCI sound card, LSIlogic SCSI disk controller, etc.

Finally, every computer needs some platform-specific firmware to first initialize the hardware, then load system software from the hard disk, CDROM, floppy, or the network. On x86 platforms, the BIOS [Compaq, Phoenix, Intel 1996] performs this role, as well as substantial run-time support, for example, to print on the screen, to read from disk, to discover hardware, or to configure the platform via ACPI. When a VMware virtual machine was first initialized, the VMM loaded into the virtual machine's ROM a copy of the VMware BIOS.

Rather than writing our own BIOS, VMware Inc. licensed a proven one from Phoenix Technologies, and acted as if it were a motherboard manufacturer: we customized the BIOS for the particular combination of chipset components emulated by VMware Workstation. This full-featured BIOS played a critical role in allowing the broadest support of guest operating systems, including legacy operating systems such as MS-DOS and Windows 95/98 that relied heavily on the BIOS.

3.4. The Role of the Host Operating System

We developed the VMware Hosted Architecture to allow virtualization to be inserted into existing systems. It consisted of packaging VMware Workstation to feel like a normal application to a user, and yet still have direct access to the hardware to multiplex CPU and memory resources.

Like any application, the VMware Workstation installer simply writes its component files onto an existing host file system, without perturbing the hardware configuration (no reformatting of a disk, creating of a disk partition, or changing of BIOS settings). In fact, VMware Workstation could be installed and start running virtual machines without requiring even rebooting the host operating system, at least on Linux hosts.

Running on top of a host operating system provided a key solution to the back-end aspect of the I/O device diversity challenge. Whereas there was no practical way to build a VMM that could talk to every I/O devices in a system, an existing host operating system could already, using its own device drivers. Rather than accessing physical devices directly, VMware Workstation backed its emulated devices with standard system calls to the host operating system. For example, it would read or write a file in the host file system to emulate a virtual disk device, or draw in a window of the host's desktop to emulate a video card. As long as the host operating system had the appropriate drivers, VMware Workstation could run virtual machines on top of it.

However, a normal application does not have the necessary hooks and APIs for a VMM to multiplex the CPU and memory resources. As a result, VMware Workstation only appears to run on top of an existing operating system when in fact its VMM can operate at system level, in full control of the hardware. Section 6.1 describes how the architecture enabled (i) both host operating system and the VMM to coexist simultaneously at system level without interfering with each other, and (ii) VMware Workstation to use the host operating system for the backend I/O emulation.

Running as a normal application had a number of user experience advantages. VMware Workstation relied on the host graphical user interface so that the content of each virtual machine's screen would naturally appear within a distinct window. Each virtual machine instance ran as a process on the host operating system, which could be independently started, monitored, controlled, and terminated. The host operating system managed the global resources of the system: CPU, memory and I/O. By separating the virtualization layer from the global resource management layer, VMware Workstation follows the *type II* model of virtualization [Goldberg 1972].

4. A PRIMER ON THE X86 PERSONAL COMPUTER ARCHITECTURE

This section provides some background on the x86 architecture necessary to appreciate the technical challenges associated with its virtualization. Readers familiar with the architecture can skip to Section 5. For a complete reference on the x86 system architecture, see the ISA reference and OS writer's guide manuals from AMD or Intel [Intel Corporation 2010]. As in the rest of the article, we refer to *x86 architecture* as Intel and AMD defined it before the introduction of 64-bit extensions or of hardware virtualization support.[3]

The x86 architecture is a complex instruction set computing (CISC) architecture with six 32-bit, general-purpose registers (`%eax, %ebx, %ecx, %edx, %esp, %ebp`). Each register can also be used as a 16-bit register (e.g., `%ax`), or even as an 8-bit register (e.g., `%ah`). In addition, the architecture has six segment registers (`%cs, %ss, %ds, %es, %fs, %gs`). Each segment register has a visible portion, the *selector*, and a hidden portion. The visible portion can be read or written to by software running at any privilege level. As the name implies, the hidden portion is not directly accessible by software. Instead, writing to a segment register via its selector populates the corresponding hidden portion, with specific and distinct semantics depending on the mode of the processor.

[3] As the 32-bit x86 is still available today in shipping processors, we use the present tense when describing elements of the x86 architecture, and the past tense when describing the original VMware Workstation.

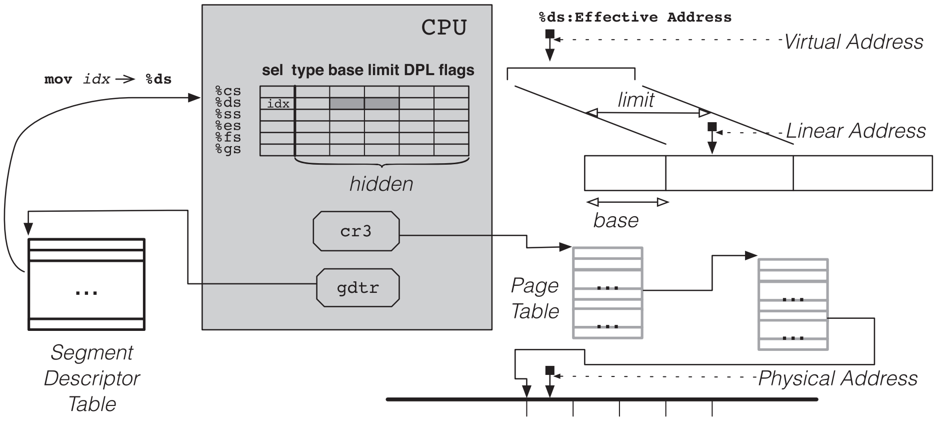


Fig. 1. Segmentation and paging in x86 protected mode, with two illustrative examples: (i) the assignment of %ds with the value idx updates both the segment selector (with idx) and the hidden portion of the segment (with the parameters loaded from the descriptor table at index idx); and (ii) the relocation of a virtual address that consists of %ds and an effective address into a linear address (through segmentation) and then a physical address (through paging).

The six segment registers each define a distinct *virtual address space*, which maps onto a single, 32-bit *linear address space*. Each virtual address space can use either 16-bit or 32-bit addresses, has a base linear address, a length, and additional attributes.

Instructions nearly always use virtual addresses that are a combination of a segment and an effective address (e.g., %ds:Effective Address in Figure 1). By default, the six segments are used implicitly as follows: the instruction pointer %eip is an address within the code segment %cs. Stack operations such as push and pop, as well as memory references using the stack pointer %esp or base pointer register %ebp, use the stack segment %ss. Other memory references use the data segment %ds. String operations additionally use the extra segment %es. An instruction prefix can override the default segment with any of the other segments, including %fs and %gs.

4.1. Segmentation in Protected Mode

In protected mode (the native mode of the CPU), the x86 architecture supports the atypical combination of segmentation and paging mechanisms, each programmed into the hardware via data structures stored in memory. Figure 1 illustrates these controlling structures as well as the address spaces that they define. When a selector of a segment register is assigned in protected mode (e.g., mov idx→ %ds in Figure 1) the CPU additionally interprets the segment selector idx as an index into one of the two segment descriptor tables (the global and the local descriptor table), whose locations are stored in privileged registers of the CPU. The CPU automatically copies the descriptor entry into the corresponding hidden registers.

Although applications can freely assign segments during their execution, system software can use segmentation as a protection mechanism as long as it can prevent applications from directly modifying the content of the global and local descriptor tables, typically by using page-based protection to prevent applications from modifying the tables. In addition, the hardware contains a useful mechanism to restrict segment assignment: each descriptor has a privilege level field (dpl), which restricts whether assignment is even possible based on the current privilege level of the CPU.

4.2. Paging

At any given time, each x86 CPU has a single linear address space, which spans the full addressable 32-bit range. Paging controls the mapping from linear addresses onto physical memory on a page-by-page basis.

The x86 architecture supports hardware page tables. The control register `%cr3` determines the root of a 2-level page table structure for a standard 32-bit physical address space (shown in Figure 1) or a 3-level page table structure for a 36-bit extended physical address space (*PAE* mode, not shown in the figure). The first-level page contains *page descriptor entries* (`pde`) and the second-level page contains *page table entries* (`pte`). The format of the page table structure is defined by the architecture, and accessed directly by the CPU. Page table accesses occur when a linear address reference triggers a miss in the translation-lookaside buffer (TLB) of the CPU.

4.3. Modes, Rings, and Flags

In addition to paging and segmentation, the 32-bit x86 architecture has a rich complexity resulting from its multi-decade legacy. For example, the system state of the CPU is distributed in a combination of control registers (`%cr0..%cr4`), privileged registers (`%idtr, %gdtr`) and privileged parts of the `%eflags` register such as `%eflags.v8086`, `%eflags.if`, and `%eflags.iopl`.

The x86 CPU has four formal operating modes: protected mode, v8086 mode, real mode, and system management mode. The CPU is in protected mode when `%cr0.pe=1`. Only protected mode uses the segment descriptor tables; the other modes are also segmented, but directly convert the selector into the base, by multiplying it by 16. As for the non-native modes: v8086 mode (when `%cr0.pe=1` and `%eflags.v8086=1`) is a 16-bit mode that can virtualize the 16-bit Intel 8086 architecture. It uses paging to convert linear addresses into physical addresses. Real mode is used by the BIOS at reset and by some operating systems, and runs directly from physical memory (i.e., it is not paged). In real mode, segment limits remain unmodified even in the presence of segment assignments; real mode can run with segment limits larger than 64KB, which is sometimes referred to as *big real mode* or *unreal mode*. System management is similar to real mode, but has unbounded segment limits. The BIOS uses system management mode to deliver features such as ACPI.

In protected mode, the architecture defines four protection rings (or levels): the bottom two bits of `%cs` defines the `current privilege level` of the CPU (`%cs.cpl`, shortened to `%cpl`). Privileged instructions are only allowed at `%cpl=0`. Instructions at `%cpl=3` can only access pages in the linear address space whose page tables entries specify `pte.us=1` (i.e., userspace access is allowed). Instructions at `%cpl=0` through `%cpl=2` can access the entire linear address space permitted by segmentation (as long as the page table mapping is valid).

The architecture has a single register (`%eflags`) that contains both condition codes and control flags, and can be assigned by instructions such as `popf`. However, some of the flags can be set only on certain conditions. For example, the interrupt flag (`%eflags.if`) changes only when `%cpl`≤`%eflags.iopl`.

4.4. I/O and Interrupts

The architecture has one interrupt descriptor table. This table contains the entry points of all exception and interrupt handlers, that is, it specifies the code segment, instruction pointer, stack segment, and stack pointer for each handler. The same interrupt table is used for both processor faults (e.g., a page fault) and external interrupts.

The processor interacts with I/O devices through a combination of programmed I/O and DMA. I/O ports may be memory mapped or mapped into a separate 16-bit I/O

Table II. List of Sensitive, Unprivileged x86 Instructions

Group	Instructions
Access to interrupt flag	`pushf, popf, iret`
Visibility into segment descriptors	`lar, verr, verw, lsl`
Segment manipulation instructions	`pop <seg>, push <seg>, mov <seg>`
Read-only access to privileged state	`sgdt, sldt, sidt, smsw`
Interrupt and gate instructions	`fcall, longjump, retfar, str, int <n>`

address space accessible by special instructions. Device interrupts, also called IRQs, are routed to the processor by either of two mechanisms in the chipset (motherboard), depending on OS configuration: a legacy interrupt controller called a PIC, or an advanced controller called an I/O APIC in conjunction with a local APIC resident on the CPU. IRQs can be delivered in two styles: edge or level. When edge-triggered, the PIC/APIC raises a single interrupt as a device raises its interrupt line. When level-triggered, the PIC/APIC may repeatedly reraise an interrupt until the initiating device lowers its interrupt line.

5. SPECIFIC TECHNICAL CHALLENGES

This section provides technical details on the main challenges faced in the development of VMware Workstation. We first present in Section 5.1 the well-known challenge of x86's sensitive, unprivileged instructions, which ruled out the possibility of building a VMM using a trap-and-emulate approach. We then describe four other challenges associated with the x86 architecture, each of which critically influenced our design. Section 5.2 describes address space compression (and associated isolation), which resulted from having a VMM (and in particular one with a dynamic binary translator) and a virtual machine share the same linear address space. Given the x86 paging and segmented architecture, virtualizing physical memory required us to find an efficient way to track changes in the memory of a virtual machine (Section 5.3). The segmented nature of the x86 architecture forced us to face the specific challenge of virtualizing segment descriptor tables (Section 5.4). The need to handle workloads that made nontrivial use of the legacy modes of the CPU forced us to also consider how to handle these modes of the CPU (Section 5.5). Finally, Section 5.6 describes a distinct challenge largely orthogonal to the instruction set architecture: how to build a system that combined the benefit of having a host operating system without the constraints of that host operating system.

5.1. Handling Sensitive, Unprivileged Instructions

Popek and Goldberg [1974] demonstrated that a simple VMM based on trap-and-emulate (direct execution) could be built only for architectures in which all virtualization-sensitive instructions are also all privileged instructions. For architectures that meet their criteria, a VMM simply runs virtual machine instructions in de-privileged mode (i.e., never in the most privileged mode) and handles the traps that result from the execution of privileged instructions. Table II lists the instructions of the x86 architecture that unfortunately violated Popek and Goldberg's rule and hence made the x86 non-virtualizable [Robin and Irvine 2000].

The first group of instructions manipulates the interrupt flag (`%eflags.if`) when executed in a privileged mode ($\texttt{\%cpl} \leq \texttt{\%eflags.iopl}$) but leave the flag unchanged otherwise. Unfortunately, operating systems used these instructions to alter the interrupt state, and silently disregarding the interrupt flag would prevent a VMM using a trap-and-emulate approach from correctly tracking the interrupt state of the virtual machine.

The second group of instructions provides visibility into segment descriptors in the global or local descriptor table. For de-privileging and protection reasons, the VMM needs to control the actual hardware segment descriptor tables. When running directly in the virtual machine, these instructions would access the VMM's tables (rather than the ones managed by the operating system), thereby confusing the software.

The third group of instructions manipulates segment registers. This is problematic since the privilege level of the processor is visible in the code segment register. For example, `push %cs` copies the `%cpl` as the lower 2 bits of the word pushed onto the stack. Software in a virtual machine that expected to run at `%cpl=0` could have unexpected behavior if `push %cs` were to be issued directly on the CPU.

The fourth group of instructions provides read-only access to privileged registers such as `%idtr`. If executed directly, such instructions return the address of the VMM structures, and not those specified by the virtual machine's operating system. Intel classifies these instructions as "only useful in operating-system software; however, they can be used in application programs without causing an exception to be generated" [Intel Corporation 2010], an unfortunate design choice when considering virtualization.

Finally, the x86 architecture has extensive support to allow controlled transitions between various protection rings using interrupts or call gates. These instructions are also subject to different behavior when de-privileging.

5.2. Address Space Compression

A VMM needs to be co-resident in memory with the virtual machine. For the x86, this means that some portion of the linear address space of the CPU must be reserved for the VMM. That reserved space must at a minimum include the required hardware data structures such as the interrupt descriptor table and the global descriptor table, and the corresponding software exception handlers. The term *address space compression* refers to the challenges of a virtual machine and a VMM co-existing in the single linear address space of the CPU.

Although modern operating system environment do not actively use the entire 32-bit address space at any moment in time, the isolation criteria requires that the VMM be protected from any accesses by the virtual machine (accidental or malicious) to VMM memory. At the same time, the compatibility criteria means that accesses from the virtual machine to addresses in this range must be emulated in some way.

The use of a dynamic binary translator adds another element to the problem: the code running via dynamic binary translation will be executing a mix of virtual machine instructions (and their memory references) and additional instructions that interact with the binary translator's run-time environment itself (within the VMM). The VMM must both enforce the isolation of the sandbox for virtual machine instructions, and yet provide the additional instructions with a low-overhead access to the VMM memory.

5.3. Tracking Changes in Virtual Machine Memory

When virtualizing memory, the classic VMM implementation technique is for the VMM to keep "shadow" copies of the hardware memory-management data structures stored in memory. The VMM must detect changes to these structures and apply them to the shadow copy. Although this technique has long been used in classic VMMs, the x86 architecture, with its segment descriptor tables and multi-level page tables, presented particular challenges because of the size of these structures and the way they were used by modern operating systems.

In the x86 architecture, privileged hardware registers contain the address of segment descriptor tables (%gdtr) and page tables (%cr3), but regular load and store instructions can access these structures in memory. To correctly maintain the shadow copy, a VMM must intercept and track changes made by these instructions. To make matters more challenging, modern operating systems do not partition the memory used for these structures from the rest of the memory.

These challenge led us to build a mechanism called *memory tracing*, described in Section 6.2.2. As we will show, the implementation tradeoffs associated with memory tracing turned out to be surprisingly complex; as a matter of fact, memory tracing itself had a significant impact on the performance of later processors with built-in virtualization support.

5.4. Virtualizing Segmentation

The specific semantics of x86 segments require a VMM to handle conditions outside the scope of the standard shadowing approach. The architecture explicitly defines the precise semantics on when to update the segment registers of a CPU [Intel Corporation 2010]: when a memory reference modifies an in-memory segment descriptor entry in either the global or the local descriptor table the contents of the hidden portions are *not* updated. Rather, the hidden portion is only refreshed when the (visible) segment selector is written to, either by an instruction or as part of a fault. As a consequence of this, the content of the hidden registers can only be inferred if the value in memory of the corresponding entry has not been modified. We define a segment to be *nonreversible* if the in-memory descriptor entry has changed as the hidden state of that segment can no longer be determined by software.

This is not an esoteric issue. Rather, a number of legacy operating systems rely on nonreversible semantics for their correct execution. These operating systems typically run with nonreversible segments in portions of its code that do not cause traps and run with interrupts disabled. They can therefore run indefinitely in that mode, and rightfully assume that the hidden segment state is preserved. Unfortunately, additional traps and interrupts caused by a VMM breaks this assumption. Section 6.2.3 describes our solution to this challenge.

5.5. Virtualizing Non-Native Modes of the Processor

The many different execution modes of the x86 such as real mode and system management mode presented a challenge for the virtualization layer, as they alter the execution semantics of the processor. For example, real mode is used by the BIOS firmware, legacy environments such as DOS and Windows 3.1, as well as by the boot code and installer programs of modern operating systems.

In addition, Intel introduced v8086 mode with the 80386 to run legacy 8086 code. v8086 mode is used by systems such as MS-DOS (through EMM386), Windows 95/98 and Windows NT to run MS-DOS applications. Ironically, while v8086 mode actually meets all of Goldberg and Popek's criteria for strict virtualizeability (i.e., programs in v8086 mode are virtualizable), it cannot be used to virtualize any real mode program that takes advantage of 32-bit extensions, and/or interleaves real mode and protected mode instruction sequences, patterns that are commonly used by the BIOS and MS-DOS extenders such as XMS's HIMEM.SYS [Chappell 1994]. Often, the purpose of the protected mode sequence is to load segments in the CPU in big real mode.

Since v8086 mode is ruled out as a mechanism to virtualize either real mode or system management mode, the challenge is to virtualize these legacy modes with the hardware CPU in protected mode.

5.6. Interference from the Host Operating System

VMware Workstation's reliance on a host operating system solved many problems stemming from I/O device diversity, but created another challenge: it was not possible to run a VMM inside the process abstraction offered to applications by modern operating systems such as Linux and Windows NT/2000. Both constrained application programs to a small subset of the address space, and obviously prevented applications from executing privileged instructions or directly manipulating hardware-defined structures such as descriptor tables and page tables.

On the flip side, the host operating system expects to manage all I/O devices, for example, by programming the interrupt controller and by handling all external interrupts generated by I/O devices. Here as well, the VMM must cooperate to ensure that the host operating system can perform its role, even though its device drivers might not be in the VMM address space.

6. DESIGN AND IMPLEMENTATION

We now demonstrate that the x86 architecture, despite the challenges described in the previous section, is actually virtualizable. VMware Workstation provides the existence proof for our claim as it allowed unmodified commodity operating systems to run in virtual machines with the necessary compatibility, efficiency and isolation.

We describe here three major technical contributions of VMware Workstation: Section 6.1 describes the VMware hosted architecture and the world switch mechanism that allowed a VMM to run free of any interference from the host operating system, and yet on top of that host operating system. Section 6.2 describes the VMM itself. One critical aspect of our solution was the decision to combine the classic direct execution subsystem with a low-overhead dynamic binary translator that could run most instruction sequences at near-native hardware speeds. Specifically, the VMM could automatically determine, at any point in time, whether the virtual machine was in a state that allowed direct execution subsystem, or whether it required dynamic binary translation. Section 6.3 describes the internals of the dynamic binary translator, and in particular the design options that enabled it to run most virtual machine instruction sequences at hardware speeds.

6.1. The VMware Hosted Architecture

From the user's perspective, VMware Workstation ran like any regular application, on top of an existing host operating system, with all of the benefits that this implies. The first challenge was to build a virtual machine monitor that was invoked within an operating system but could operate without any interference from it.

The VMware Hosted Architecture, first disclosed in Bugnion et al. [1998], allows the co-existence of two independent system-level entities – the host operating system and the VMM. It was introduced with VMware Workstation in 1999. Today, although significant aspects of the implementation have evolved, the architecture is still the fundamental building block of all of VMware's hosted products.

In this approach, the host operating system rightfully assumes that it is in control of the hardware resources at all times. However, the VMM actually does take control of the hardware for some bounded amount of time during which the host operating system is temporarily removed from virtual and linear memory. Our design allowed for a single CPU to switch dynamically and efficiently between these two modes of operation.

6.1.1. System Components. Figure 2 illustrates the concept of system-level co-residency of the hosted architecture, as well as the key building blocks that implemented it. At any point in time, each CPU could be either in the host operating system

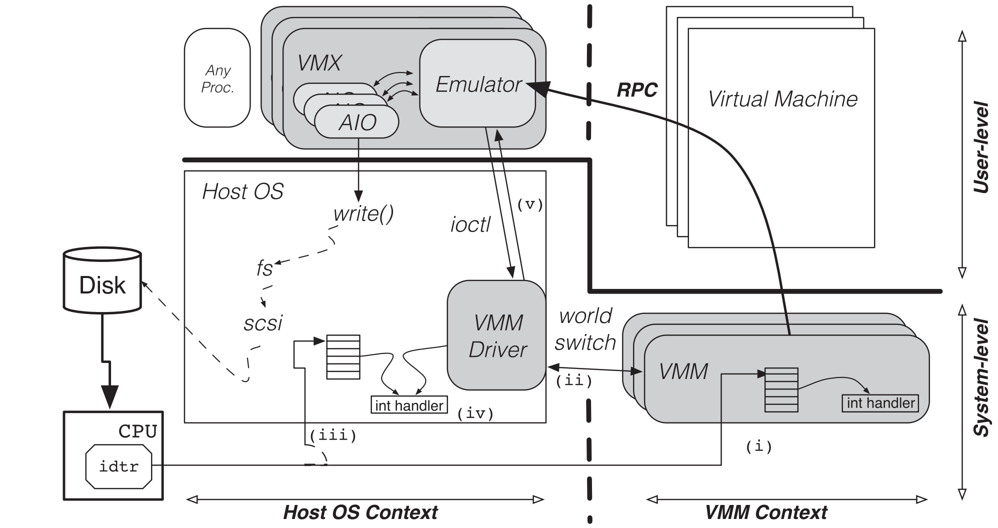


Fig. 2. The VMware Hosted Architecture. VMware Workstation consists of the three shaded components. The figure is split vertically between host operating system context and VMM context, and horizontally between system-level and user-level execution. The steps labeled (i)–(v) correspond to the execution that follows an external interrupt that occurs while the CPU is executing in VMM context.

context – in which case the host operating system was fully in control – or in the VMM context – where the VMM was fully in control. The transition between these two contexts was called the *world switch*.

The two contexts operated independently of each other, each with their own address spaces, segment and interrupt descriptor tables, stacks, and execution contexts. For example, the `%idtr` register defined a different set of interrupt handlers for each context (as illustrated in the figure).

Each context ran both trusted code in its system space and untrusted code in user space. In the operating system context, the classic separation of kernel vs. application applied. In the VMM context, however, the VMM was part of the trusted code base, but the entire virtual machine (including the guest operating system) is actually treated as an untrusted piece of software.

Figure 2 illustrates other aspects of the design. Each virtual machine was controlled by a distinct VMM instance and a distinct instance of a process of the host operating system, labeled the VMX. This multi-instance design simplified the implementation while supporting multiple concurrent virtual machines. As a consequence, the host operating system was responsible for globally managing and scheduling resources between the various virtual machines and native applications.

Figure 2 also shows a kernel-resident driver. The driver implemented a set of operations, including locking physical memory pages, forwarding interrupts, and calling the world switch primitive. As far as the host operating system was concerned, the device driver was a standard loadable kernel module. But instead of driving some hardware device, it drove the VMM and hid it entirely from the host operating system.

At virtual machine startup, the VMX opened the device node managed by the kernel-resident driver. The resulting file descriptor was used in all subsequent interactions between the VMX and the kernel-resident driver to specify the virtual machine. In particular, the main loop of the VMX's primary thread repeatedly issued `ioctl(run)` to trigger a world switch to the VMM context and run the virtual machine. As a result, the VMM and the VMX's primary thread, despite operating in distinct contexts and address spaces, jointly execute as co-routines within a single thread of control.

6.1.2. I/O and Interrupts. As a key principle of the co-resident design, the host operating system needed to remain oblivious to the existence of the VMM, including in situations where a device raises an interrupt. Since external physical interrupts were generated by actual physical devices such as disks and NICs, they could interrupt the CPU while it was running in either the host operating system context or the VMM context of some virtual machine. The VMware software was totally oblivious to the handling of external interrupts in the first case: the CPU transferred control to the handler as specified by the host operating system via the interrupt descriptor table.

The second case was more complex. The interrupt could occur in any VMM context, not necessarily for a virtual machine with pending I/O requests. Figure 2 illustrates specifically the steps involved through the labels (i)–(v). In step (i), the CPU interrupted the VMM and started the execution of the VMM's external interrupt handler. The handler did not interact with the virtual machine state, the hardware device or even the interrupt controller. Rather, that interrupt handler immediately triggered a world switch transition back to the host operating system context (ii). As part of the world switch, the `%idtr` was restored back to point to the host operating system's interrupt table (iii). Then, the kernel-resident driver transitioned control to the interrupt handler specified by the host operating system (iv). This was actually implemented by simply issuing an `int <vector>` instruction, with `<vector>` corresponding to the original external interrupt. The host operating system's interrupt handler then ran normally, *as if* the external I/O interrupt had occurred while the VMM driver were processing an `ioctl` in the VMX process. The VMM driver then returned control back to the VMX process at userlevel, thereby providing the host operating system with the opportunity to make preemptive scheduling decisions (v).

Finally, in addition to illustrating the handling of physical interrupts, Figure 2 also shows how the VMware Workstation issued I/O requests on behalf of virtual machines. All such virtual I/O requests were performed using remote procedure calls [Birrell and Nelson 1984] between the VMM and the VMX, which then made normal system calls to the host operating system. To allow for overlapped execution of the virtual machine with its own pending I/O requests, the VMX was organized as a collection of threads (or processes depending on the implementation): the *Emulator* thread was dedicated solely to the main loop that executed the virtual machine and emulated the device front-ends as part of the processing of remote procedure calls. The other threads (*AIO*) were responsible for the execution of all potentially blocking operations.

For example, in the case of a disk write, the Emulator thread decoded the SCSI or IDE `write` command, selected an AIO thread for processing, and resumed execution of the virtual machine without waiting for I/O completion. The AIO thread in turn issued the necessary system calls to write to the virtual disk. After the operation completed, the Emulator raised the virtual machine's virtual interrupt line. That last step ensured that the VMM would next emulate an I/O interrupt within the virtual machine, causing the guest operating system's corresponding interrupt handler to execute to process the completion of the disk write.

6.1.3. The World Switch. The world switch depicted in Figure 2 is the low-level mechanism that frees the VMM from any interference from the host operating system, and vice-versa. Similar to a traditional context switch, which provides the low-level operating system mechanism that loads the context of a process, the world switch is the low-level VMM mechanism that loads and executes a virtual machine context, as well as the reverse mechanism that restores the host operating system context.

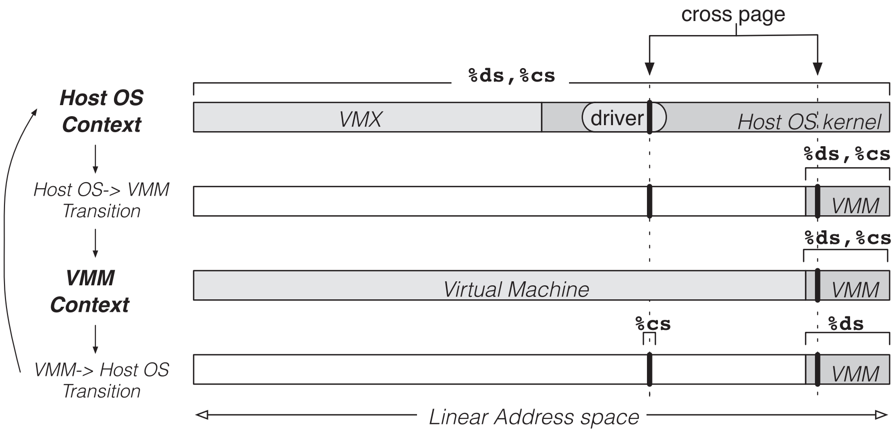


Fig. 3. Virtual and linear address spaces during a world switch.

Although subtle in a number of ways, the implementation was quite efficient and very robust. It relied on two basic observations: first, any change of the linear-to-physical address mapping via an assignment to `%cr3` required that at least one page—the one containing the current instruction stream—had the same content in both the outgoing and incoming address spaces. Otherwise, the instruction immediately following the assignment of `%cr3` would be left undetermined. The second observation is that certain system-level resources could be undefined as long as the processor did not access them, for example, the interrupt descriptor table (as long as interrupts are disabled) and the segment descriptor tables (as long as there are no segment assignments). By undefined, we mean that their respective registers (`%idtr` and `%gdtr`) could point temporarily to a linear address that contains random bits, or even an invalid page. These two observations allowed us to develop a carefully crafted instruction sequence that saved the outgoing context and restored an independent one.

Figure 3 illustrates how the world switch routine transitioned from the host operating system context to the VMM context, and subsequently back to the starting point. We observe that the VMM ran in the very high portion of the address space—the top 4MB actually—for reasons that we will explain later. The *cross page* was a single page of memory, used in a very specific manner that is central to the world switch. The cross page was allocated by the kernel-resident driver into the host operating system's kernel address space. Since the driver used standard APIs for the allocation, the host operating system determined the address of the cross page. Immediately before and after each world switch, the cross page was also mapped in the VMM address space. The cross page contained both the code and the data structures for the world switch. In an early version of VMware Workstation, the cross page instruction sequence consisted of only 45 instructions that executed symmetrically in both directions, and with interrupts disabled, to:

(1) first, save the old processor state: general-purpose registers, privileged registers, and segment registers;
(2) then, restore the new address space by assigning `%cr3`. All page table mappings immediately change, except the one of the cross page.
(3) Restore the global segment descriptor table register (`%gdtr`).
(4) With the `%gdtr` now pointing to the new descriptor table, restore `%ds`. From that point on, all data references to the cross page must use a different virtual address

to access the same data structure. However, because `%cs` is unchanged, instruction addresses remain the same.

(5) Restore the other segment registers, `%idtr`, and the general-purpose registers.
(6) Finally, restore `%cs` and `%eip` through a `longjump` instruction.

The world switch code was wrapped to appear like a standard function call to its call site in each context. In each context, the caller additionally saved, and later restored, additional state on its stack not handled by the world switch, for example, the local descriptor table registers (`%ldtr`), the segments that might rely on that table such as `%fs`, as well as debug registers.

Both host operating systems (Linux and Windows) configured the processor to use the *flat memory model*, where all segments span the entire linear address space. However, there were some complications because the cross page was actually not part of any of the VMM segments, which only spanned the top 4MB of the address space.[4] To address this, we additionally mapped the cross page in a second location within the VMM address space.

Despite the subtleties of the design and a near total inability to debug the code sequence, the world switch provided an extremely robust foundation for the architecture. The world switch code was also quite efficient: the latency of execution, measured on an end-to-end basis that included additional software logic, was measured to be 4.45 μs on a now-vintage 733-Mhz Pentium III CPU [Sugerman et al. 2001]. In our observation, the overall overhead of the hosted architecture was manageable as long as the world-switch frequency did not exceed a few hundred times per second. To achieve the goal, we ended up emulating portion of the networking and SVGA devices within the VMM itself, and made remote procedure calls to the VMX only when some backend interaction with the host operating system was required. As a further optimization, both devices also relied on batching techniques to reduce the number of transitions.

6.1.4. Managing Host-Physical Memory. The VMX process played a critical role in the overall architecture as the entity that represented the virtual machine on the host operating system. In particular, it played a critical role in the allocation, locking, and the eventual release of all memory resources. The VMX managed the virtual machine's physical memory as a file mapped into its address space, for example, using `mmap` on Linux. Selected pages were kept pinned into memory in the host operating system while in use by the VMM. This provided a convenient and efficient environment to emulate DMA by virtual I/O devices. A DMA operation became a simple `bcopy, read`, or `write` by the VMX into the appropriate portion of that mapped file.

The VMX and the kernel-resident driver together also provided the VMM with the host-physical addresses for pages of the virtual machine's guest-physical memory. The kernel-resident driver locked pages on behalf of the VMX-VMM pair and provided the host-physical address of these locked pages. Page locking was implemented using the mechanisms of the host operating system. For example, in the early version of the Linux host products, the driver just incremented the *use count* of the physical page.

As a result, the VMM only inserted into its own address space pages that had been locked in the host operating system context. Furthermore, the VMM unlocked memory according to some configurable policy. In the unlock operation, the driver first ensured that the host operating system treated the page as *dirty* for purposes of swapping, and only then decremented the use count. In effect, each VMM cooperatively regulated

[4] We opted to not run the VMM in the flat memory model to protect the virtual machine from corruption by bugs in the VMM. This choice also simplified the virtualization of 16-bit code by allowing us to place certain key structures in the first 64KB of the VMM address space.

its use of locked memory, so that the host operating system could apply its own policy decisions and if necessary swap out portions of the guest-physical memory to disk.

Our design also ensured that all memory would be released upon the termination of a virtual machine, including in cases of normal exit (*VM poweroff*), but also if the VMX process terminated abnormally (e.g., `kill -9`) or if the VMM panicked (e.g., by issuing a specific RPC to the VMX). To avoid any functional or data structure dependency on either the VMM or the VMX, the kernel-resident driver kept a corresponding list of all locked pages for each virtual machine. When the host operating system closed the device file descriptor corresponding to the virtual machine, either explicitly because of a `close` system call issued by the VMX, or implicitly as part of the cleanup phase of process termination, the kernel-resident driver simply unlocked all memory pages. This design had a significant software engineering benefit: we could develop the VMM without having to worry about deallocating resources upon exit, very much like the programming model offered by operating systems to user-space programs.

6.2. The Virtual Machine Monitor

The main function of the VMM was to virtualize the CPU and the main memory. At its core, the VMware VMM combined a direct execution subsystem with a dynamic binary translator, as first disclosed in Devine et al. [1998]. In simple terms, direct execution was used to run the guest applications and the dynamic binary translator was used to run the guest operating systems. The dynamic binary translator managed a large buffer, the translation cache, which contained safe, executable translations of virtual machine instruction sequences. So instead of executing virtual machine instructions directly, the processor executed the corresponding sequence within the translation cache.

This section describes the design and implementation of the VMM, which was specifically designed to virtualize the x86 architecture *before* the introduction of 64-bit extensions or hardware support for virtualization (VT-x and AMD-v). VMware's currently shipping VMMs are noticeably different from this original design [Agesen et al. 2010].

We first describe the overall organization and protection model of the VMM in Section 6.2.1. We then describe two essential building blocks of the VMM that virtualize and trace memory (Section 6.2.2) and virtualize segmentation (Section 6.2.3). With those building blocks in place, we then describe how the direct execution engine and the binary translator together virtualize the CPU in Section 6.2.4.

6.2.1. Protecting the VMM. The proper configuration of the underlying hardware was essential to ensure both correctness and performance. The challenge was to ensure that the VMM could share an address space with the virtual machine without being visible to it, and to do this with minimal performance overheads. Given that the x86 architecture supported both segmentation-based and paging-based protection mechanisms, a solution might have used either one or both mechanisms. For example, operating systems that use the flat memory model only use paging to protect themselves from applications.

In our original solution, the VMM used segmentation, and segmentation only, for protection. The linear address space was statically divided into two regions, one for the virtual machine and one for the VMM. Virtual machine segments were *truncated* by the VMM to ensure that they did not overlap with the VMM itself.

Figure 4 illustrates this, using the example of a guest operating system that uses the flat memory model. Applications running at `%cpl=3` ran with truncated segments, and were additionally restricted by their own operating systems from accessing the guest operating system region using page protection.

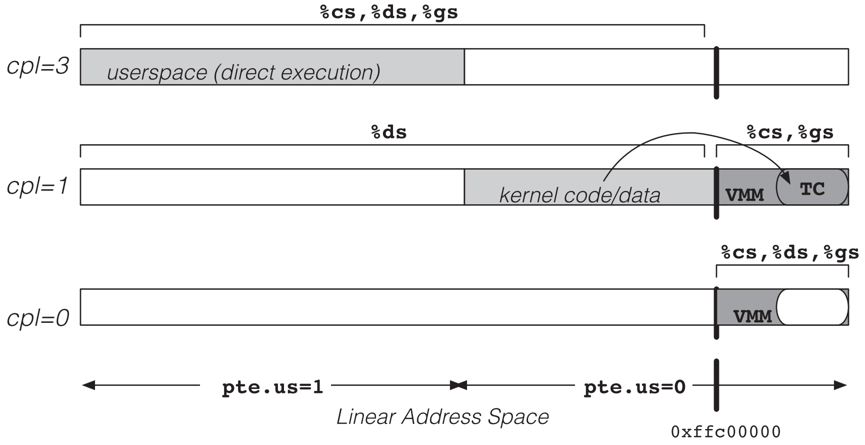


Fig. 4. Using segment truncation to protect the VMM. In this example, the virtual machine's operating system is designed for the flat memory model. Applications run under direct execution at user-level (`cpl=3`). The guest operating system kernel runs under binary translation, out of the translation cache (TC), at `cpl=1`.

When running guest kernel code via binary translation, the hardware CPU was at `%cpl=1`. Binary translation introduced a new and specific challenge since translated code contained a mix of instructions that needed to access the VMM area (to access supporting VMM data structures) and of original virtual machine instructions. The solution was to reserve one segment register, `%gs`, to always point to the VMM area: instructions generated by the translator used the `<gs>` prefix to access the VMM area, and the binary translator guaranteed (at translation time) that no virtual machine instructions would ever use the `<gs>` prefix directly. Instead, translated code used `%fs` for virtual machine instructions that originally had either an `<fs>` or `<gs>` prefix. The three remaining segments (`%ss, %ds, %es`) were available for direct use (in their truncated version) by virtual machine instructions. This solution provided secure access to VMM internal data structures from within the translation cache without requiring any additional run-time checks.

Figure 4 also illustrates the role of page-based protection (`pte.us`). Although not used to protect the VMM from the virtual machine, it is used to protect the guest operating system from its applications. The solution is straightforward: the `pte.us` flag in the actual page tables was the same as the one in the original guest page table. Guest application code, running at `%cpl=3`, were restricted by the hardware to access only pages with `pte.us=1`. Guest kernel code, running under binary translation at `%cpl=1`, did not have the restriction.

Of course, virtual machine instructions may have had a legitimate, and even frequent, reason to use addresses that fell outside of the truncated segment range. As a baseline solution, segment truncation triggered a general protection fault for every outside reference that was appropriately handled by the VMM. In Section 6.3.4, we will discuss an important optimization that addresses these cases.

Segment truncation had a single, but potentially major limitation: since it reduces segment limits but does not modify the base, the VMM had to be in the topmost portion of the address space.[5] The only design variable was the size of the VMM itself.

In our implementation, we set the size of the VMM to 4MB. This sizing was based on explicit tradeoffs for the primary supported guest operating systems, in particular

[5] Alternatively, segment truncation could be used to put the VMM in the bottom-most portion of the address space. It was ruled out as many guest operating systems use that region extensively for legacy reasons.

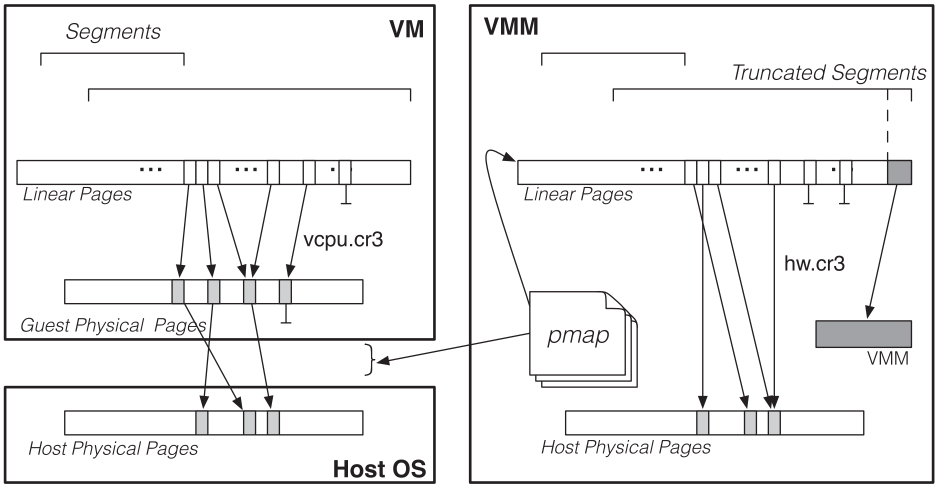


Fig. 5. Virtualizing memory. This shows mappings between virtual, linear, guest-physical, and host-physical memory, as seen and managed by the virtual machine, the host operating system, and the VMM.

Windows NT/2000. 4MB was sufficient for a practical VMM with a translation cache and other data structures large enough to fit the working set of the virtual machine, and yet small enough to minimize (but not eliminate) interference with these guest operating systems. In retrospect, there was a bit of fortuitousness in this choice, as a slightly different memory layout by a popular guest operating system would have had potentially severe consequences. Indeed, a guest operating system that made extensive use of the address space that overlapped the VMM would have been very expensive or even impractical to virtualize using segmentation-based protection.

We did evaluate alternatives that relied on paging rather than segmentation. The appeal was that one could then pick one or more portions of the linear address space that were unused by the guest, put the VMM in those locations, and conceivably even hop between locations. Unfortunately, the design alternatives all had two major downsides: first, the x86 architecture defined only one single protection bit at the page level (`pte.us`). Since we require three levels of protection (VMM, guest kernel, guest applications), the VMM would need to maintain two distinct page tables: one for guest applications and VMM, the other for guest kernel (running at `%cpl=3`!) and VMM. Second, and probably most critically, binary translators always need a low-overhead and secure mechanism to access their own supporting data structures from within translated code. Without hardware segment limit checking, we would lose the opportunity to *use* hardware-based segmentation and instead would have had to build a software fault isolation framework [Wahbe et al. 1993].

6.2.2. Virtualizing and Tracing Memory. This section describes how the VMM virtualized the linear address space that it shares with the virtual machine, and how the VMM virtualized guest-physical memory and implemented the memory tracing mechanism.

Figure 5 describes the key concepts in play within the respective contexts of the virtual machine, the host operating system, and the VMM. Within the virtual machine, the guest operating system itself controlled the mapping from virtual memory to linear memory (segmentation — subject to truncation by the VMM), and then from linear address space onto the guest-physical address space (paging — through a page table structure rooted at the virtual machine's `%cr3` register). As described in Section 6.1.4, guest-physical memory was managed as a mapped file by the host operating system, and the kernel-resident driver provided a mechanism to lock selected pages into memory, and provided the corresponding host-physical address of locked pages.

As shown on the right side of Figure 5, the VMM was responsible for creating and managing the page table structure, which was rooted at the *hardware* `%cr3` register while executing in the VMM context. The challenge was in managing the hardware page table structure to reflect the composition of the page table mappings controlled by the guest operating system (linear to guest-physical) with the mappings controlled by the host operating system (guest-physical to host-physical) so that each resulting valid `pte` always pointed to a page previously locked in memory by the driver. This was a critical invariant to maintain at all times to ensure the stability and correctness of the overall system. In addition, the VMM managed its own 4MB address space.

The pmap. Figure 5 shows the *pmap* as a central data structure of the VMM; nearly all operations to virtualize the MMU accessed it. The pmap table contained one entry per guest-physical page of memory, and cached the host-physical address of locked pages. As the VMM needed to tear down all `pte` mappings corresponding to a particular page before the host operating system could unlock it, the pmap also provided a backmap mechanism similar to that of Disco [Bugnion et al. 1997] that could enumerate the list of `pte` mappings for a particular physical page.

Memory tracing. Memory tracing provided a generic mechanism that allowed any subsystem of the VMM to register a *trace* on any particular page in guest physical memory, and be notified of all subsequent accesses to that page. The mechanism was used by VMM subsystems to virtualize the MMU and the segment descriptor tables, to guarantee translation cache coherency, to protect the BIOS ROM of the virtual machine, and to emulate memory-mapped I/O devices. The *pmap* structure also stored the information necessary to accomplish this. When composing a `pte`, the VMM respected the trace settings as follows: pages with a write-only trace were always inserted as read-only mappings in the hardware page table. Pages with a read/write trace were inserted as invalid mappings. Since a trace could be requested at any point in time, the system used the backmap mechanism to downgrade existing mappings when a new trace was installed.

As a result of the downgrade of privileges, a subsequent access by any instruction to a traced page would trigger a page fault. The VMM emulated that instruction and then notified the requesting module with the specific details of the access, such as the offset within the page and the old and new values.

Unfortunately, handling a page fault in software took close to 2000 cycles on the processors of the time, making this mechanism very expensive. Fortunately, nearly all traces were triggered by guest kernel code. Furthermore, we noticed an extreme degree of instruction locality in memory traces: for example, only a handful of instructions of a kernel modified page table entries and triggered memory traces. For example, Windows 95 has only 22 such instruction locations. We used that observation and the level of indirection afforded by the binary translator to adapt certain instructions sequences into a more efficient alternative that avoided the page fault altogether (see Section 6.3.4 for details).

Shadow page tables. The first application of the memory tracing mechanism was actually the MMU virtualization module itself, responsible for creating and maintaining the page table structures (`pde, pte`) used by the hardware.

Like other architectures, the x86 architecture explicitly calls out the absence of any coherency guarantees between the processor's hardware TLB and the page table tree. Rather, certain privileged instructions flush the TLB (e.g.,`invlpg, mov %cr3`). A naive

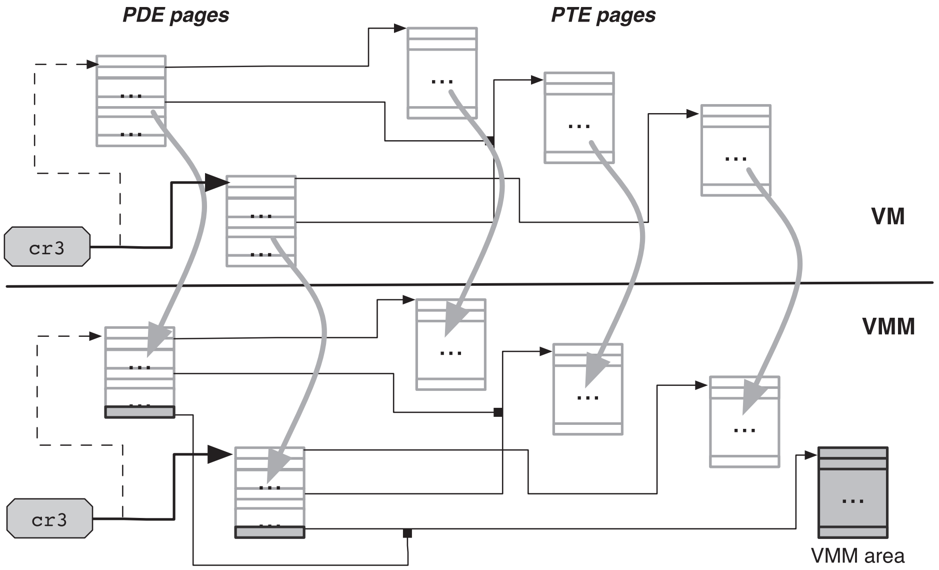


Fig. 6. Using shadow page tables to virtualize memory. The VMM individually shadows pages of the virtual machine and constructs the actual linear address space used by the hardware. The topmost region is always reserved for the VMM itself.

virtual MMU implementation would discard the entire page table on a TLB flush, and lazily enter mappings as pages are accessed by the virtual machine. Unfortunately, this generates many more hardware page faults, which are orders of magnitude more expensive to service than a TLB miss.

So instead, the VMM maintained a large cache of shadow copies of the guest operating system's pde/pte pages, as shown in Figure 6. By putting a memory trace on the corresponding original pages (in guest-physical memory), the VMM was able to ensure the coherency between a very large number of guest pde/pte pages and their counterpart in the VMM. This use of shadow page tables dramatically increased the number of valid page table mappings available to the virtual machine at all times, even immediately after a context switch. In turn, this correspondingly reduced the number of spurious page faults caused by out-of-date page mappings. This category of page faults is generally referred to as *hidden page faults* since they are handle by the VMM and not visible to the guest operating system. The VMM could also decide to remove a memory trace (and of course the corresponding shadow page), for example, when a heuristic determined that the page was likely no longer used by the guest operating system as part of any page table.

Figure 6 shows that shadowing is done on a page-by-page basis, rather than on an address space by address space basis: the same pte pages can be used in multiple address spaces by an operating system, as is the case with the kernel address space. When such sharing occurred in the operating system, the corresponding shadow page was also potentially shared in the shadow page table structures. The VMM shadowed multiple pde pages, each potentially the root of a virtual machine address space. So even though the x86 architecture does not have a concept of address-space identifiers, the virtualization layer emulated it.

The figure also illustrates the special case of the top 4MB of the address space, which is always defined by a distinct pte page, managed separately, which defines the address space of the VMM itself.

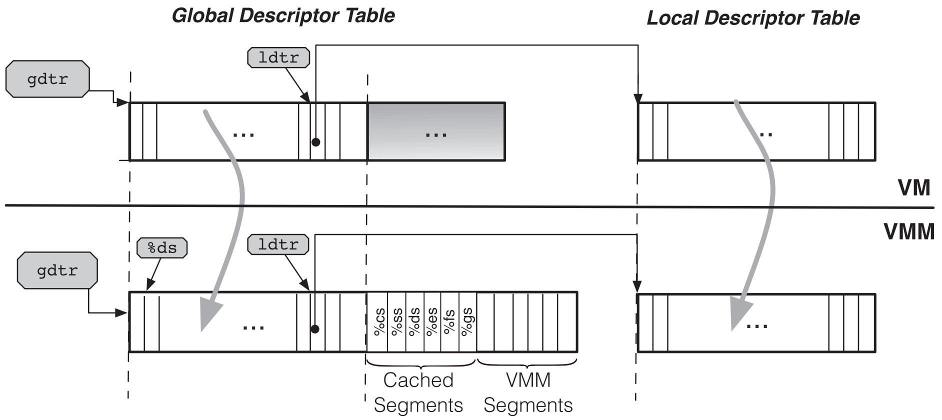


Fig. 7. Using shadow and cached segment descriptors to virtualize segmentation. Shadow and cached descriptors are always truncated by the VMM for protection.

6.2.3. Virtualizing Segment Descriptors. Segmentation played a significant role in the x86 architecture. It also played a crucial role in the design of the VMware VMM as the only isolation mechanism between the VMM and the virtual machine. Therefore, in virtualizing segments, the VMM needed to also ensure that all virtual machine segments would always be truncated, and that no virtual machine instruction sequence could ever directly load a segment used by the VMM itself. In addition, virtualizing segments was particularly challenging because of the need to correctly handle nonreversible situations (see Section 5.4) as well as real and system management modes.

Figure 7 illustrates the relationship between the VM and the VMM's descriptor tables. The VMM's global descriptor table was partitioned statically into three groups of entries: (i) shadow descriptors, which correspond to entries in a virtual machine segment descriptor table, (ii) cached descriptors, which model the six loaded segments of the virtual CPU, and (iii) descriptors used by the VMM itself.

Shadow descriptors formed the lower portion of the VMM's global descriptor table, and the entirety of the local descriptor table. Similar to shadow page tables, a memory trace kept shadow descriptors in correspondence with the current values in the virtual machine descriptor tables. Shadow descriptors differed from the original in two ways: first, code and data segment descriptors were *truncated* so that the range of linear address space never overlapped with the portion reserved for the VMM. Second, the descriptor privilege level of guest kernel segments was adjusted (from 0 to 1) so that the VMM's binary translator could use them (translated code ran at `%cpl=1`).

Unlike shadow descriptors, the six *cached descriptors* did not correspond to an in-memory descriptor, but rather each corresponded to a segment register in the virtual CPU. Cached segments were used to emulate, in software, the content of the hidden portion of the virtual CPU. Like shadow descriptors, cached descriptors were also truncated and privilege adjusted.

The combination of shadow descriptors and cached descriptors provided the VMM with the flexibility to support nonreversible situations as well as legacy modes. As long as the segment was reversible, shadow descriptors were used. This was a precondition to direct execution, but also led to a more efficient implementation in binary translation. The cached descriptor corresponding to a particular segment was used as soon as the segment became nonreversible. By keeping a dedicated copy in memory, the VMM effectively ensured that the hardware segment was at all times reversible.

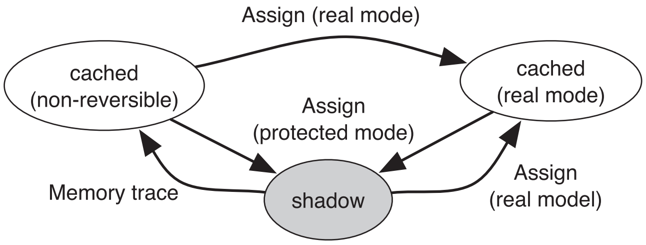


Fig. 8. State machine representation of the use of shadow and cached segments.

The decision of whether to use the shadow or cached descriptor was made independently for each segment register according to the state machine of Figure 8:

- When a segment became nonreversible following a trace write to the descriptor location in memory defining one or more CPU segments. The VMM copied the previous content of the descriptor (prior to the trace write) to the cached location(s). The cached descriptors are used from there on.
- When a segment register was assigned (e.g., `mov <sel>, %ds`) while in *real mode* or in *system management mode*. The segment becomes cached. The visible portion and the base of the cached descriptor take on the new value from the assignment. The rest comes from the descriptor used by the segment register immediately prior to the assignment (i.e., shadowed or cached), subject to truncation according to the new base.
- When a segment register was assigned while in protected mode. The VMM used the shadow segment once again.

In addition, cached segments were also used in protected mode when a particular descriptor did not have a shadow. This could occur only when the virtual machine's global descriptor table was larger than the space allocated statically for shadow segments. In practice, we mostly avoided this situation by sizing the VMM's GDT to be larger than the one used by the supported guest operating systems. Our implementation allocated 2028 shadow GDT entries, and could easily have grown larger.

This state machine had a very significant implication. Direct execution could only be used if all of the six segments registers were in a *shadow* state. This was necessary since the visible portion of each register is accessible by user-level instructions, and user-level software can furthermore assign any segment to a new value without causing a trap. Of course, binary translation offered a level of indirection that allowed the VMM to use the cached entries instead of the shadow based on the state machine of each segment. Although the state machine was required for correctness and added implementation complexity, the impact on overall performance was marginal. Indeed, all of the supported 32-bit guest operating systems generally ran user-level code with all segments in a shadowed state (thus allowing direct execution).

As discussed in the context of segment truncation, segmentation played a critical role in protecting the VMM by ensuring that shadowed and cached segments never overlap with the VMM address space (see Section 6.2.1). Equally important, one needed to ensure that the virtual machine could never (even maliciously) load a VMM segment for its own use. This was not a concern in direct execution as all VMM segments had a `dpl`≤`1`, and direct execution was limited to `%cpl=3`. However, in binary translation, the hardware protection could not be used for VMM descriptors with `dpl=1`. Therefore, the binary translator inserted checks before all segment assignment instructions to ensure that only shadow entries would be loaded into the CPU.

ALGORITHM 1: x86 Virtualization Engine Selection Algorithm

```
Input: Current state of the virtual CPU
Output: True if the direct execution subsystem may be used;False if binary translation must
        be used instead
if $!cr0.pe$ then
    return false
if $eflags.v8086$ then
    return true
if $(eflags.iopl \geq cpl)||(!eflags.if)$ then
    return false;
foreach $seg \leftarrow (cs, ds, ss, es, fs, gs)$ do
    if “seg is not shadowed” then
        return false;
end
return true
```

6.2.4. Virtualizing the CPU. As mentioned at the beginning of Section 6.2, a primary technical contribution of the VMware VMM is the combination of direct execution with a dynamic binary translator. Specifically, one key aspect of the solution was a simple and efficient algorithm to determine (in constant time) whether direct execution could be used at any point of the execution of the virtual machine. If the test failed, the dynamic binary translator needed to be used instead. Importantly, this algorithm did not depend on any instruction sequence analysis, or make any assumptions whatsoever about the instruction stream. Instead, the test depended only on the state of the virtual CPU and the reversibility of its segments. As a result, it needed only to run when at least one of its input parameters might have changed. In our implementation, the test was done immediately before returning from an exception and within the dispatch loop of the dynamic binary translator.

Algorithm 1 was designed to identify all situations where virtualization using direct execution would be challenging because of issues due to ring aliasing, interrupt flag virtualization, nonreversible segments, or non-native modes.

- Real mode and system management mode were never virtualized through direct execution.
- Since v8086 mode (which is a subset of protected mode) met Popek and Goldberg’s requirements for strict virtualization, we used that mode to virtualize itself.
- In protected mode, direct execution could only be used in situations where neither ring aliasing nor interrupt flag virtualization was an issue.
- In addition, direct execution was only possible when all segments were in the shadow state according to Figure 8.

Algorithm 1 merely imposed a set of conditions where direct execution could be used. An implementation could freely decide to rely on binary translation in additional situations. For example, one could obviously build an x86 virtualization solution that relied exclusively on binary translation.

The two execution subsystems shared a number of common data structures and rely on the same building blocks, for example, the tracing mechanism. However, they used the CPU in very different ways: Table III provides a summary view of how the hardware CPU resources were configured when the system was (i) executing virtual machine instructions directly, (ii) executing translated instructions, or (iii) executing instructions in the VMM itself. We note that the unprivileged state varied significantly with the the mode of operation, but that the privileged state (`%gdtr, %cr3,...`) did not

Table III. Configuration and Usage of the Hardware CPU When Executing (i) in Direct Execution, (ii) Out of the Translation Cache and (iii) VMM Code

VM indicates that hardware register has the same value as the virtual machine register. VMM indicates that the hardware register is defined by the VMM.

	Resource	Direct Execution	Translation Cache	VMM
Unprivileged	`%cpl`	3	1	0 or 1
	`%eflags.iopl`	<`%cpl`	≥`%cpl`	≥`%cpl`
	`%eflags.if`	1	1	0 or 1
	`%eflags.v8086`	0/1	0	0
	`%ds, %ss, %es`	shadow	shadow or cached	VMM
	`%fs`	shadow	VM `%fs` or `%gs`	VMM
	`%gs`	shadow	VMM (top 4MB)	VMM
	`%cs`	shadow	VMM (top 4MB)	VMM
	`%eip`	VM	VMM (in TC)	VMM
	Registers (`%eax`,...)	VM	VM	VMM
	`%eflags.cc`	VM	VM	VMM
Priv.	`%cr0.pe`	1: Always in protected mode		
	`%cr3`	shadow page table		
	`%gdtr`	VMM GDT with shadow/cached/VMM entries		
	`%idtr`	VMM interrupt handlers		

depend on the mode of execution. Indeed, some privileged registers such as `%gdtr` and `%idtr` were re-assigned only during the world switch. Others such as `%cr3` were re-assigned only when the VM issued the corresponding instruction.

Direct Execution Subsystem. When direct execution was possible, the unprivileged state of the processor was identical to the virtual state. This included all segment registers (inc. the `%cpl`), all `%eflags` condition codes, as well as all `%eflags` control codes (`%eflags.iopl, %eflags.v8086, %eflags.if`). However, no ring compression, or interrupt virtualization complications could arise because Algorithm 1 is satisfied. For the same reason, all segments registers were reversible and shadowed.

The implementation of the direct execution subsystem was relatively straightforward. The VMware VMM kept a data structure in memory, the `vcpu`, that acted much like a traditional process table entry in an operating system. The structure contained the virtual CPU state, both unprivileged (general-purpose registers, segment descriptors, condition flags, instruction pointer, segment registers) and privileged (control registers, `%idtr, %gdtr, %ldtr`, interrupt control flags, ...). When resuming direct execution, the unprivileged state was loaded onto the real CPU. When a trap occurred, the VMM first saved the unprivileged virtual CPU state before loading its own.

Binary Translation Subsystem. The binary translator shared the same `vcpu` data structure with the direct execution subsystem. This simplified implementation and reduced the cost of transitioning between the two subsystems.

Table III shows that the binary translator had (unlike the direct execution subsystem) more flexibility in the mapping of the virtual CPU state onto the hardware. Indeed, it loaded a carefully designed subset of the virtual CPU state into the hardware. Specifically, that subset included three segment registers (`%ds, %es, %ss`), all general-purpose registers (`%eax, %ebx, %ecx, %edx, %esi, %edi, %esp, %ebp`) as well as the condition codes within the `%eflags` register (but not the control codes of that register). Although segment registers could point to a shadow or a cached entry, the underlying descriptor always led to the expected (although possibly truncated) virtual address space defined by the guest operating system. The implication was that any instruction that operated *only* on these three segments, the general-purpose registers, or any of the condition codes could execute *identically* on the hardware without any overheads. This observation was actually a central design point of VMware's dynamic binary translator, as the majority of instructions actually met the criteria.

There were obviously expected differences between the virtual CPU state and the hardware state. As with all dynamic binary translators, the hardware instruction pointer `%eip` was an address within the translation cache and not the virtual CPU's `%eip`. The code segment `%cs` was a VMM-defined segment that spanned the full top 4MB of the address space in 32-bit mode, or a subset of it in 16-bit mode.

The binary translator treated `%fs` and `%gs` differently. According to the x86 architecture, both were used only by instructions that have an explicit segment override prefix. `%fs` was used by the binary translator as a scratch segment register to emulate instructions that have either a `<fs>` or `<gs>` prefix: the corresponding sequence first loaded `<fs>` and then used it as the prefix of the virtual machine instruction. This pattern freed up `<gs>` to be available at all times to securely access the VMM portion of the address space. This provided the binary translator the flexibility to generate instruction sequences where the protection of the VMM was enforced in hardware for all virtual machine references (by ensuring that no <%gs> prefixes were passed through the translator unmodified), and at the same time have a low-overhead mechanism to access supporting data structures or the virtual machine state.

6.3. The Binary Translator

Emulation is the big hammer of virtualization: one system can be emulated by another to provide 100% compatibility. Obviously, the price in terms of efficiency can be extremely high, for example if one uses simple interpretation techniques. More advanced forms of emulation using binary translation [Sites et al. 1993] have been proposed to provide fast machine simulation [Rosenblum et al. 1997].

A binary translator converts an input executable instruction sequence into a second binary instruction sequence that can execute natively on the target system. A *dynamic* binary translator performs the translation at run-time by storing the target sequences into a buffer called the translation cache.

At least when running on conventional processors, these fast simulators based on dynamic binary translation tend to have an intrinsic performance impact in that the translated sequence runs at a fraction of the speed of the original code. These overheads are typically due to memory accesses that require either some form of address relocation in software [Witchel and Rosenblum 1996] or the need to perform some form of software fault isolation or sandboxing. With Embra, for example, the workload slowdown was nearly an order of magnitude [Witchel and Rosenblum 1996] – very fast for a complete machine simulator, but way too slow to be the foundation for a virtual machine monitor.

VMware's dynamic binary translator had a different design point, with the explicit goal to run translated sequences with minimal slowdown or overhead. Specifically, the goal was to run instruction sequences consisting of memory movement instructions, stack operations, ALU operations, and branches (conditional and unconditional) at native or very near to native speeds.

The binary translator's essential function was to allow *any virtual machine instruction sequence, executed in any mode and system configuration of the virtual machine*, to be correctly emulated. The VMM used it as its big hammer, but used it only when necessary. Whenever possible, the VMM used the direct execution subsystem instead.

The philosophy behind VMware's binary translator was grounded in a few simple goals and requirements.

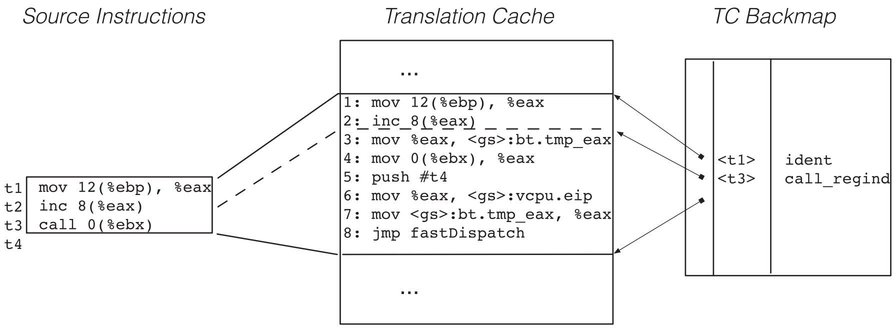
- In all cases, translate the instruction sequence to faithfully emulate the semantics of the x86 architecture.
- Remain invisible to the guest (other than possibly through changes in timing).
- Don't assume anything about the input sequence. In particular, don't assume any of the conventions that are applicable when an executable is generated by a compiler: for example, calls and returns are not always paired together, the stack is not always valid, jumps can point to the middle of an instruction. Instead, assume that the code could be malicious and buggy.
- Assume that code will change, either because the memory is reused, the code is patched, or (and this does happen) the code sequence modifies itself.
- Design and optimize for system-level code: the code will contain privileged instructions, manipulate segments, establish critical regions, trigger exceptions, initiate I/O, or reconfigure memory (virtual or linear).
- Design and optimize to run *at* system level rather than conventional simulators, which run as unprivileged processes: rely on hardware for protection of the VMM, and in particular the truncated segments; design the binary translator to work in conjunction with the VMM's interrupt and exception handlers.
- Keep it simple. Don't try to optimize the generated code sequence. Build a basic x86-to-x86 translator, not an optimizing compiler.

6.3.1. Overview. The binary translator was inspired by SimOS's Embra run-time [Witchel and Rosenblum 1996], which itself was influenced by Shade [Cmelik and Keppel 1994]. Like Embra, its components included a decoder, code generator, dispatcher, chaining logic, supporting callouts, the translation cache coherency logic, and the mechanism to synchronize state on exceptions and interrupts. When VMware Workstation first shipped, the binary translator consisted of approximately 27 thousand lines of C source code according to SLOCCount [Wheeler 2001], or approximately 45% of the total VMM line count.

The dispatch function looked up the location in the translation cache corresponding to the current state of the virtual CPU. If none was found, it invoked the translator, which first decoded a short instruction sequence (no longer than a basic block) starting at the current virtual CPU instruction pointer, generated a corresponding (but translated) instruction sequence and stored it in the translation cache. The dispatch function then transferred control to the location in the translation cache. The translated code consisted of native x86 instructions within the translation cache, and could encode calls to support routines. These *callouts* typically invoked the dispatcher again, to close the loop.

Two main data structures complemented the translation cache: a *lookup* table provided a mechanism to identify existing translations by their entry point, thereby allowing for reuse; and a *tc backmap* table provided a way to associate metadata to ranges of translated code at a fine granularity. This was used, for example, to synchronize state when the code sequence is interrupted.

The translator did not attempt to improve on the original code; we assumed that the OS developers took care in optimizing it. Rather, the goal was to minimize manipulation and disruption of the source sequence. Case in point, we translated instructions one-by-one and always maintained those guest instruction boundaries in the translated code. As described previously in Table III, translated code directly used the hardware for three segments (`%ds`, `%es`, `%ss`), all general-purpose registers and all condition code (of the `%eflags` register). Since the VMM also used the hardware to configure the virtual machine's linear address space (through shadow page tables) and enforced the isolation boundaries in hardware (through segment truncation), instructions that



```
t1  mov 12(%ebp), %eax
t2  inc 8(%eax)
t3  call 0(%ebx)
t4
```

```
...
mov 12(%ebp), %eax
inc 8(%eax)
mov %eax, <gs>:bt.tmp_eax
mov 0(%ebx), %eax
push #t4
mov %eax, <gs>:vcpu.eip
mov <gs>:bt.tmp_eax, %eax
jmp fastDispatch
...
```

Fig. 9. Example of binary translation for a simple basic block.

relied *only* on these segments, registers and flags could be copied *identically* in the translated code, where they executed at native speed.

Figure 9 provides a simple example of binary translation for a basic block of three instructions. In this example, the first two instructions are identically translated since they depend only on `%ss`, `%ebp`, `%ds` and `%eax`, which are all mapped onto hardware. The basic block ends with a memory-indirect call, wich manipulates `%eip` and therefore cannot be identically translated. The translated sequence temporarily uses `%eax` as a scratch register, pushes the return address on the stack, stores the call target address in memory (in `<gs>:vcpu.eip`), and finally jumps to a shared subroutine. Within that sequence, the `push` instruction operates on virtual machine operands by putting the return address (#t4) on its stack. The other instructions manipulate VMM operands, and in particular rely on the `<gs>` prefix to access VMM structures.

This translation of a memory-indirect call is typical of all instructions with unpredictable control transfer, which were handled in a similar manner. In these sequences, the `fastDispatch` subroutine used a specialized table indexed only by the virtual address to lookup a possible destination address within the translation cache, and called out to C code only if none was found.

Branches (conditional or unconditional) offer predictable, `%eip`-relative control flow transfers. Although branches could not be translated identically, the use of a technique called *chaining* [Cmelik and Keppel 1994] nearly eliminated their run-time overheads. Chaining converts an unconditional branch into another branch, and a conditional branch into at most two branches.

6.3.2. Using Partial Evaluation. As a baseline implementation, the translator could generate a general-purpose callout for all sensitive instructions that require some form of run-time emulation, such as privileged instructions, interrupt-sensitive instructions (`pushf`, `popf`, `cli`, `sti`), and segment assignment instructions. As callouts were very expensive (often around 1000 cycles on certain processors), it was necessary to handle as many common cases inline, that is, within the translation cache.

Unfortunately, many sensitive instructions also have very complex semantics that routinely are described through multiple pages of pseudocode in the architectural manuals [Intel Corporation 2010]. Handling all situations through inlining would be impractical. Take for example, the `cli` instruction, which clears the interrupt flag. The Intel manual specifies 8 different outcomes based on the state of CPU, and in particular based on the values of `%cr0.pe`, `%cpl`, `%eflags.iopl` and `%eflags.v8086`. However, when the virtual machine's guest operating system is running in protected mode at `%cpl=0`, the semantics are reduced to simply always clearing the `%eflags.if` bit of the virtual CPU.

To inline these sensitive instructions, the binary translator applied the partial evaluation [Jones 1996] of the virtual CPU context at translation time. This effectively shifted the complexity of emulating a sensitive instruction from run-time to translation time. The binary translator acted as a program specializer that generated different translation sequences by using specific states of the virtual CPU as static input variables (context) to the compiler. Specifically, the input context consisted of (i) the code size - 16bit or 32bit, (ii) `%cr0.pe` (protected mode or real/system management mode), (iii) `%eflags.v8086`, (iv) `%cpl` (the current privilege level), (v) `%cpl<=%eflags.iopl` (i.e., whether the virtual CPU can control the interrupt flag), and (vi) whether the guest is in the flat memory model.[6]

Partial evaluation assumes that the context at execution time matches the context used at translation time. As a result, the context was part of the key to the translation cache lookup function. As the lookup function was used by the dispatch and the chaining logic, this ensured the consistency between translation-time and run-time state when entering the translation cache, as well as throughout the execution within the translation cache. As a consequence, the same code sequence may be translated multiple times for different contexts. But in practice, this situation happened very rarely since software is normally written explicitly for only one context. Note also that the lookup key included the translation unit's virtual and physical address, because the translator emulated a complete system with multiple overlapping address spaces, and the same virtual address could actually refer to different instruction sequences.

6.3.3. Operating at System-Level. The translated code ran in an execution environment specifically optimized for it, in particular the hardware context set up by the VMM for memory virtualization and isolation. In addition:

- The translated code ran at `%cpl=1` as this (i) ensured that any trap or interrupt would trigger a stack switch and (ii) allowed access to the guest kernel address space, that is, pages with `pte.us=0`.
- In addition, the translated code ran at `%eflags.iopl=1`, which allowed it to control the interrupt flag (`cli, sti, popf`) and establish critical regions.
- While generated code always ran at `%cpl=1`, the VMM's static code used both `%cpl=0` and `%cpl=1`. The VMM switched between privilege levels lazily, as needed, and without switching stacks. Therefore, functions could be called at one level and return at another. For example, callouts from the translation cache always started at `%cpl=1`. However, if in handling a callout a function must execute a privileged instruction, it dropped to `%cpl=0`. The VMM always returned to `%cpl=1` before reentering the translation cache.
- The binary translator reserved a few distinct exception vectors for its own exclusive use. This is used to generate very compact callouts, using only two bytes of code (`int <vector>`) from the translation cache to the VMM. One of the interesting characteristics of such calls is that they lead to a stack switch and write the calling `%eip` onto the VMM stack. This was the preferred method for space-efficient callouts such as the ones terminating a basic block. To chain one block to another, the interrupt instruction was overwritten with a branch.
- The translated code could rely on the truncated segments setup by the VMM to deliver a hardware-based protection model. This eliminated the need to perform bounds-checks in software. This was discussed in Section 6.2.4.

[6] Some of the these contexts (e.g., `%eflags.v8086`) would actually indicate that direct execution is possible. The binary translator nevertheless supported them for a debugging mode that ran entirely without the direct execution engine.

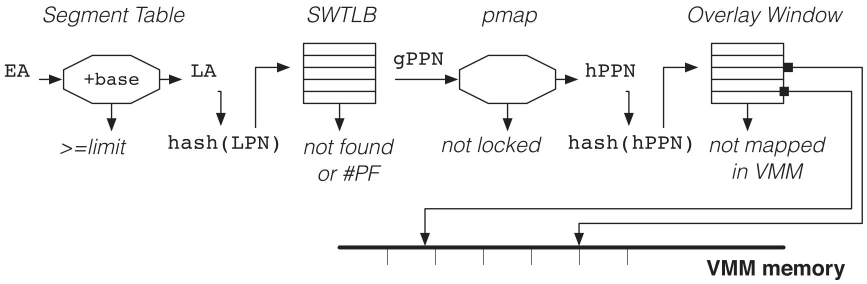


Fig. 10. Steps involved in software relocation.

6.3.4. Adaptive Binary Translation. As previously described, the binary translator relied on the system-level execution environment set up specifically for it by the VMM. We now describe another aspect of the two-way relationship between the binary translator and the VMM, where the binary translator becomes an extremely powerful tool to help avoid the cost of hardware-based protection mechanisms, and in particular the high cost of handling general protection and page faults. Adams and Agesen [2006] measured the minimal cost of handling such faults to be more than 2000 processor cycles on an Intel Pentium 4.

Adaptive binary translation refers to situations where a particular event or condition leads to the binary translator purposefully generating a new and distinct translation, optimized for this event. It was used in three situations described in more detail here:

Adaptive Translation of Memory Tracing Instructions. The first situation consisted of instructions that almost always generated page faults (such as those that modified page table entries). To avoid the high overheads for page fault handling, the adapted translation generated a callout that emulates the memory reference without causing a page fault.

As we described in Section 6.2.2, memory tracing was a central mechanism of the VMM. The page fault handler decoded the instruction, computed the effective address, and then emulated the memory access *without* directly using the original virtual or linear address (as this would create another page fault), instead by performing all of the memory translation steps in software. Figure 10 illustrates those steps: the first step converted the instruction's virtual address (segment and effective address) into a linear address by adding the segment base and checking the limit. The corresponding linear page number provided the index into a set-associative *software TLB* that mapped to guest-physical page numbers. The pmap module kept track of the guest-physical (gPPN) to host-physical (hPPN) mappings as well as the trace information. Finally, the VMM dedicated some pages of its address space (32 pages in the first implementation, or 3% of the total available address space) to an overlay window used for the temporary mappings of host pages. When the page was mapped into the overlay window, this provided an alternative de-referenceable address to the same location in memory to perform the memory access.

The implementation was optimized, with the information from the pmap and the overlay window cached directly in the software TLB structure. As a result, the cost of handling such a memory trace was dominated by the page fault itself. When code running in direct execution accessed such a page, there was little room for further optimization. Binary translation, on the other hand, offered many alternatives, and in particular the option to translate the offending instruction differently so that it would proactively call out rather than cause a page fault. We called this form of adaptive binary translation the *trace simulation* callout.

This technique leveraged the tremendous locality of trace accesses: the few instructions that caused memory traces, for example, the routines manipulating the `pte` entries, mostly always triggered memory traces. In such situations, the adapted translator dynamically generated a sequence that computed the effective address and called out to a memory reference emulation routine. This replaced a page fault with a callout (saving 800 cycles on a Pentium 4) and furthermore eliminated the cost of decoding the instruction and computing the effective address.

This form of adaptive binary translation was refined with subsequent versions of the VMware VMM. Adams and Agesen [2006] studied the cost of memory tracing under either adaptive binary translation (as described here) or a virtualization approach relying exclusively on direct execution (when enabled by hardware support): as long as memory traces remained frequent, the performance benefits of adaptive binary translation more than made up for the overhead of relying on binary translation.

Instructions that Access High Memory. Segment truncation (see Section 6.2.1) reduced the size of the virtual address spaces, and attempts to reference a virtual address outside of the truncated range caused a general protection fault. The general protection fault handler had to validate that the address was within the virtual machine's original range, and emulate the access accordingly.

Here as well, adaptive binary translation provided additional opportunities for optimization, in particular for handling the operation entirely within the translated code. Rather than logically going through the steps of Figure 10 in C code, we generated instruction sequences that performed the equivalent task within the context of the translation cache.

The approach combined two classic software engineering techniques: partial evaluation and caching. Partial evaluation ensured that the virtual CPU is in protected mode at `%cpl=0` with the flat memory model. As for caching, the implementation collapsed the various steps of Figure 10 into a direct-mapped cache that only contained already-mapped pages. This simplified the logic so that the inlined code simply (i) computed the effective address, (ii) detected the possible corner case where the memory reference would straddle two pages, (iii) performed a hash lookup in the direct mapped cache, (iv) added the page offset, (v) restored the flags, and (vi) issued the instruction with a `<gs>` prefix.

Maintaining Translation Cache Coherency. Finally, adaptive binary translation was also key for optimizations of the mechanism that ensures the coherency of the translation cache.

The basic design of any dynamic binary translator must ensure that the instruction sequences in the translation cache stay in sync with the source instruction sequences, in our case within the virtual machine. To maintain the coherency of the translation cache, the system needed to (i) detect changes, and (ii) appropriately reflect the change in the cache.

For system-level simulators, the generic solution is to leverage the memory relocation mechanisms, whether those are implemented in a hardware TLB [Dehnert et al. 2003; Ebcioglu and Altman 1997] or in a software MMU [Witchel and Rosenblum 1996]. In our system, we simply used the generic memory tracing mechanism for this purpose. Unfortunately, this baseline solution alone did not perform well when running some guest operating systems, in particular MS-DOS and Windows 95/98. Three important patterns emerged.

— *Pages that Contain Both Code and Data*. This is a form of *false sharing* at the page level that causes unnecessary traces, not a coherency violation. Although less common with 32-bit, compiled applications, legacy applications and drivers compiled or assembled as 16-bit code did intermix code and data on the same page.
— *Call-Site Patching*. This is a well-known technique that allows for distinct modules, compiled as position-independent code, to directly call each other. For example, Windows 95 used the `int <0x20>` instruction, followed by literals, to encode a cross-VxD call [King 1995]. The exception handler then patched the call site and replaced it with a regular call instruction once both VxD modules were in memory.
— *Self-Modifying Code*. Although generally frowned upon, this technique was nonetheless found in practice in a number of areas, including games and graphics drivers hand-written in assembly code.

The binary translator addressed each of these three patterns distinctly [Bugnion 1998]. It kept track of the exact set of addresses used as input to the translator, at the byte level, so that a memory trace implied an actual coherency problem only when the modification range overlaps with the input bytes.

When a true coherency violation did occur, the binary translator had to ensure that that translation would never be used again. Embra [Witchel and Rosenblum 1996] solved this in a brute force way by simply flushing the entire translation cache. Inspired by Embra, the first versions of the VMware binary translator took the same approach. Support for selective invalidation was introduced in a subsequent version, three years after the first shipment [Agesen et al. 2010].

In some cases, handling memory traces caused by false sharing resulted in significant performance costs. To minimize these overheads on selected pages, we removed the trace and instead relied on adaptive binary translation to ensure coherency. Specifically, the translator generated a prelude for each basic block from those pages that verified at run-time that the current version of the code was identical to the translation-time version. The prelude consisted of a series of comparisons between the translation-time instruction byte stream (encoded in operands) and the current instruction byte stream in the memory of the virtual machine. A policy framework, driven on a page-by-page basis via some history counters, helped optimize between the two lesser evils: to use the memory trace method when the code was frequently executed, and the adaptive binary translation when writes to that page dominated the performance profile. In practice, early implementations of the policy framework mainly avoided pathological decisions to ensure a stable system.

Call-site patching occurred frequently in Windows 95/98 operating systems where an `int <0x20>` instruction was replaced with a `call` instruction about 2000 times during boot. As a workload-specific optimization, we chose to simply never cache instruction sequences that included an `int <0x20>` instruction.

True self-modifying code was another matter. During the development, we identified patterns where the code that constituted the inner loop of an algorithm is repeatedly updated by the outer loop. Specifically, the outer loop only modified the *immediate* or *displacement* operands of some instructions, that is, the outer loop updated constants of instructions of the inner loop. We adapted the prelude technique to (i) update the constant values (immediate and displacement) of these modified instructions in the translation cache, and (ii) verify that all other bytes in the basic block were unmodified.

6.3.5. Precise Exception Handling. The full-system nature of the VMware binary translator had a number of additional implications on the core design, in particular around exception handling. For example, the instruction pointer `%eip` found in an exception frame (i.e., on the VMM's stack) can point to an address within the translation cache. But to emulate an exception, the VMM needs to reconstruct the virtual machine's `%eip`. Similarly, a translated instruction sequence may temporarily use general-purpose registers as scratch registers.

We will use the example of Figure 9, introduced earlier, to illustrate how the binary translation system handles synchronous and asynchronous exceptions. Synchronous exceptions, for example, page faults, general-protection faults, division-by-zero, can only occur on instructions that manipulate virtual machine operands. In this example, only instructions (1), (2), (4), or (5) are in the category; the other instructions only manipulate VMM operands, and are guaranteed to never fault. On the other hand, the code always runs with interrupts enabled, so an asynchronous exception can occur anywhere in the sequence.

The binary translator handled both the synchronous and the asynchronous case. The first, but very important, insight is that these two types of exceptions can be handled totally differently. For synchronous exceptions, the x86 architecture mandated that no partial side effects remain from the faulting instruction. The state needed to be rolled back to the start of the virtual instruction, as if each instruction executed as an atomic transaction. For asynchronous exceptions, the solution merely guaranteed that any side effects from the asynchronous exception would be applied to the virtual machine state *immediately after* the completion of the current virtual machine instruction.

Figure 9 illustrates the role of the *TC backmap* to handle synchronous exceptions: The TC backmap structure was a table, filled at translation time, and used by the exception handlers to efficiently map any location in the translation cache to (i) its corresponding virtual machine `%eip`, (ii) which *template* (distinct code pattern generated by the translator) was used to generate the code, and (iii) the offset within that template. Each distinct translation template had its own custom callback, which was responsible for reconstructing the state. For example, the *ident* template was used for all instructions that were merely copied (without any transformation) from the virtual machine to the translation cache. Its corresponding callback was a `nop`, since the processor guaranteed atomicity in the case of exceptions. In contrast, the callback for register indirect calls (*call_regind*) had to (i) assert that the fault occurred at offset (4) or (5) only, and (ii) restore `%eax` from its scratch location in memory. The first shipping VMware binary translator had 26 distinct templates and 26 corresponding custom handlers, leaving the complexity of handling state rollback to the developer of each template.

Luckily, asynchronous exceptions, that is, external I/O interrupts, provided much more flexibility. Their first and immediate consequence is a world switch back to the host operating system context (see Section 6.1.2). But as a common and important side effect, the VMM could raise the virtual CPU's interrupt line. The dynamic binary translator provided precise guarantees that any side effects of an asynchronous exception would be applied no later than the *next* virtual machine instruction boundary. Each custom callback could either rollback the state (when possible) or identify all possible exit points of the translated code sequence. These exit points were temporarily replaced with a callout instruction, so that execution could resume for a few cycles until the next virtual machine instruction boundary. The software interrupt instructions were inserted idempotently, so the system also safely handled multiple (asynchronous) interrupts that fired within the same virtual instruction.

7. EVALUATION

We evaluate the VMware virtual machine monitor in four main areas: the compatibility of the virtual machine (Section 7.1), the isolation properties of the virtual

Table IV. Handling Sensitive Instructions in Direct Execution (DE) and Binary Translation (BT)

Instruction	Description	DE	BT	Comments
`sgdt, sidt, sldt`	Copy `%gdtr, %idtr, %ldtr` into a register	x	√	According to Intel: available (but not useful) to applications
`smsw`	legacy pre-386 instruction that returns parts of `%cr0`	x	√	Legacy instruction only
`pushf, popf`	Save/restore condition code and certain control flags	√	√	Interrupt virtualiztion problem - avoided by design by using BT in all sensitive situations
`lar, verr, verw`	Load Access Rights, Verify a segment for reading/writing	√	√	Level sensitive instruction comparing `%cpl` with the segment's `dpl`. However, DE only occurs at `%cpl=3` and the `dpl` adjustment is only from 0 to 1, so userspace instructions will execute correctly, and system code is always in binary translation.
		x	√	If the VM defines a GDT large enough to hold non-shadowed descriptors, the instruction will behave incorrectly and clear `zf`. However, this never happens with any of the supported guest operating systems.
`lsl`	Load Segment Limit	x	√	Incorrect limit is returned on truncated segments
`pop <seg>, push <seg>, mov <seg>`	Manipulate segment registers	√	√	Ring aliasing problem - avoided by design by using BT in all sensitive situations (ring aliasing or non-reversibility)
`fcall, longjmp, retfar, str`	Far call, long jump or far returns to a different privilege level (via a call gate)	√	√	Use BT to avoid ring aliasing problem; don't allow call gates in hardware descriptor tables
`int <n>`	Call to interrupt procedure	√	√	In DE, always triggers an interrupt or a fault, as expected

machine (Section 7.2), the classification of VMware Workstation according to the classic literature on virtualization (Section 7.3), and finally on the performance of the system (Section 7.4).

7.1. Compatibility

Robin and Irvine [2000] published a widely quoted, detailed analysis of the Intel Pentium's ability to support a secure virtual machine monitor. In particular, they identified sensitive, yet unprivileged instructions that made the processor non-virtualizable, at least using exclusively a trap-and-emulate approach. Table IV lists these instructions and discusses how our solution addresses the issues raised by the authors. Our approach fell short in only two minor cases.

(1) Instructions that access privileged state at user-level without causing faults. These instructions were `sidt`, `sgdt`, `sldt`, and `smsw`. They were sensitive in direct execution since they return information configured by the VMM, not the guest operating system. Fortunately, Intel's own architecture manual described these instructions as "useful only by operating-system software" [Intel Corporation 2010]. The `sgdt` instruction was used by a few applications to explicitly test whether the software was running in a virtual machine.

(2) The `lsl (load segment limit)` instruction reported the incorrect limit because of segment truncation. To our knowledge, this limitation was never identified as the root cause of a field defect.

Both of these gaps only appeared during direct execution. The binary translator could handle these instructions correctly, but the performance impact of abandoning direct execution would have been unacceptable. Fortunately, neither limitation had a practical consequence.

In presenting Intel's Virtualization Technology (VT-x), Uhlig et al. [2005] also described a number of limitations to software-only virtualization in the absence of architectural support. In their analysis, these limitations would include issues such as ring aliasing, address space compression, interrupt virtualization, access to privileged state, hidden segment registers, and non-native modes. VMware Workstation addressed all of these issues, with the exception of the two minor caveats listed previously.

7.2. Isolation and Security

Robin and Irvine [2000] also raised some credibility issues around VMware's claims, in particular in the areas of security and isolation. Their paper was written soon after the release of the first VMware products, at a time when no disclosures on its internals were publicly available.

The isolation of each virtual machine is strong by design, having access only to virtualized resources, and never to the underlying physical names. The design of the VMM furthermore enforced the boundary between the virtual machine and the VMM by using hardware segmentation. The routines that manipulate the hardware descriptor tables are well isolated in a module that is easy to reason about. VMware performed independent reviews of these properties, and VMware Workstation has been used for security-sensitive applications such as the NSA's NetTop [Meushaw and Simard 2000].

7.3. Formal Classification of Virtualization Systems

The classic literature on virtualization provides a theoretical framework that can classify a VMM according to two orthogonal considerations: whether the architecture is virtualizable, and how that VMM manages resources. First, Popek and Goldberg [1974] provided the theoretical framework for virtualizability, and also described the concept of a *hybrid virtual machine monitor* and of a hybrid virtualization criterion.

> THEOREM. A hybrid virtual machine (HVM) monitor may be constructed for any conventional third generation machine in which the set of user sensitive instructions are a subset of privileged instructions.
> In order to argue the validity of the theorem, it is first necessary to characterize the HVM monitor. The difference between a HVM monitor and a VMM is that, in the HVM monitor, *all* instructions in virtual supervisor mode will be interpreted. Otherwise, the HVM can be the same as the VM monitor [...].

The VMM nearly met the requirements of an HVM as defined in that theorem, if one substitutes virtual supervisor mode with Algorithm 1, substitutes the interpreter with the dynamic binary translator, and allows the caveats of Section 7.1.

Second, in this thesis, Goldberg [1972] proposed the following classification for VMMs.

> — Type I. the VMM runs on a bare machine
> — Type II. the VMM runs on an extended host, under the host operating system
>
> [...] In both Type I and Type II VCS, the VMM creates the VCS computing environment. However in a Type I VCS, the VMM on a bare machine must perform the system's scheduling and (real) resource allocation. Thus, the Type I VMM may include such code not specifically needed for a VCS. In a Type II VCS, the resource allocation and environment creation functions for VM are more clearly split. The operating system does the normal system resource allocation and provides a standard extended machine.

Here as well, the definition must be adapted to the more modern environment. The concept of a *standard extended machine* does not exist in modern operating systems, and was replaced with the world switch mechanism. Nevertheless, VMware Workstation is clearly a type II VMM, in that the host operating system is responsible for all resource management decisions.

7.4. Performance

A number of papers have been published that describe various aspect of the end-to-end performance of VMware's products. Sugerman et al. [2001] published the first academic paper on VMware Workstation, focusing on the performance characteristics of network I/O, and using Workstation 2.0 as the baseline for their study. They quantify the cost of a world switch operation to 4.45 μs on a now vintage 733 MHz Pentium III desktop. Their study shows that significant performance optimizations can be achieved by reducing the frequency of world switches. This can be done by handling certain device emulation logic within the VMM (and in some cases within the translation cache itself), and by batching RPCs to the VMX to initiate I/O.

In their paper that studies the evolution of VMware's x86 virtual machine monitor, Agesen et al. [2010] quantify some of the key characteristics of VMware's approach. First, the cost of running the binary translator, that is, the logic that generates executable code inside the translation cache, is negligible. A near worst-case workload is one that boots and immediately shuts down a guest operating system, since it offers little reuse opportunities and runs a lot of kernel initialization routines that only execute once. Their analysis shows that the translator itself takes less than 5% of the overall execution time and generates a total of 28,000 16-bit basic blocks and 933,000 32-bit basic blocks, for an average cost of 3 μs per translation.

In their study comparing software VMM and hardware-assisted VMMs, Adams and Agesen [2006] quantify the overheads of the VMware VMM (without hardware support) to be no more than 4% when running the CPU-intensive SPECint 2000 and SPECjbb2005 benchmarks, but substantially higher—in the 25%+ range—for system-intensive benchmarks such as compilation and web serving workloads. Such slowdowns are explained by a combination of factors, such as the cost of synchronizing shadow page tables, and the overheads of going through the host operating system for all I/O access.

Despite these positive results, the performance of guest operating systems was at times limited by a variety of idiosyncratic interactions that stressed overheads in our VMM, for example, the use of self-modifying code and the false sharing of traced executable code with unrelated data or the frequent use of the top 4MB portion, both discussed in Section 6.3.4. Other examples included the extensive use of 16-bit code under binary translation, false sharing of segment descriptor tables with unrelated data, the emulation of I/O devices that polled certain I/O ports heavily, etc. As we encountered these situations, we made significant engineering investments to mitigate them, but by no means were we able to fully eliminate all of our performance cliffs.

8. LESSONS LEARNED

The first shipping VMware product was the result of an intense 15-month development effort. Some of the decisions made during that initial phase had a long-term impact on the quality, performance, stability, and evolution of the product. In this section, we highlight some of the key lessons learned.

8.1. Benefits of the Hosted Architecture

There is a well-known development productivity gap between user-space software and kernel software: user-space processes start quickly, support a broad range of debugging tools, generally execute deterministically, and fail in predictable ways, for example by dumping core, at which point all resources are automatically reclaimed by the operating system. In contrast, developing a kernel or kernel module often leads to long boot sequences, subtle concurrency bugs, resource leaks after failures, and sometimes the overall corruption of the system.

So, in developing the infrastructure for the VMM, we created an environment that could provide developers with productivity comparable to user-space development. The kernel-resident device driver took care of automatically reclaiming all resources, even in the case of a corrupted VMM. As a result, developers could debug, crash, and restart virtual machines nearly as if they were dealing with regular processes. We also added to the VMM a flexible logging environment, automatic VMM core dumps on panics, and built-in VMM profiling capabilities.

Still, we voluntarily kept as much code outside of the VMM as was possible. All of the CPU and memory virtualization logic was within the VMM, but most of the device models (front-end and back-end) ran as part of the VMX process in user-space. Only the performance-sensitive portions of the front-ends would eventually be moved into the VMM. Although this partitioning of functionality was required because of the severe address space limitations of the VMM, it also turned out to be beneficial by limiting the complexity of VMM.

8.2. The VMM - More than the Sum of Its Parts

We began building the VMM according to standard software engineering practices of modularity and abstraction: each major component—the binary translator, the MMU handling, segment truncation logic, the trap-and-emulate subsystem, etc.—was originally implemented as an independent module with a well-defined boundary and interface. However, we quickly realized that these modules were not so easy to decouple. Rather, they interacted with each other in subtle ways that were critical for performance. One basic example: we introduced caches that crossed multiple abstraction boundaries, for example, to emulate instructions that accessed high memory as shown in Figure 10. More fundamentally, we realized that the modules all significantly informed each others' behavior, impacting the overall design. For example, the page fault handler could directly signal the binary translator to adaptively retranslate a block of instructions in order to avoid further page faults. Similarly, segmentation management was simultaneously responsible for enabling very low overhead execution of binary translated code, maximizing circumstances where direct (trap-and-emulate) execution was possible, and protecting the isolation of the VMM. The consequence of this was some lack of modularity and independence in solving problems and making seemingly local implementation decisions.

Our view of the binary translator shifted in response to this recognition, too. At first, we were primarily concerned of the cost of running translated code. We quickly realized, however, that it also was a source of performance benefits: the adaptive translation process it enabled became the key optimization mechanism in the VMM.

8.3. Guest-Specific Optimizations

One of the most important principles of systems design is to deal effectively with common-case situations, without paying too high of a penalty when the uncommon case hits. Like any system, many parts of the VMM exhibited a significant amount of spatial and/or temporal locality. For example, the use of 16-bit translation caches had tremendous temporal locality: all workloads use it during bootstrap, but many workloads totally stop accessing it after a while. We leveraged that temporal observation to free up all of the space allocated to 16-bit caches when they became unused, which allowed us to grow the 32-bit translation cache. Such optimizations were oblivious to the precise nature of the guest operating system.

A VMM is only as useful as the operating system that it runs. In a universe where operating systems are designed to be architecturally friendly to virtualization, a balanced design would include adjustments to the operating system to help its own execution in a virtual machine. This was the case in mainframe systems [Creasy 1981], in Disco, and more recently in the context of paravirtualization [Barham et al. 2003]. At VMware however, we explicitly focused on supporting unmodified operating systems, including both current and older versions of these operating systems. As a result, the product contained a number of optimizations that are very guest-specific: we chose pragmatically to deliver a VMM that met the efficiency criteria for a selected number of supported guest operating systems. Here are some examples.

- 4MB ended up being the “lucky number” for the size of the VMM. 4MB is large enough to hold the working set of all data structures mapped in memory (and in particular the translation cache), but small enough to only marginally interfere with guest operating systems. It turns out that certain guests, for example, the Windows 2000 family, access the next lower 4MB of memory much more frequently.
- Specifically, the top 4 MB of memory is used to store certain data structures in Windows 2000 (and 32-bit successors), and in particular the Windows Kernel Processor Control Region (KPCR), which is part of the Windows Hardware Abstraction Layer, and hardcoded in two pages at `0xffdff000` and `0xffe00000` [Solomon and Russinovich 2000].[7] Although the inline relocation algorithms described in Section 6.3.4 provided relief from a first-order performance overhead, we chose to further optimize the handling of these very specific instructions that accessed the KPCR. We dedicated two pages of the VMM’s address space to permanently map these two KPCR pages — clearly an extremely ad-hoc approach. This enabled further optimizations of the translated code. For example, instructions with a fixed effective address could be emulated as a single instruction.
- Contemporary versions of Linux used `int <0x80>` to make system calls. In a generic solution, that is, trap-and-emulate, all `int` instructions from `%cpl=3` trigger a general protection fault. As an optimization, the VMM configured the corresponding entry 0x80 in the interrupt table to directly transfer control to an optimized routine already running at `%cpl=1` to quickly transition into binary translation mode.

[7] We learned only in the preparation of this article of the precise nature of the KPCR data structure that was located in high memory. During the development phase, that piece of information was irrelevant; instead, we only observed that a significant fraction of the VMM time was spent emulating accesses to these two pages in memory.

— We special-cased the `int <0x20>` instructions in the translator, and never cached them in the translation cache (see Section 6.3.4). This eliminated the subsequent coherency conflict resulting from the cross-VxD call-site patching by Windows 95/98.
— We had to special-case a single 4MB range of Windows 95/98's linear address space located at `0xc1380000` as the guest operating system made an incorrect architectural assumption with respect to TLB coherency. Specifically, Windows 95/98 appears to modify `pte` mappings and then use the mappings without flushing the TLB (or at least that entry in the TLB), effectively assuming that they were not present in the TLB. This assumption was no longer acceptable in a virtualized setting with shadow page tables having much larger capacity than an actual TLB. We resorted to disabling certain MMU tracing optimizations for that memory range when Windows 95/98 was the selected guest operating system.

We see from these examples that we sometimes had to apply guest-specific techniques to achieve our goal of providing hardware-level virtualization. With time, the engineering team added additional guest-specific optimizations, driven primarily by the need to adapt to innovations released in newer operating systems, for example, to support the Linux `vsyscall` page efficiently. At the same time, the ongoing innovations and re-factoring of the code base had at times the positive side-effect of eliminating certain guest-specific optimizations, for example, the Windows 95/98 `int <0x20>` special case was removed once the binary translator could efficiently invalidate selected translations.

8.4. Dealing with Architectural Ambiguity

In theory, matching hardware's behavior is merely a matter of reading the appropriate manual or datasheet and applying the specified semantics. In practice, particularly with an architecture as rich as x86—both in history and diversity—many significant details end up undocumented or under-documented. And, of course, guest operating systems often end up relying upon particular implementations of these quirks.

There are numerous cases of this in the instruction set architecture itself. For example, how to deal with the `don't care` bit of particular instructions (e.g., `lar`) or the `rep` prefixes of instructions that don't expect them, such as `nop`. Websites such as `www.sandpile.org` [Ludloff 1996] proved extremely useful. As a last resort, we wrote specific unit tests to reverse-engineer the actual implementation of certain instruction sequences on real hardware. We compiled these test vectors into a bootable floppy image that ran both on a real computer and in a virtual machine.

Most of the other corner cases manifested in the "chipset", or virtual motherboard, which covered both the memory controller and the core I/O controller (including all the oldest, most legacy functionality). One well-known historical example is the handling of unaligned accesses in real mode at address `0xf000:0xffff`, which was so important to legacy MS-DOS software that it led to the introduction of a feature in all x86 platforms in which the 20th address line in physical memory could be selectively controlled by software. Entire book chapters have been devoted to this unusual A20 line [Chappell 1994], which helped us quite a bit as we tried to understand its semantics. In general, we employed an assortment of tactics to narrow down the appropriate behavior: trying to read between the lines in datasheets; combing books and online articles about PCs; reading and debugging the BIOS source we had licensed; and disassembling and tracing the execution of guest operating system drivers. In a handful of cases, we wrote little boot loaders and MS-DOS programs to verify very specific details.

Beyond the documentation ambiguities, we also ran into actual CPU-specific implementation issues during our development. In one example, we identified a situation where segment truncation caused a specific instruction (`rep cmps`) to take a general protection fault, with the CPU reporting the fault on the wrong instruction (the location following the one that took the fault). In another instance, the behavior of a specific instruction (`sidt`, in 16-bit protected mode) proved to be different from that described in every version of the processor manual we could find.

8.5. The Importance of Soundness

As previously described, with our guest-driven implementation strategy, we were able to build a successful system while leaving some architectural features unimplemented. We were consistently less successful when we tried to make simplifications or take shortcuts in features we did implement. If a guest operating system used a feature of the architecture, it generally exercised it comprehensively enough that failing to handle corner cases would either panic the VMM or crash the guest. We experienced two cases of this with our early handling of I/O devices: overlapping I/O port registration and level triggered interrupt lines. The VMM initially panicked if multiple handlers claimed the same I/O port ranges. Sloppy operating system code, however, sometimes very briefly violated that assumption during initialization and we had to support the possibility (it never accessed overlapping ports though, which would likely have unpredictable results on real hardware). Similarly, the architecture allows hardware interrupts that are edge-triggered (one-time) or level-triggered (repeated until the device driver clears them). Our implementation was edge-only at first, but a guest can program PCI devices to share interrupt lines, which relies upon level-triggering. Before we understood this, we were mystified by the intermittent flakiness that only happened with certain combinations of devices and guests, for example, Windows 2000 with both networking and USB support.

We also attempted—and similarly regretted—taking shortcuts in the dynamic binary translator. For example, early versions of the system sometimes ignored the direction flag that should control string instructions (`%eflags.df`). This worked in nearly all cases, as software traditionally sets the flag immediately before using it. Nevertheless, until we handled it correctly, the system suffered from rare silent misbehaviors. Also, the early binary translator used an incomplete instruction interpreter to emulate memory traces; this caused a steady stream of bugs until we introduced a comprehensive interpreter.

9. THE EVOLUTION OF VMWARE WORKSTATION

The technology landscape has obviously changed dramatically in the decade following the development of the original VMware Virtual Machine Monitor.

In Section 6, we described the design and implementation of the original VMware Workstation—from inception up to version 2.0 of the product. Following that release, significant portions of the system were rewritten to improve performance and robustness, and to adapt to new software and hardware requirements. Over the years, the VMM has been the subject of constant refinements and improvements, with an investment of well over a hundred man-years. Agesen et al. [2010] provide a great overview of the evolution of the system beyond the original design described in this article. The authors highlight in particular the intrinsic tension between the various paths involved in tuning the MMU, further improvements in the adaptive binary translation framework, for example, to accelerate APIC TPR register access, and the design and implementation of the symmetric multiprocessor VMM.

The hosted architecture (Section 6.1) is still used today for state-of-the-art interactive products such as VMware Workstation, VMware Player, and VMware Fusion, the product aimed at Apple OS X host operating systems. The world switch, and its ability to separate the host operating system context from the VMM context, remains the foundational mechanism of VMware's hosted products today. Although the implementation of the world switch has evolved with the years, primarily to adjust to new modes in the processor (64-bit long mode) and with a new class of nonmaskable interrupts, the fundamental idea of having totally separate address spaces for the host operating system and the VMM remains valid today.

The original VMM protection design, based exclusively on segment truncation (Section 6.2.1), is still largely in use today by current VMware products, at least when their VMM relies on a dynamic binary translator rather than hardware-based virtualization. Segmentation is still used to enforce protection when running in binary translation. However, the protection mechanisms were later augmented to additionally rely on a combination of segmentation and paging in direct execution [Agesen and Sheldon 2004]. This work was motivated by the emergence of the new Linux NPTL library [Drepper and Molnar 2003], which used negative offsets from a segment register to access thread-local structures, resulting in general protection faults due to segmentation even though the linear address is within the virtual machine range.

VMware's approach to implementing memory virtualization (Section 6.2.2) has evolved substantially since the original VMware VMM. Although memory virtualization consists of a number of mechanisms that are individually relatively easy to understand, the policy heuristics that control and tune them ended up being very complex and unfortunately somewhat guest- and workload-specific. Over the years, each version of VMware's products has improved the experience across an ever-broadening set of workloads and requirements (including SMP virtual machines) [Agesen et al. 2010]. Adaptive binary translation played an important role in reducing the cost of memory tracing, even to the extent of nearly negating the benefits of hardware support for CPU virtualization in the first generation of processors [Adams and Agesen 2006]. Today, the challenge of composing address spaces in software has been removed with the relatively recent availability of hardware support for nested paging [Bhargava et al. 2008].

The general organization of the segment virtualization subsystem (Section 6.2.3), and in particular the static separation between shadow, cached and VMM segments, is still in use when the VMM relies on dynamic binary translation. The approach described here for the original VMM ensured the synchronous update of shadow descriptors (using the tracing mechanism). We augmented it later with a mechanism that allowed for the deferred update of shadow descriptors [Lim et al. 2000], motivated by guest operating systems that co-located segment descriptors on the same page as other data structures, leading to a large amount of false sharing.

Finally, the engineering team largely rewrote the binary translator itself following VMware Workstation 2. Although the structures and mechanisms described in Section 6.3 remained mostly present, the code was largely re-implemented and the new design formalized the notion of a compiled code fragment, and introduced selective invalidations of translations to replace the original all-or-none approach [Agesen et al. 2010]. This helped improve the adaptive binary translation policies used throughout the system substantially. Adaptive binary translation was expanded with new uses, including emulating the chipset. Return instructions were handled using a novel approach [Agesen 2006]. Cross-instruction optimizations were introduced to reduce cases where `%eflags` had to be spilled into the VMM area. Finally, with the introduction of 64-bit support (a.k.a. *long mode*), the binary translator took advantage of the larger register file to use long mode in all situations, even to run 16-bit and 32-bit code [Chen 2009].

Hardware-assisted virtualization such as Intel VT-x and AMD-v were introduced in two phases. The first phase, starting in 2005, was designed with the explicit purpose of eliminating the need for either paravirtualization or binary translation [Uhlig et al. 2005]. With this first phase of hardware support, a VMM could be built entirely using trap-and-emulate. Unfortunately, that system would still require some form of memory tracing to shadow page tables. And memory tracing, which normally relies on the adaptive binary translator for performance, would become much more expensive. Adams and Agesen [2006] discussed the tradeoffs of the two approaches, and concluded that the software approach remains superior, at least in the absence of MMU support for virtualization.

The second phase of hardware support, *nested page tables*, was introduced by AMD in 2007 [Bhargava et al. 2008] and similarly by Intel in 2008 (as *extended page tables*). Combined with CPU virtualization support, it effectively eliminated the most common use for memory tracing. Furthermore, nested paging simplifies the implementation of memory tracing itself. Today, VMware's VMM mostly uses a hardware-based, trap-and-emulate approach whenever the processor supports both virtualization and nested page tables.

The emergence of hardware support for virtualization had a significant impact on VMware's guest operating system-centric strategy. In the original VMware Workstation, the strategy was used to dramatically reduce implementation complexity at the expense of compatibility with the full architecture. Today, full architectural compatibility is expected because of hardware support. The current VMware guest operating system-centric strategy focuses on performance optimizations for selected guest operating systems.

Hardware support for virtualization has also helped improve the hardware-independent encapsulation property of live virtual machines. In particular, the FlexMigration [Intel Corporation 2008] architectural extension allowed a VMM to control the bits reported by the CPUID instruction, even in direct execution. This allows live migration of virtual machines across different processor generations by restricting the reported processor features on both hosts to match.

10. RELATED WORK

This section mirrors Section 6, which describes the design and implementation.

10.1. Hosted Architectures

Connectix Virtual PC (later Microsoft VirtualPC), which first launched in June 2001 [Connectix Corporation 2001], was probably the closest analog to VMware Workstation. A commercial product, it supported unmodified guest operating systems and ran as a separate product on top of Windows host operating systems. Today in 2012, that technology is used to provide Windows XP compatibility in Windows 7 ("Windows XP mode with Virtual PC") [Microsoft Corporation 2009]. Another commercial product, Parallels [Parallels Corporation 2006] provides the same feature on Apple OS X host operating systems, similar to VMware Fusion. Little has been published on either system.

Linux KVM [Kivity 2007] is the most relevant open-source hosted VMM available today, as it incorporates a number of concepts of system-level co-location directly into Linux. KVM was designed from the ground up to assume architectural support for virtualization. This eliminates the need to have a world switch to avoid interference from the host operating system. As a result, the VMM can be incorporated into the kernel-resident driver, which itself is part of the Linux kernel. Both VMware Workstation and KVM have a concept of a user-space proxy, which is responsible for most device emulation. Both rely on the host operating system to schedule resources.

User-Mode Linux [Dike 2001] and UMLinux [Sieh and Buchacker 2002] each run a guest Linux operating system kernel directly on the CPU in user space. This requires adding special signal handling to support virtual interrupts, relinking the guest kernel into a different range of addresses, and paravirtualizing the interrupt flag. This scheme is similar to the one used in SimOS's original direct execution mode [Rosenblum et al. 1995] on MIPS systems. Using such an approach, the performance is often determined by the implementation of certain low-level mechanisms by the host operating system. King et al. [2003] used UMLinux to study how changes in the host operating system could further improve performance. In contrast, VMware's system-level co-location approach dramatically reduces the frequency of interaction with the host operating system.

The VMware hosted architecture is still used today for all of VMware's interactive products running on x86 desktops and laptops. It has also been ported to the ARM architecture to offer virtualization solutions on mobile platforms and smartphones [Barr et al. 2010].

10.2. ESX Server

In addition to VMware Workstation, VMware developed ESX Server [Waldspurger 2002], a *type I* bare-metal architecture that effectively eliminates any dependency on a host operating system for I/O, resource management and scheduling. ESX Server was architected to separate virtualization from global resource management, allowing both products to share a nearly unmodified VMM. By controlling all global resources, ESX Server could provide richer controls and stronger performance isolation guarantees. By handling I/O directly within the hypervisor, ESX Server could reduce the virtualization overheads of I/O intensive workloads.

Both products exist today, each focused on a different market. ESX Server is used for datacenter deployments, where I/O performance and fine-grain resource management are most relevant. The hosted architecture of VMware Workstation, Fusion and Player is used on desktops and laptops, where co-existence with an existing installation and peripheral diversity are most relevant.

10.3. Other Hypervisors

A hypervisor is generally defined as a virtualization solution that runs directly on the hardware without depending on a separately managed host operating system; it is often used synonymously with Goldberg's *type I VMM* architecture.

The ESX Server hypervisor is the combination of the vmkernel and the VMM. The vmkernel is responsible for global resource management as well as all I/O operations; in particular, the vmkernel natively runs the performance critical device drivers.

The Xen hypervisor [Barham et al. 2003] has evolved from its root in paravirtualization to offer full virtualization on CPUs with hardware support. Xen differs from ESX Server in that the hypervisor only virtualizes and manages the CPU and memory resources, and delegates I/O operations, including the device drivers, to a privileged Linux virtual machine called *dom0*. Microsoft Hyper-V [Microsoft Corporation 2008] shares the same architecture, with a Windows root partition replacing Xen's dom0.

10.4. VMM Protection and Design

Segmentation, and specifically the use of truncated segments, is commonly used to provide protection between execution domains. Xen used a similar approach [Barham et al. 2003]. More recently, Vx32 [Ford and Cox 2008] and the Google Native Client [Yee et al. 2010] have used segmentation to build lightweight sandboxing solutions. Both systems are limited to run only 32-bit code in the sandbox as Intel's x86-64 implementations have removed segmentation support for 64-bit code. The VMware VMM faced the same issues in supporting 64-bit guest operating systems when using the binary translator [Agesen et al. 2010].

The VMware VMM has a general memory tracing mechanism used to maintain the coherency of the shadow page tables, the segment tables, and the translation cache. Xen originally relied on explicit registration of MMU pages (which required guest modifications) to avoid the complexity of shadowing, but subsequently supported shadow page tables. The need for memory tracing has been removed with the combined introduction of hardware support for virtualization in the CPU (VT-x) and in the MMU (via Nested Page Tables [Bhargava et al. 2008]).

The VMware VMM relies on the combination of a trap-and-emulate direct execution subsystem and a binary translator, with very frequent switches driven by the state of the virtual CPU. Most other virtualization solutions rely exclusively on a single approach: Xen, KVM, and User Mode Linux use only direct execution, and require guest changes or hardware support. QEMU [Bellard 2005] uses binary translation exclusively. SimOS included multiple CPU simulation engines, one based on direct execution, one on binary translation, and one on instruction set simulation. However, unlike the VMware VMM, the switch between execution engines in SimOS was a heavyweight operation.

10.5. The Binary Translator

The VMware binary translator was designed to only support x86-to-x86 binary translation, and to run only as part of a system-level VMM. This led to a compact and simple design. QEMU [Bellard 2005] is a dynamic binary translation framework capable of running unmodified operating systems and applications. QEMU offers an extensive set of cross-architectural capabilities. In contrast, the VMware VMM is simplistic in that it merely copies without modifications most instructions of a given sequence. Both systems support chaining and overlapping address spaces. QEMU emulates the MMU in software, and has a mechanism similar to SimOS's Embra [Witchel and Rosenblum 1996] to deal with pages that contain both code and data. Like VMware, it supports exceptions and interrupts, but uses a different mechanism based on code re-translation (in the synchronous case) and identifying the next chaining point (in the asynchronous case), rather than the next instruction boundary.

The VMware binary translator, QEMU, and Embra were all designed to run on existing processors that were never specifically designed to support efficient dynamic binary translation. Other dynamic binary translators such as DAISY [Ebcioglu and Altman 1997] and the Transmeta Crusoe [Dehnert et al. 2003] took an opposite approach and co-designed hardware and software for optimal performance. For example, both ensure that the architectural state of the VM can be mapped efficiently on the hardware state, with plenty of additional register resources available to the translated code, allowing for numerous optimizations. In contrast, the VMware translator is a 32bit-to-32bit translator with zero free incremental resources. Also, both systems have hardware support to enable speculative execution and recovery from exceptions. In contrast, the VMware translator relies on hand-written logic to undo partial side-effects of a faulting instruction.

The evaluation of Crusoe, which was aimed at running the same x86-based operating systems as VMware such as Windows, identified many of the guest operating system specific issues that we encountered in the development of the VMware binary translator. For example, Crusoe had dedicated hardware to handle pages that share code and data. That extra logic reduced the number of page faults present during a Win95/98 boot by more than 50×. In addition, Cruose used a software mechanism to self-check translation that is similar to VMware's prelude. Crusoe also handled self-modifying code by adding a level of indirection in the translated code, whereas the VMware translator used the prelude to update the translation.

11. CONCLUSION

In setting out to introduce the old idea of virtual machines to the commodity x86 platform in 1998, we knew upfront that the processor architecture was of daunting complexity and considered not to be virtualizable; and that the commodity computing industry had disaggregated into an ecosystem, with different vendors controlling the computers, CPUs, peripherals, operating systems, and applications, none of them asking for virtualization. We chose to build our solution independently of all of these vendors, effectively inserting our technology between them. This choice had significant implications:

- To offer a familiar user experience without requiring factory pre-install by a computer vendor, we relied on a hosted architecture, in which VMware Workstation installed like any regular application, on top of an existing operating system. This also allowed the virtualization layer to use the host operating system, and its pre-existing device drivers.
- To allow the virtualization layer to co-exist with the user's existing host operating system, we developed a special kernel-resident device driver that interacted in all respects with the host operating system as a standard device driver. This approach enabled VMware Workstation to appear to the user like a normal application, and to run on top of commercial host operating systems, alongside other applications. At the same time, the primary function of that driver was to implement the world switch mechanism, which enabled a system-level VMM to run free of the constraints of the host operating systems.
- To handle the lack of virtualization support in x86, the VMM combined a traditional trap-and-emulate approach to virtualization with a fast dynamic binary translator. The VMM selected the appropriate technique dynamically based on the state of the virtual CPU. The use of segmentation-based protection eliminated most of the overheads traditionally associated with system-level dynamic binary translation. The use of partial evaluation and adaptive binary translation further helped improve performance by reducing the frequency of processor exceptions. The resulting approach met the compatibility, performance and isolation criteria for virtual machine monitors.
- To abstract the large diversity of peripherals, we chose to emulate each virtual device through a split front-end component (visible to the virtual machine) and back-end component (interacting with the host operating system). This additionally gave VMware virtual machines the key attribute of hardware-independent encapsulation.
- To handle the daunting complexity of the x86 architecture, we applied a guest operating system-centric strategy to our development effort, only virtualizing the subset of possible architectural combinations necessary to run relevant operating systems.

Although hardware and operating systems have changed substantially since 1998, many of the contributions laid out in this article are still in use today. The hosted architecture still provides state-of-the-art interactive virtualization products on desktops and laptops, and even in upcoming mobile devices. Although x86 CPUs have added hardware support for virtualization, binary translation is still used by VMMs for legacy CPU modes as well as in places where adaptive binary translation can provide a performance win.

The commercial success of virtualization in general, and of VMware in particular, has transformed the industry: according to IDC [2009], deployments of virtual machines have been exceeding in volume those of physical hosts since 2009. Virtualization is as popular today as it was on mainframes decades ago.

ACKNOWLEDGMENTS

VMware was started with a simple vision to bring virtualization to the x86 industry. We express our most sincere gratitude to everyone who helped transform this vision into reality. We also thank Ole Agesen, David Mazieres, Diego Ongaro, John Ousterhout, and Carl Waldspurger for their careful review of drafts of this article.

REFERENCES

ADAMS, K. AND AGESEN, O. 2006. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XII)*. 2–13.

AGESEN, O. 2006. Binary translation of returns. In *Workshop on Binary Instrumentation and Applications*. 7–14.

AGESEN, O. AND SHELDON, J. W. 2004. Restricting memory access to protect data when sharing a common address space. U.S. Patent 7,277,999.

AGESEN, O., GARTHWAITE, A., SHELDON, J., AND SUBRAHMANYAM, P. 2010. The evolution of an x86 virtual machine monitor. *Operating Systems Review 44*, 4, 3–18.

AMD CORPORATION. 1998. Network Products: Ethernet Controllers Books 2.

BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T. L., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. 2003. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*. 164–177.

BARR, K., BUNGALE, P. P., DEASY, S., GYURIS, V., HUNG, P., NEWELL, C., TUCH, H., AND ZOPPIS, B. 2010. The VMware mobile virtualization platform: Is that a hypervisor in your pocket? *Operating Systems Review 44*, 4, 124–135.

BELLARD, F. 2005. QEMU, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*. 41–46.

BHARGAVA, R., SEREBRIN, B., SPADINI, F., AND MANNE, S. 2008. Accelerating two-dimensional page walks for virtualized systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIII)*. 26–35.

BIRRELL, A. AND NELSON, B. J. 1984. Implementing remote procedure calls. *ACM Trans. Comput. Syst. 2*, 1, 39–59.

BUGNION, E. 1998. Dynamic binary translator with a system and method for updating and maintaining coherency of a translation cache. U.S. Patent 6,704,925.

BUGNION, E., DEVINE, S., GOVIL, K., AND ROSENBLUM, M. 1997. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Trans. Comput. Syst. 15*, 4, 412–447.

BUGNION, E., DEVINE, S. W., AND ROSENBLUM, M. 1998. System and method for virtualizing computer systems. U.S. Patent 6,496,847.

CHAPPELL, G. 1994. *DOS Internals*. Addisson-Wesley.

CHEN, Y.-H. 2009. Dynamic binary translation from x86-32 code to x86-64 code for virtualization. M.S. thesis, Massachusetts Institute of Technology.

CMELIK, R. F. AND KEPPEL, D. 1994. Shade: A fast instruction-set simulator for execution profiling. In *Proceedings of the 1994 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. 128–137.

COMPAQ, PHOENIX, INTEL. 1996. BIOS Boot Specification, v1.0.1. `http://www.phoenix.com/resources/specs-bbs101.pdf`.

CONNECTIX CORPORATION. 2001. Connectix Virtual PC for Windows (Press Release). Retrieved on the Internet Archive's Wayback Machine.

CREASY, R. 1981. The origin of the VM/370 Time-sharing system. *IBM J. Res. Develop 25*, 5, 483–490.

CUSTER, H. 1993. *Inside Windows NT*. Microsoft Press.

DEHNERT, J. C., GRANT, B., BANNING, J. P., JOHNSON, R., KISTLER, T., KLAIBER, A., AND MATTSON, J. 2003. The Transmeta Code Morphing - Software: Using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proceedings of the 1st IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 15–24.

DEVINE, S. W., BUGNION, E., AND ROSENBLUM, M. 1998. Virtualization system including a virtual machine monitor for a computer with a segmented architecture. U.S. Patent 6,397,242.

DIKE, J. 2001. User Mode Linux. In *Proceedings of the 5th Annual Ottawa Linux Symposium (OLS)*.

DREPPER, U. AND MOLNAR, I. 2003. The Native POSIX Thread Library for Linux. RedHat White Paper.

EBCIOGLU, K. AND ALTMAN, E. R. 1997. DAISY: Dynamic compilation for 100% architectural compatibility. In *Proceedings of the 24th International Symposium on Computer Architecture (ISCA)*. 26–37.

FORD, B. AND COX, R. 2008. Vx32: Lightweight user-level sandboxing on the x86. In *Proceedings of the USENIX Annual Technical Conference*. 293–306.

GELSINGER, P. 1998. Personal Communication (Intel Corp. CTO).

GOLDBERG, R. P. 1972. Architectural principles for virtual computer systems. Ph.D. thesis, Harvard University, Cambridge, MA.

GOLDBERG, R. P. 1974. Survey of virtual machine research. *IEEE Computer Magazine 7*, 6, 34–45.

IDC. 2009. Server Virtualization Hits Inflection Point as Number of Virtual Machines to Exceed Physical Systems in 2009 (Press Release).
`http://www.idc.com/about/viewpressrelease.jsp?containerId=prUK21840309`.

INTEL CORPORATION. 2008. Intel Virtualization Technology FlexMigration Application Note. Tech. rep.

INTEL CORPORATION. 2010. *Intel64 and IA-32 Architectures Software Developer's Manual Volume 2 (2A and 2B)*.

INTERNATIONAL STANDARDS ORGANIZATION. 1988. Information processing – Volume and file structure of CD-ROM for information interchange. In *ISO 9660-1988*.

JONES, N. D. 1996. An introduction to partial evaluation. *ACM Comput. Surv. 28*, 3, 480–503.

KING, A. 1995. *Inside Windows 95*. Microsoft Press.

KING, S. T., DUNLAP, G. W., AND CHEN, P. M. 2003. Operating system support for virtual machines. In *USENIX Annual Technical Conference, General Track*. 71–84.

KIVITY, A. 2007. KVM: The Linux virtual machine monitor. In *Proceedings of the 2007 Ottawa Linux Symposium (OLS)*. 225–230.

LIM, B.-H., LE, B. C., AND BUGNION, E. 2000. Deferred shadowing of segment descriptors in a virtual machine monitor for a segmented computer architecture. U.S. Patent 6,785,886.

LUDLOFF, C. 1996. Sandpile: The world's leading source for technical x86 processor information. `http://www.sandpile.org`.

MEUSHAW, R. AND SIMARD, D. 2000. NetTop: Commercial technology in high assurance applications. *NSA Tech Trend Notes 9*, 4.

MICROSOFT CORPORATION. 2008. Windows Server 2008R2 Hyper-V.
`http://www.microsoft.com/en-us/server-cloud/windows-server/hyper-v.aspx`.

MICROSOFT CORPORATION. 2009. Windows XP Mode and Windows Virtual PC.
`http://www.microsoft.com/windows/virtual-pc/`.

NELSON, M., LIM, B.-H., AND HUTCHINS, G. 2005. Fast transparent migration for virtual machines. In *Proceedings of the USENIX Annual Technical Conference, General Track*. 391–394.

PARALLELS CORPORATION. 2006. Parallels Desktop for the Mac.
`http://www.parallels.com/products/desktop/`.

POPEK, G. J. AND GOLDBERG, R. P. 1974. Formal requirements for virtualizable third generation architectures. *Commun. ACM 17*, 7, 412–421.

ROBIN, J. S. AND IRVINE, C. E. 2000. Analysis of the Intel Pentium's ability to support a secure virtual machine monitor. In *Proceedings of the 9th Conference on USENIX Security Symposium,* vol. 9.

ROSENBLUM, M., HERROD, S. A., WITCHEL, E., AND GUPTA, A. 1995. Complete computer system simulation: The SimOS approach. *IEEE Parall. Distrib. Tech. 3*, 34–43.

ROSENBLUM, M., BUGNION, E., DEVINE, S., AND HERROD, S. A. 1997. Using the SimOS machine simulator to study complex computer systems. *ACM Trans. Model. Comput. Simul. 7*, 1, 78–103.

SIEH, V. AND BUCHACKER, K. 2002. UMLinux – A versatile SWIFI tool. In *Proceedings of the 4th European Dependable Computing Conference (EDCC)*. 159–171.

SITES, R. L., CHERNOFF, A., KIRK, M. B., MARKS, M. P., AND ROBINSON, S. G. 1993. Binary translation. *Commun. ACM 36*, 2, 69–81.

SOLOMON, D. A. AND RUSSINOVICH, M. E. 2000. *Inside Microsoft Windows 2000* 3rd Ed. Microsoft Press.

SUGERMAN, J., VENKITACHALAM, G., AND LIM, B.-H. 2001. Virtualizing I/O devices on VMware Workstation's hosted virtual machine monitor. In *Proceedings of the USENIX Annual Technical Conference, General Track*. 1–14.

UHLIG, R., NEIGER, G., RODGERS, D., SANTONI, A. L., MARTINS, F. C. M., ANDERSON, A. V., BENNETT, S. M., KÄGI, A., LEUNG, F. H., AND SMITH, L. 2005. Intel virtualization technology. *IEEE Comput. 38*, 5, 48–56.

WAHBE, R., LUCCO, S., ANDERSON, T. E., AND GRAHAM, S. L. 1993. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP)*. 203–216.

WALDSPURGER, C. A. 2002. Memory resource management in VMware ESX server. In *Proceedings of the 5th Symposium on Operating System Design and Implementation (OSDI)*.

WALDSPURGER, C. A. AND ROSENBLUM, M. 2012. I/O virtualization. *Commun. ACM 55*, 1, 66–73.

WHEELER, D. A. 2001. SLOCCount. http://www.dwheeler.com/sloccount/.

WHITAKER, A., SHAW, M., AND GRIBBLE, S. D. 2002. Scale and performance in the Denali isolation kernel. In *Proceedings of the 5th Symposium on Operating System Design and Implementation (OSDI)*.

WITCHEL, E. AND ROSENBLUM, M. 1996. Embra: Fast and flexible machine simulation. In *Proceedings of the 1996 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. 68–79.

YEE, B., SEHR, D., DARDYK, G., CHEN, J. B., MUTH, R., ORMANDY, T., OKASAKA, S., NARULA, N., AND FULLAGAR, N. 2010. Native client: A sandbox for portable, untrusted x86 native code. *Commun. ACM 53*, 1, 91–99.

Received April 2012; accepted July 2012