Dynamic Binary Translation from x86-32 code to x86-64 code for Virtualization

by

Yu-hsin Chen

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
June 2009

Author . . . .
Department of Electrical Engineering and Computer Science
May 18, 2009

Certified by . . . .
Robert T. Morris
Associate Professor
Thesis Supervisor

Certified by . . . .
Jeffrey W. Sheldon
VMware
Thesis Supervisor

Accepted by . . . .
T. P. Orlando
Chairman, Department Committee on Graduate Theses

Dynamic Binary Translation from x86-32 code to x86-64 code for Virtualization

by

Yu-hsin Chen

Submitted to the Department of Electrical Engineering and Computer Science
on May 18, 2009, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

The goal of this project is to enhance performance of virtual machines and simplify the design of the virtual machine monitor by running 32-bit x86 operating systems in x86-64 mode. In order to do so, 32-bit operating system binary code is translated into x86-64 binary code via "widening binary translation"; x86-32 code is "widened" into x86-64 code.

The main challenge of widening BT is emulating x86-32 legacy segmentation in x86-64 mode. Widening BT's solution is to emulate segmentation in software. Most of the overhead for software segmentation can be optimized away.

The main contribution of widening BT is simplification of the VMM, which reduces the human cost of maintaining a complicated VMM. Widening BT also improves performance of 32-bit guest operating systems running in virtual machines and demonstrates the independence of virtual machines from physical hardware. With widening BT, legacy hardware mechanisms like segmentation can be dropped. Therefore widening BT reduces hardware's burden of backwards-compatibility, encouraging software/hardware co-design.

Thesis Supervisor: Robert T. Morris
Title: Associate Professor

Acknowledgments

This work is an extension of the virtual machine monitor developed by VMware's Monitor Group and would have been impossible without their extensive help and support. Particular thanks to Ole Agesen, Ross Knippel, and Jeffrey Sheldon for advice and careful code reviews. I'd also like to thank my thesis advisor Robert Morris for asking insightful questions and helping me smooth out confusing concepts.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A virtual machine is a tightly isolated software container that can run its own operating system and applications as if it were a physical computer [11]. The operating system that runs inside of a virtual machine is called a "guest". A virtual machine monitor (VMM) has access to the physical hardware and exports an image of the underlying hardware resources to the guest operating system; this image is called the virtual machine. Virtual machines are widely used in industry for consolidation, availability, load balancing, and compatability.

VMware focuses on virtualizing the x86 ISA. Many 32-bit operating systems such as Windows and Linux have been developed for 32-bit x86 hardware (x86-32). When the x86 ISA was extended to 64-bit address space and 64-bit registers (the extended architecture is called x86-64), 64-bit operating systems were developed to run on the new 64-bit CPUs. 64-bit CPUs have a x86-32 hardware mode to support legacy 32-bit code. Code that runs in x86-64 hardware mode is called "64-bit code" and have a 64-bit codesize, while code that runs in x86-32 hardware mode is called "32-bit code" and have a 32-bit codesize.

Before this work, in order to export an image of x86-32 hardware to a 32-bit guest operating system, VMware's VMM ran in x86-32 hardware mode—even if the underlying CPU was 64-bit and capable of running in x86-64 mode. The VMM also needs to run in x86-64 mode in order to support 64-bit guest operating systems. Therefore, VMware's VMM can run in both x86-32 mode and x86-64 mode.

Maintaining a VMM that can run in both x86-32 mode and x86-64 mode is complicated.

The goal of this thesis is to dynamically translate 32-bit guest operating system binary code into x86-64 mode binary code that can run in x86-64 mode under a 64-bit VMM. This binary translation process is called "widening BT"; x86-32 mode code is "widened" into x86-64 mode code. Widening BT allows the VMM to take advantage of 64-bit CPUs; running both 32-bit and 64-bit guest operating systems in x86-64 mode simplifies the VMM and enhances performance.

The main benefits of x86-64 mode are the large 64-bit address size and 16 general purpose registers (GPR). The large 64-bit address space makes the VMM faster by allowing allocation of more address space for VMM datastructures. The 64-bit address space allows the VMM and 32-bit guest to reside at disjoint addresses. There are 16 64-bit GPRs in x86-64 mode, twice as many as in x86-32 mode. The 8 extra registers in 64-bit mode can be used by the VMM for intermediate computation even when executing translations, when the guest register values are live. Widening BT also greatly optimizes virtual system calls by using special 64-bit structures such as the interrupt stack table (IST).

While x86-64 architecture is an extension of x86-32 architecture, widening BT must deal with some differences between them. The primary challenge is to emulate x86-32 segmentation (a memory protection mechanism) efficiently in x86-64 mode, which does not support it. Widening BT's solution is to emulate segmentation in software. Most of the overhead for software segmentation can be optimized away.

Most of the benefits of running in x86-64 mode are reaped by the VMM. Maintaining the VMM is much simpler with widening BT because the VMM does not need to run in both x86-32 mode and x86-64 mode. The human cost savings of maintaining a VMM that only runs in one hardware mode (x86-64 mode) is far more important than the small performance gains from running in x86-64 mode. On the other hand, it is difficult for widening BT to dynamically optimize guest code to take advantage of the 64-bit address space and extra GPRs; additional hardware support may help the BT infrastructure to dynamically optimize guest code.

Widening BT demonstrates the degree of independence virtual machines have from physical hardware by allowing 32-bit guest operating systems to run in x86-64 mode. With widening BT, legacy hardware mechanisms like segmentation can be dropped, especially considering that modern operating systems use segmentation infrequently. Therefore widening BT reduces hardware's burden of backwards-compatibility.

1.1 Organization of paper

Chapter 1 explains the motivations behind implementing widening BT. Chapter 1 describes differences between x86-64 and x86-32 mode and explains the benefits that can be gained from translating x86-32 code to x86-64 code (widening BT). At the end of chapter 1 is an overview of the design decisions for widening BT.

Chapter 2 reviews everything you need to know about x86 in order to understand this paper. Chapter 2 explains the difference between x86 hardware modes x86-32 and x86-64 in detail. In x86 terminology, x86-32 corresponds to "legacy and compatibility modes" and x86-64 corresponds to "long mode".

Chapter 3 provides background on x86 virtualization. Chapter 3 explains why virtualizing x86 is difficult and how binary translation overcomes these difficulties. This chapter introduces the VMware BT infrastructure before widening BT was implemented.

Chapter 4 gives examples of actual widened (x86-64) translations of common x86-32 instructions. Chapter 4 demonstrates how to generate x86-64 translations that are almost identical to the x86-32 source while preserving operand size and address size. Chapter 4 also demonstrates how to generate more complicated translations for stack operations and control flow.

Chapter 5 discusses the main challenge of widening BT: emulating x86-32 mode segmentation in x86-64 mode, which has very limited segmentation. Widening BT's solution is to use software segmentation checks. Chapter 5 explains why software segmentation is used and how to eliminate most of the software segmentation checks.

Chapter 6 explains how guest system calls are implemented and why widening BT leads to faster guest system calls.

Chapter 7 compares the performance of 32-bit guest operating systems running in x86-64 mode under VMM64 (made possible with widening BT) with the performance of 32-bit guest operating systems running in x86-32 mode under VMM32.

Chapter 8 evaluates widening BT's performance gains and simplification of the VMM. Chapter 8 analyzes lessons learned from implementing widening BT.

1.2 Why widening binary translation

"Widening" binary translation from x86-32 to x86-64 mode greatly simplifies the virtual machine monitor programming model. It lets VMware run x86 guests from all modes in x86-64 mode under a x86-64 mode VMM. Furthermore, a x86-64 mode VMM can take advantage of x86-64 performance benefits that a x86-32 mode VMM cannot. Obvious benefits are the huge 64-bit address space and twice as many general purpose registers. Section 6.2 even demonstrates how special x86-64 structures such as the interrupt stack table (IST) allow faster system calls.

1.2.1 Simpler virtual machine monitor

VMware's virtual machine monitor can run in both x86-32 and x86-64 modes and switch between them. VMware's VMM's x86-32 mode is named VMM32 and the VMM's x86-64 mode is named VMM64. VMM64 and VMM32 are *modes* of one VMM. VMM64 and VMM32 are not two separate virtual machine monitors themselves. VMM64 and VMM32 are not separate VMMs because they share almost all their state.

x86 is difficult to virtualize bcause certain privileged state is visible to user code and some privileged instructions do not trap when run at user level. Therefore VMware uses binary translation to separate the virtual state of the CPU from the physical state of the CPU. VMware's VMM binary translated x86-64 mode code in VMM64 and x86-32 mode code in VMM32. The resulting translations are run in the same codesize as the source instructions (see Figure 1-1).

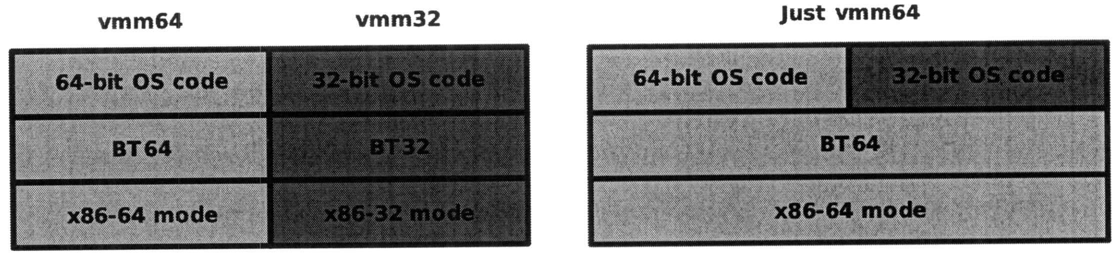

Figure 1-1: BT64 emits x86-64 translations while BT32 emits x86-32 translations. Shown on left: Before widening BT, translations were run in the same codesize as the source instructions and the VMM ran in both x86-32 and x86-64 modes. Shown on right: After widening BT, all translations are x86-64 code and the VMM runs solely in x86-64 mode.

The dependencies and interactions between the VMM's x86-64 (VMM64) and x86-32 (VMM32) modes and address spaces greatly complicate the programming model of the VMM. Detangling the VMM's x86-64 and x86-32 mode will result in cleaner code and performance gains. 32-bit guests currently run in x86-32 mode, while 64-bit guests run in x86-64 mode. On a 64-bit CPU, switching between the two modes of the VMM to run 32-bit and 64-bit guests is expensive and complicated: each switch consumes about twenty thousand cycles to save and restore a large amount of state.

Optimizations to *avoid* restoring some state in expensive VMM mode transitions further complicate the VMM code. Immense amounts of careful reasoning and logic are required in order to determine when state must be saved and restored. Removing VMM mode VMM32 greatly simplifies reasoning about the VMM. Simplifying the VMM to save developers' time is the main benefit of widening BT.

Widening BT will allow VMware to get rid of the VMM32 monitor mode. In less than 10 years, the majority of x86 CPUs in use will be 64-bit CPUs that support x86-64, allowing VMware to abandon VMM32 and always use widening binary translation to run 32-bit guests under VMM64.

1.2.2 Large 64-bit address space

Widening BT allows the VMM to take advantage of x86-64's large 64-bit address space. By allowing the VMM and guest operating system to have disjoint address spaces, widening BT prevents performance cliffs caused by competition between the VMM and guest for addresses in the same address space. With a large 64-bit address space, the VMM also has more room to allocate large VMM data structures.

VMM and guest can have disjoint address spaces

In order to minimize the cost of transitioning between accessing guest data structures and VMM data structures, the VMM and the guest need to share an address space. Widening BT will prevent performance cliffs caused by competition between 32-bit guests and the x86-32 VMM mode (VMM32) for addresses in the same 32-bit address space. When a 32-bit guest runs under x86-32 mode, it expects access to all 32-bit addresses, but the VMM must reside in the same address space. To protect the VMM, access by the guest to the addresses occupied by the VMM must be intercepted by expensive page protection traps. When the VMM runs in x86-32 with 32-bit guest, the VMM allots itself a tiny budget of 4MB of address space to hide in. In x86-64 mode, by contrast, the VMM can reside anywhere in the 64-bit address space above the low 4GB. If 32-bit guests were translated to run in x86-64 mode, they could be allotted a full 32-bit subspace disjoint from the addresses used by the VMM, rendering expensive traps unnecessary. By mapping the 32-bit guest at the very bottom 4G of the 64-bit address space, translated guest instructions can directly access guest memory without changing the guest's virtual address in emitted code (see Figure 1-2).

By mapping the guest address space in the bottom 4G and the VMM at the top of the 64-bit address space, widening BT can also take advantage of hardware features to restrict the guest to the low 4G of address space, so the VMM can be protected without traps, and in more contexts than before.

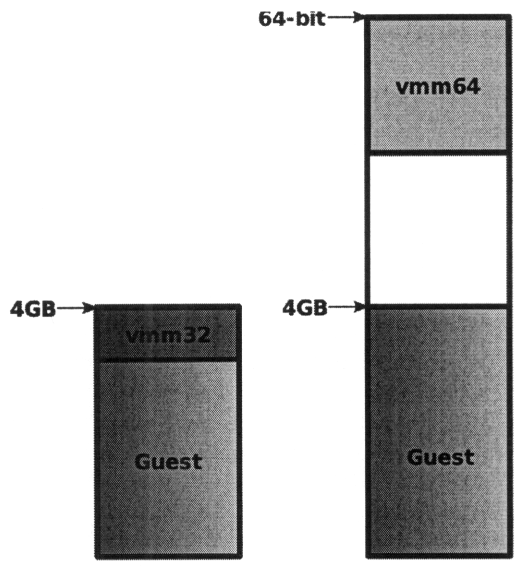

Figure 1-2: Comparison between VMM32 in 32-bit address space and VMM64 in 64-bit address space. 32-bit address space is on the left while 64-bit address space is on the right. 64-bit address space allows a larger VMM that does not overlap with the guest address space.

More space to allocate large VMM datastructures

Because VMM32 and the guest share an address space, when using a 32-bit address space VMM32 was forced to use as little of the address space as possible in order to maximize the space available to the guest. VMM64 will have much more space available in a large 64-bit address space and therefore run faster. For example, modern 32-bit guests have a hard time fitting their working sets into the tiny 3MB of the 32-bit translation cache (TC). A translation cache is a cache of guest code translations. Once guest code is translated, the translation is placed in the cache and executed many times. With a 64-bit address space, the 32-bit translation cache can be much larger. More address space can be allotted to large VMM datastructures which cannot fit in the 4MB VMM address space under x86-32 mode. Without a large enough address space, the VMM has to temporarily map in what it needs, which causes considerable overhead. Figure 1-2 depicts how the x86-64 VMM (VMM64) can be larger in the 64-bit address space.

1.2.3 Sixteen 64-bit registers

Aside from having a large 64-bit address space, x86-64 mode has 16 64-bit registers while x86-32 mode has 8 32-bit registers. The 8 extra registers in x86-64 mode are extremely useful to the VMM.

For example, when executing translations of 32-bit guest code, most physical registers are bound to guest register values. This makes binary translation easier; the translations can be nearly identical to the original guest code. However, by keeping 32-bit guest's registers "live" when executing translations, 8 registers are used—if the guest is running under VMM32 in x86-32 mode, all 8 available GPRs are used (Figure 1-3). If VMM32 wants to emit software checks into translations, the software checks will have to save the live guest register contents to memory in order to free some registers up for computation, and at the end of software checks, the clobbered registers need to be restored from memory.

On the other hand, with widening BT VMM64 can emit software checks that just use the 8 extra x86-64 mode registers `%r8` through `%r15` (Figure 1-3), avoiding many memory accesses from saving and restoring registers for computation. The translations emitted by VMM64 rarely need 8 registers for computation, so a few registers can be used to cache oft-used values such as segment base of `DS` (when non-flat) or zero. One can also envision a binary translator that adaptively recompiles hot paths through the code to take advantage of 16 registers, though this feature was not implemented.

1.3 Challenge

Aside from solving challenges common to all implementations of virtualization (such as de-privileging guest operating system code and maintaining a shadow copy of guest state), widening BT needs to deal with the differences between x86-32 and x86-64.

Widening BT's primary challenge is emulating x86-32 segmentation in x86-64

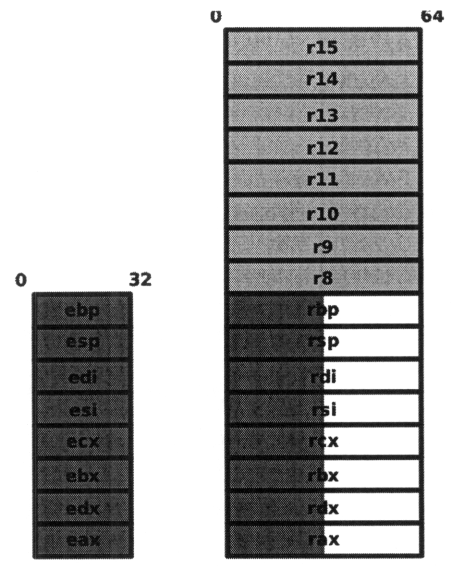

Figure 1-3: Comparison between 8 GPRs in x86-32 mode (on left) and 16 GPRs in x86-64 mode (on right). Red represents registers that hold live guest 32-bit values when executing translations. Green represents extra registers that VMM64 can use.

mode. Intel x86-64 mode has no segment limits while under AMD x86-64 mode, only some segments may have limits (cite manuals for details). Widening BT emits code to perform software segmentation checks on data memory accesses. Most of these software segmentation checks can be optimized. Widening BT eliminates the dynamic cost of the software segmentation checks for modern operating systems with flat (base 0 and limit the maximum 4G) code, data, and stack segments. Chapter 5 demonstrates how to emulate segmentation in detail.

1.4 The big picture

With widening BT, 32-bit guests are run in x86-64 mode with a x86-64 VMM (what was known as VMM64); with widening BT, there is no need for separate x86-32 and x86-64 VMM modes. The 32-bit guest is mapped in the bottom 4G and the VMM at the top of the 64-bit address space to avoid performance cliffs caused by overlapping the VMM's and guest's address spaces (Figure 1-2). This mapping also allows widening BT to take advantage of hardware features to restrict the guest to the low 4G of address space, so the VMM can be protected in more contexts than before. Protecting the VMM in turn allows more guest application code to be run directly on hardware and translations to be run at all privilege levels, opening optimization opportunities.

The VMM translates guest operating system code (privileged code) but executes most guest application code (unprivileged code) directly on hardware in compatibility mode (section 2.1.2 explains compatibility mode). Because guest operating system code is privileged and may take control of hardware resources if allowed to run directly on hardware, the VMM uses binary translation to emulate guest privileged instructions; binary translation "de-privileges" guest operating system code. Most guest registers are bound to their physical counterparts during the execution of translations so the binary translator can generate translations that are as close to the original guest instructions are possible. On the other hand, guest application code is already unprivileged and is safe to run directly on hardware. Figure 3-1 shows how guest operating system code is binary translated while guest application code is executed directly on hardware.

1.5 Evaluation

The result of balancing performance benefits of x86-64 mode and the cost of emulating x86-32 segmentation is that most 32-bit guest operating systems perform better under VMM64 in x86-64 mode than VMM32 in x86-32 mode. For VMware to discard VMM32, VMM64 simply needs to show "good enough" performance; even if VMM64 were slightly slower than VMM32, the human cost savings of simplifying the VMM programming model would make cutting VMM32 worth it.

OS install benchmark and micro-benchmarks were used to compare performance between running in x86-64 mode and x86-32 mode. The OS install benchmark times how long it takes to install a particular operating system followed by booting and shutting down the operating system at least once. It contains a mix of I/O and computation, is easy to run for different operating system, and is fairly consistent run to run. Micro-benchmarks include forkwait, getpid, signal, and switch. These micro-benchmarks make many system calls, which incur the most overhead under BT; system calls are the workload BT is worst at.

While a few old guests such as OS/2, Windows 95 and 98 are slightly slower when virtualized under x86-64 mode, the cost of maintaining VMM32 is greater than these slight setbacks. With widening BT, VMware can discard VMM32 once 64-bit CPUs are ubiquitous and focus on only maintaining VMM64.

By translating x86-32 code to x86-64 code, the VMM is simplified and performance of guest operating systems is improved. Translating x86-32 guest code to x86-64 code is cheap, with performance in x86-64 mode slightly better than x86-32 mode. Widening BT's main contribution is showing how to emulate x86-32 segmentation in x86-64 mode; segmentation hardware is used infrequently enough in modern operating systems that should it be dropped, software segmentation can replace it. By relaxing the amount of backwards-compatibility hardware must support, designers are free to extend the hardware and architecture.

Chapter 2

Introduction to x86

This chapter explains all you need to know about x86 in order to understand widening BT. The two main vendors of x86-based processors are Intel and AMD. The x86 architecture is a variable instruction length, primarily two-address CISC design with emphasis on backward compatibility. Each time the x86 architecture is extended, all previous operating modes are included for backward compatibility.

The x86 programming environment for the general-purpose instructions consists of the set of registers and address space. The operating mode determines the default operand size, address size, and register size of the programming environment. This chapter will describe each component of the programming environment in x86-32 mode and then describe how x86-64 mode extends it.

In order to run guests from all modes in 64-bit mode (x86-64), the VMM needs to provide a x86-32 execution environment that meets the guest's expectations.

2.1 Operating modes

The two main operating modes of x86 are *legacy mode* and *long mode*. Legacy mode supports legacy 16-bit and 32-bit operating systems and applications. Long mode is a combination of a 64-bit processor's 64-bit mode (x86-64) and a compatibility mode. Long mode's 64-bit mode has extended 64-bit registers and 64-bit addressing while long mode's compatibility mode provides backward compatibility with

16-bit and 32-bit code. x86-64 refers to long mode's 64-bit mode while x86-32 includes refers to legacy mode and long mode's compatibility mode.

Table 2.1 shows each operating mode's default operand size, address size, and register size. The OS column shows what codesize (64-bit, 32-bit or 16-bit) the operating system must support to handle trapping. For example, long mode needs an operating system that supports 64-bit (x86-64) code because traps switch into that code size.

Table 2.1: Operating modes

Operating mode		OS	Address	Operand	GPR
Long mode	64-bit mode	64-bit	64	32	64
	Compatibility mode		32	32	32
			16	16	32
Legacy mode	Protected mode	32/16-bit	32	32	32
			16	16	32
	Virtual 8086 mode		16	16	32
	Real mode	16-bit	16	16	32

2.1.1 Legacy mode

Real mode:

Real mode is the operating mode of 8086. Processors later than the 8086 support real mode. Real mode is the initial operating mode at power on. Real mode is still used for the BIOS, system management mode (e.g. power management). Real mode has a 20-bit (1MB) segmented memory address space. In real mode, each 16-bit segment register contains a value that represents the 16 most significant bits of the base of the 20-bit linear address. Adding the base to a 16-bit offset yields an actual linear address.

Virtual 8086 mode:

Virtual 8086 mode is a hybrid mode that allows real mode code to run while being supervised by a protected mode operating system. While real mode is implicitly privileged, virtual 8086 only runs in user mode. Virtual 8086 mode runs with the same segmentation and codesize as real mode. Virtual 8086 mode is only available under 32-bit protected mode. Virtual 8086 mode will not work under 16-bit protected mode or 64-bit codesize.

Protected mode:

Protected mode provides user and kernal modes (hence the name). Protected mode has a segmentation scheme that is very different from real mode's. In protected mode, segment registers store a segment selector which indexes into a segment descriptor table. The segment descriptor selected by the selector contains the base and limit of the segment. Protected mode also introduced 2-level paging.

2.1.2 Long mode

Long mode consists of two submodes, 64-bit mode and compatibility mode.

64-bit mode:

64-bit mode (x86-64) enables a 64-bit operating system to run applications written to access 64-bit linear address space. Default address size is 64-bit and default operand size is 32-bit. 64-bit mode has 64-bit flat linear address, twice as many GPRs as legacy mode, with each GPR twice as wide. The 64-bit mode instruction pointer is also 64-bit wide, and a new instruction-pointer-relative addressing mode (aka `%rip`-relative) is available. Furthermore, long mode introduces a new mechanism for delivering interrupts that is faster and more flexible.

Compatibility mode:

Compatibility mode supports 32-bit applications. Like legacy mode, compatibility mode executes 16-bit/32-bit code. Applications running under compatibility mode can only access the first 4G of linear address space.

Although long mode has compatibility mode to support 32-bit and 16-bit code, because long mode exceptions always cause the hardware to switch into 64-bit mode for exception handling, operating systems must contain 64-bit code to run in long mode. Another difference is that long mode always has 4-level paging enabled, while legacy mode has optional 2-level paging.

2.2 Programming environment

The programming environment for the general-purpose instructions consists of the set of registers and address space. The set of registers that define the programming environment are the general-purpose registers, the flags register, and the segment registers.

At a high level, 64-bit (x86-64) mode extends the programming environment of x86-32 mode by doubling the number and width of GPRs and doubling the number of address bits.

2.2.1 General Purpose Registers

In x86-32 mode, there are 8 general-purpose registers (GPRs). Each GPR is 32 bits wide. The GPRs available in x86-32 mode are `%eax`, `%ebx`, `%ecx`, `%edx`, `%esi`, `%edi`, `%esp`, `%ebp`. While these registers are "general", certain instructions use them for special purposes. Many instructions assign specific registers to hold operands. For example, stack operations assume that the stack is loaded in `%esp`, while string instructions use the contents of `%ecx`, `%esi` and `%edi` as operands.

In x86-64 mode, there are 16 general-purpose registers available. These include the 8 GPRs present in x86-32 mode and 8 new GPRs (`%r8` through `%r15`). `%r8` through `%r15` are available by using a REX instruction prefix. All 16 GPRs can be promoted to 64-bit. All of these registers can be accessed at the byte (8-bit), word (16-bit), double-word (32-bit) and quad-word (64-bit) level.

Operand size determines the number of valid bits in the destination GPR:

64-bit operand size generates 64-bit result in destination GPR.

32-bit operand size generates 32-bit result, zero-extended to a 64-bit result if the destination GPR is 64-bit bit.

8-bit and 16-bit operand sizes generate a 8-bit or 16-bit result. The upper bits (e.g. 56 bits or 48 bits in x86-64 mode) of the destination GPR are *not* modified by the operation.

2.2.2 Instruction pointer register

In x86-32 mode, the `%eip` register holds the 32-bit offset of the next instruction to be executed. In x86-64 mode, the `%rip` contains holds the 64-bit offset of the next instruction. x86-64 mode also allows instruction-pointer-relative addressing (aka `%rip`-relative addressing), where the effective address is determined by adding a displacement to the `%rip` of the next instruction.

2.2.3 Flags register

The flags register is a program status and control register. In x86-32 mode the flags register is called `%eflags`. In x86-64 the flags register (called `%rflags`) is extended to 64 bits; however, the upper 32 bits of the `%rflags` are reserved, and the lower 32 bits of `%rflags` are the same as `%eflags`.

There are a lot of flags stored in `%rflags`. However, for the purposes of this paper, you only need to know these flags [6]:

ALU status flags: The status flags indicate the results of arithmetic instructions such as CMP, ADD, SUB, MUL and DIV. ALU status flags include carry flag (CF), zero flag (ZF), sign flag (SF), overflow flag (OF), and more. ALU status flags are set by arithmetic instructions and used for conditional instructions (such as conditional moves and conditional branches).

Interrupt enabled flag (IF): Controls the response of the processor to interrupt requests. Set to respond to interrupts; cleared to inhibit interrupts.

Direction flag (DF): Controls string instructions (MOVS, CMPS, SCAS, LODS and STOS). Setting the direction flag causes the string instructions to auto-decrement (to process strings from high address to low addresses). Clearing the direction flag causes the string instructions to auto-increment (process strings from low addresses to high addresses).

I/O privilege level field (IOPL): Indicates the I/O privilege level of the currently running program or task. The current privilege level (CPL) of the currently running program or task must be less than or equal to the I/O privilege level to access the I/O address space. This field can only be modified by the POPF and IRET instructions when operating at CPL0. When IOPL=3, user code has access to the I/O address space.

2.3 Current Privilege Level (CPL)

The CPL is the privilege level of the currently executing program or task. There are four possible privilege levels from 0 to 3, often written as CPL0, CPL1, CPL2 and CPL3. CPL0 is the most privileged while CPL3 is the least privileged. The CPL dictates when privileged instructions may be executed. Privileged instructions, which control system functions such as the loading of system registers, may only be executed when CPL is 0. If one of these instructions is executed when CPL is not 0, a general-protection exception (#GP) is generated. The CPL can also be used to restrict the addressable domain. When the processor is in CPL 0, 1, or 2, it can access all pages; when it is in CPL3, it can only access user-level pages [7].

The CPL is stored in bits 0 and 1 of the %cs and %ss segment registers. Normally, the CPL is equal to the privilege level of the code segment from which instructions are being fetched. The processor changes the CPL when program control is transferred to a code segment with a different privilege level.

2.4 Memory

There are two parts to x86 memory management: segmentation and paging [7]. Both segmentation and paging provide memory protection. Segmentation provides a mechanism for isolating code, data, and stack so that multiple programs can run on the same processor without interfering with one another. Paging provides a mechanism for implementing a conventional demand-paged, virtual-memory system where sections of a program's execution environment are mapped into physical memory as needed. Paging can also be used to provide isolation between multiple tasks.

The use of paging is optional; there is a mode bit to disable and enable paging. When operating in protected mode, some form of segmentation must be used. There is no mode bit to disable segmentation, but many modern operating systems choose to just set up flat segmentation with a zero base and a maximal limit, thereby effectively disabling it. This is called flat mode.

While legacy and compatibility modes have segmentation, x86-64 mode segmentation is limited. In order to translate x86-32 mode to x86-64 mode, widening BT must emulate legacy protected segmentation.

2.4.1 Paging

When using x86's paging mechanism, the linear address space is divided into pages (4KB each). The operating system maintains a **page directory** and a set of **page tables** to keep track of the page mappings. When a program attempts to access an address location in the linear address space, the processor uses the page directory and page tables to translate the linear address into a physical address and then performs the requested operation (read or write) on the memory location.

If the page being accessed is not currently in physical memory, the processor interrupts execution of the program (by generating a **page-fault** exception). The operating system may then read the page into physical memory from the disk and continue executing the program. When paging is implemented properly in the operating-system, the swapping of pages between physical memory and the disk is transparent to the correct execution of a program. Even programs written for 16-bit processors can be paged (transparently) when they are run in virtual 8086 mode.

Page protection allows restricting access to pages based on two privilege levels: supervisor and user. Supervisor mode is for the operating system, other system software (e.g. device drivers), and protected system data (e.g. page tables). User mode is for application code and data. If the processor is operating at a CPL 0, 1 or 2, it is in supervisor mode; if it is operating at CPL3, it is in user mode. When the processor is in supervisor mode, it can access all pages; when it is in user mode, it can access only user-level pages [7].

2.4.2 Segmentation

In x86 IA-32 there is another layer of memory indirection called "segmentation" between virtual addresses and physical addresses.

$$\text{virtual address} \xrightarrow{\textit{segmentation}} \text{linear address} \xrightarrow{\textit{paging}} \text{physical address}$$

Figure 2-3 shows how segmentation and paging work in x86 when dereferencing a memory address.

x86 supports a segmented memory model, where memory appears to a program as a group of independent address spaces called **segments**. Segments are regions of the linear address space. Segmentation was introduced to the 286 in late 1980's, as memory got cheaper and the need grew for more address bits than the 286's 16-bit 64KB address limit.

Segments can be used to hold the code, data and stack for a program or to hold system data structures. Each program can be assigned its own set of segments. The hardware enforces boundaries between segments to ensure that one program does not interefere with another program by writing into the other program's segments. The segmentation mechanism also allows typing of segments so that the operations that may be performed on a particular type of segment can be restricted [7].

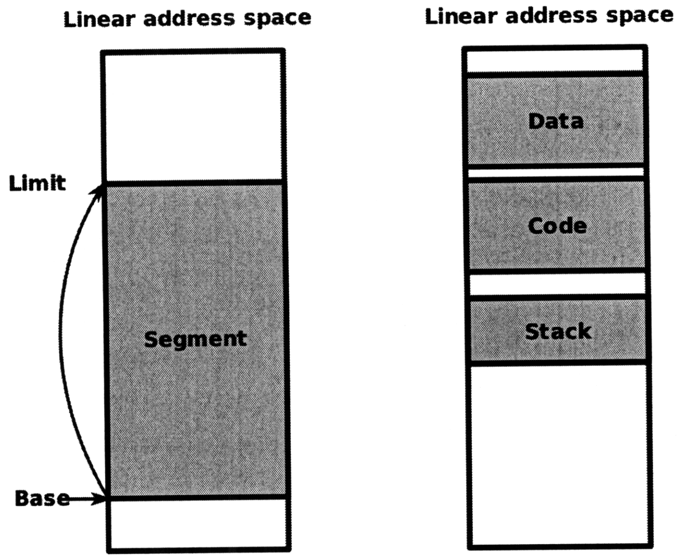

Figure 2-1: The segment base describes where the segment begins. The segment limit is the size of the segment. Each program can have up to 6 different segments at a time.

There are six 16-bit **segment registers**, allowing code to use six segments at a time. A segment register contains a 16-bit **segment selector** that points into a **descriptor table** (such as the global descriptor table, GDT) to a data structure called a **segment descriptor**. Each segment has a segment descriptor, which specifies where the segment begins in the linear address space (the **base** field), the size of the segment (the **limit** field), the access rights and a privilege level for the segment, and the segment type (code or data). Each segment can be assigned a privilege level from 0 to 3 (where 0 is most privileged and 3 is least privileged).

To address a byte in a segment, a program issues a virtual address, which consists of a **segment selector** identifying the segment to be accessed and an **offset** into the segment. If the offset of the memory access exceeds the limit of the segment to be accessed, the hardware GPs (general protection fault) because the memory access is outside the bounds of the segment.

If the memory access offset is within the limit, the hardware accesses the memory at the linear address, which is calculated as the sum of the base and offset

(Figure 2-2).

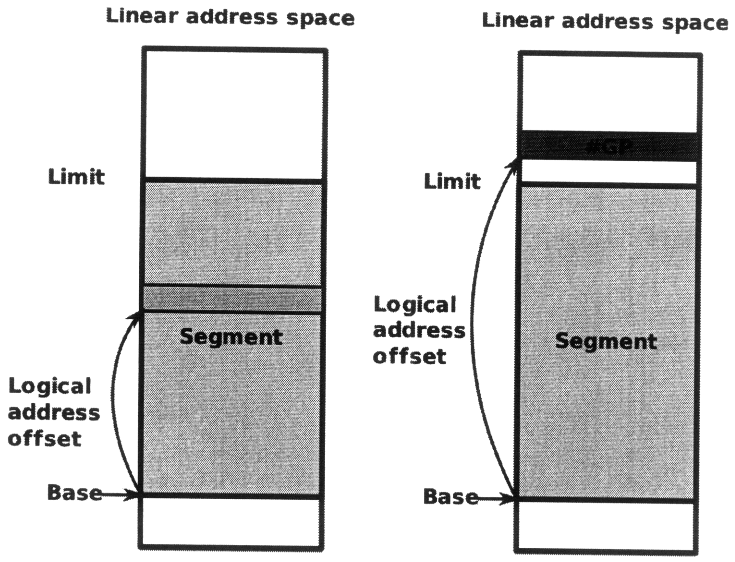

Figure 2-2: Accesses within the segment succeed (left). Accesses outside the segment limit GP (right).

Each of the six segments may have special uses. Code, data, and stacks are respectively contained in seperate segments `CS`, `DS` and `SS`, which are identified by 16-bit selectors contained in `%cs`, `%ds`, and `%ss` segment registers. There are also extra segments `ES`, `FS`, and `GS`, making a total number of 6 segments. A **segment instruction prefix** allows you to specify which segment to use for a memory-accessing instruction. See table 2.2 for default segment selection rules.

While x86-32 mode has full segmentation, x86-64 mode segmentation has no limits and only `FS` and `GS` have segment bases. AMD x86-64 mode has a special mode bit that turns on segment limits for `DS`, `ES`, `SS` and `GS`, but Intel x86-64 mode does not. In order to translate x86-32 mode to x86-64 mode, widening BT must emulate legacy protected segmentation.

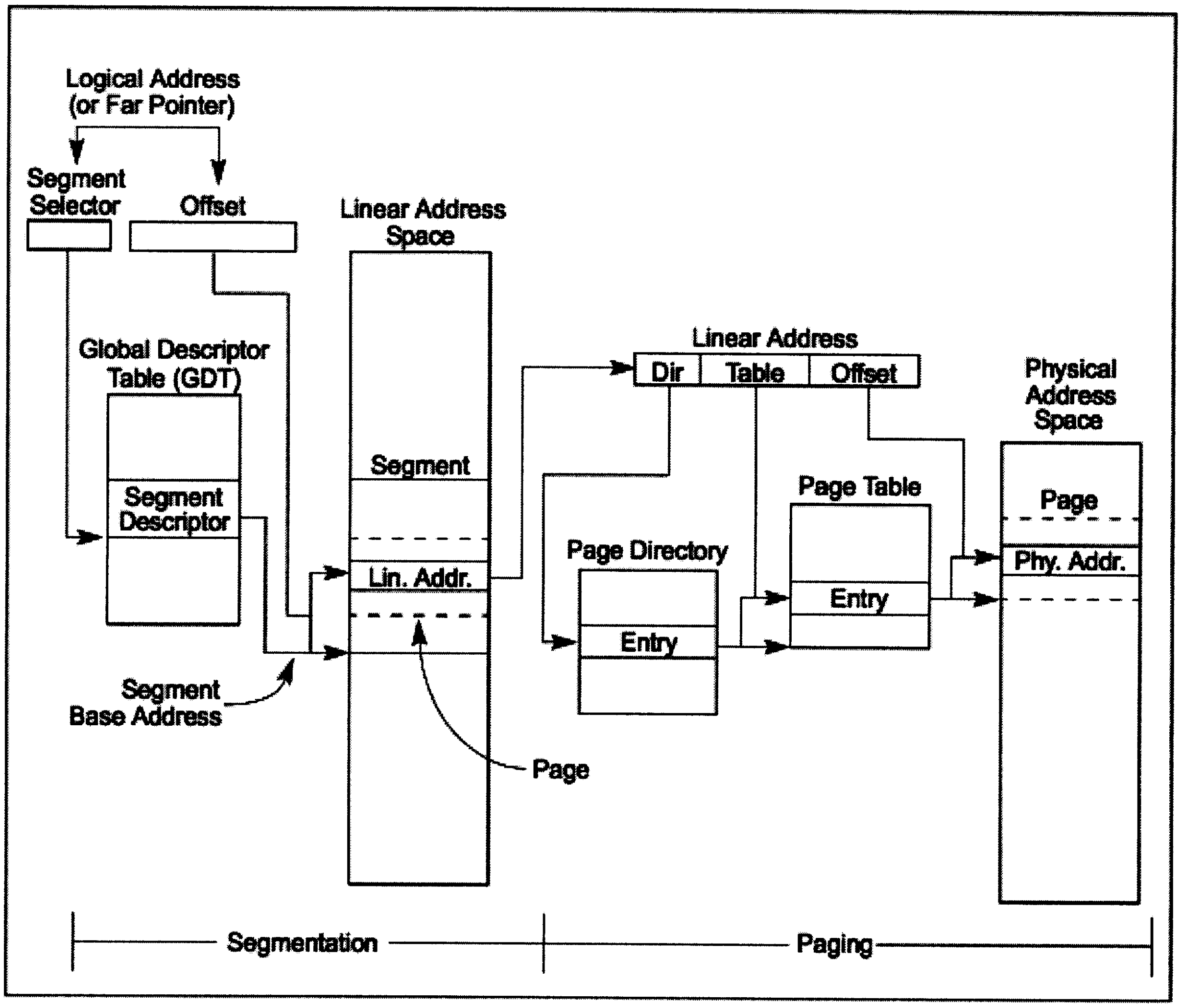

Figure 2-3: Segmentation and paging (Intel Section 3.1 Figure 3-1).

Table 2.2: Default segment selection rules.

Reference Type	Register Used	Segment Used	Default Selection Rule
Instructions	CS	Code Segment	All instruction fetches.
Stack	SS	Stack Segment	All stack pushes and pops. Any memory reference which uses the ESP or EBP register as a base register.
Local Data	DS	Data Segment	All data references, except when relative to stack or string destination.
Destination Strings	ES	Data Segment pointed to with the ES register	Destination of string instructions.

Chapter 3

Introduction to x86 Virtualization

This chapter introduces VMware's binary translation infrastructure before widening BT was implemented. A virtual machine monitor manages a guest operating system, which runs inside of a virtual machine. VMware uses one VMM per guest operating system. Each guest operating system lives in a separate address space. The VMM is responsible for running the guest operating system. VMware's VMM may have direct access to the hardware, but it is not responsible for scheduling. The "host" operating system is responsible for managing hardware resources of the physical machine and therefore responsible for scheduling the virtual machines (the guest operating systems and their corresponding VMMs). The "host" operating system can be a general purpose operating system like Linux or Windows, or VMware's VMkernel.

This chapter describes some common challenges of x86 virtualization and how VMware's VMM uses binary translation to overcome them. Challenges include de-privileging guest operating system code, protecting the VMM from guest memory accesses, and updating guest shadow data structures.

3.1 De-privileging

The way a VMM manages a guest operating system is analogous to the way an operating system manages an application. An operating system works by running at the most privileged level while restricting applications to running at lower privilege levels. Privilege levels are important for the containment and management of programs. To maintain isolation, unprivileged applications are not allowed to access privileged state; if an unprivileged application attempts to access privileged state, the access is trapped by hardware and the operating system is notified of the attempt. The difference between managing a user application and managing a guest is that the guest operating system expects to have privileged access to all hardware resources, while normal user applications do not; a VMM must go a step farther to trick the guest operating system into thinking it is privileged without allowing it to take over the machine.

The guest operating system code cannot be directly executed on the hardware because it contains privileged instructions that can take control of hardware resources. In order for a virtual machine monitor to manage guest operating system, the virtual machine monitor must be more privileged than the guest code running in the virtual machine. Therefore, the first step to virtualization is **de-privileging** guest code running in the virtual machine. The privileged instructions must be de-privileged before they are run.

One way to de-privilege guest operating systems is to run them at a user privilege level (any CPL > 0, such as CPL3 in x86) such that the privileged instructions trap. Once trapped, the privileged instructions can be emulated such that the guest does not realize it has been de-privileged. This method of de-privileging is called trap-and-emulate, and it assumes that all instructions reading or writing privileged state will trap when executed in an unprivileged context. However, there are a number of obstacles that prevent virtualizing x86 through trap-and-emulate.

First, certain privileged state is visible to user (unprivileged) code. The guest can observe that it has been de-priviledged when it reads its code segment selector (%cs) since the current priviledge level (CPL) is stored in the low two bits of %cs. The guest can also examine flags with PUSHF, which doesn't trap at user-level (CPL > 0).

Second, some privileged instructions do not trap when run at user level. For example, in priviledged code POPF ("pop flags") may change both ALU flags (e.g., ZF, the zero flag, which is non-privileged) and system flags (e.g., IF, the interrupt flag, a privileged flag that controls interrupt delivery). For a de-privileged guest, the VMM needs kernel mode POPF to trap so that the VMM can emulate it against the virtual IF. Unfortunately, a de-priviledged POPF, like any user-mode POPF, simply suppresses attempts to modify IF; no trap happens.

These obstacles to x86 virtualization can be overcome if the guest instructions are interpreted instead of run directly on a physical CPU, seperating the virtual state of the CPU from the physical state of the CPU. While interpreting guest instructions can prevent leakage of priviledged state and can correctly implement non-trapping priviledged instructions, interpretation is by no means efficient. The interpreter may burn hundreds of physical instructions per guest instruction in its fetch-decode-execute cycle. An alternate way of separating the virtual state of the CPU from the physical state is needed.

3.2 Binary translation

Binary translation effectively overcomes x86 virtualization obstacles while remaining performant. Binary translation is the process for emulating a source instruction architecture on a target instruction architecture by decoding the source machine code, translating into the target machine code and executing on the target machine. Binary translation is often used when the source and target architectures are very different; Java JIT compilers (e.g. LaTTe [10]), Rosetta [4] and QEMU [5] are all examples of this. Most importantly, binary translation can be used to implement **system-level virtualization**, which provides a complete simulation of a host machine's underlying architecture. When used for system virtualization, the source and target architectures for binary translation are very similar. For system virtualization, the target architecture is most often a **de-privileged subset** of the source architecture.

In general, Binary translation is performant because the overhead of translation is amortized over the many times the translated target code is run; translate once, run many times. For x86 virtualization, binary translation is particularly performant, especially when the vast majority of x86 instructions can be translated identically to run at native speed on host hardware. Even for widening BT, most x86 instructinos can be translated nearly identically. Software virtual machine monitors such as VMware Workstation and Virtual PC use binary translation to virtualize x86, allowing x86 virtual machines to be used for server consolidation, fault containment, security and resource management.

3.3 VMware's virtual machine monitor

There are many flavours of binary translation. VMware's VMM uses a binary translator with the following properties [1]:

Binary: VMM translates input guest x86 binary code into de-privileged target x86 code. The de-privileged target x86 ISA is a subset of the x86 ISA.

Dynamic: Translation happens at runtime, interleaved with execution of generated code. At runtime, a dynamic translator has access to complete linked program, program state (CPU and memory), and can adapt to changes in program behavior.

On demand: Guest code is translated only when it is about to execute. This laziness side-steps the problem of telling code and data apart or decoding variable-length instructions without knowing instruction offset.

System level: No assumptions are made about guest code behavior. The VMM faithfully emulates executing every single instruction of guest x86 code. An application-level translator may make assumptions about guest code based on a higher-level application binary interface (ABI).

Adaptive: Translated code is adjusted in response to guest behavior changes to improve overall efficiency. For example, VMware's binary translator can detect which instructions would have page faulted (because of tracing) and generate translations to avoid faulting.

Before widening BT, VMware's virtual machine monitor performed **same-codesize** binary translation. The VMM translates x86-32 code to x86-32 code and x86-64 code to x86-64 code, but not x86-32 code to x86-64 code. In order to execute x86-64 and x86-32 translations, VMware's virtual machine monitor can run in both x86-32 and x86-64 modes and switch between them. Under same-codesize BT, the translations are always run in the same codesize as the source instructions. The VMM's x86-32 mode is named VMM32 and the VMM's x86-64 mode is named VMM64 (Figure 1-1).

The VMM does not binary translate all guest code. The VMM uses binary translation to de-privilege guest kernel (CPL0) code, but for efficiency and simplicity, it executes guest user (CPL3) code directly on hardware (with some exceptions). This execution mode is called **direct execution**.

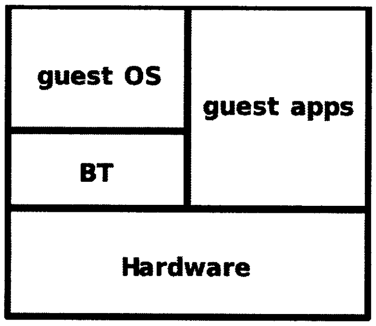

Figure 3-1: Guest OS privileged code is binary translated into unprivileged code that is safe to run on the hardware. Guest application (user) code is already unprivileged and safe to run on the hardware.

Before widening BT, VMware's binary translator translated 32-bit guest operating system code into x86-32 code running on x86-32 mode hardware under a x86-32 virtual machine monitor (VMM32), not taking advantage of x86-64 CPUs at all. Widening BT translates 32-bit guest operating systems into x86-64 code, where they can run on x86-64 mode hardware under a x86-64 VMM (VMM64, see Figure 3-2).

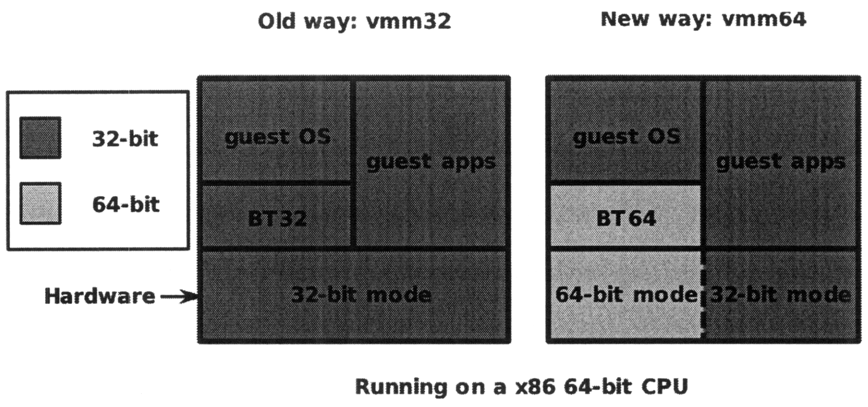

Figure 3-2: Widening BT changes the target codesize of the VMM's binary translator to be always 64-bit.

3.3.1 Binary translation process

The main steps of VMware's binary translation process are decoding, translation, and chaining.

The translator begins by reading the guest's memory at the address indicated by the guest PC (program counter/instruction pointer), classifying the bytes as prefixes, opcodes or operands as the bytes are read, effectively decoding guest bytes into instructions.

The translator then accumulates decoded guest instructions into a **translation unit** (TU), stopping at a terminating instruction (usually control flow) or a fixed-size cap. The translation unit forms a single-entry multiple-exit basic block. Translating from x86 to x86 subset, most unprivileged code in a TU can be translated identically (also known as IDENT). Most virtual registers are bound to their physical counterparts during execution of translations in order to facilitate IDENT translation. Control flow instructions cannot be translated identically because translation does not preserve layout. Instead, the translator turns each control flow instruction into two translator-invoking continuations, one for each of the successors (fall-through and taken-branch, more in section 4.4).

Each translator invocation consumes one TU and produces one **compiled code fragment** (CCF), also a single-entry multiple-exit basic block. Because VMware's binary translator is dynamic, translation of TUs is interleaved with execution of corresponding CCFs. To speed up inter-CCF transfers, the translator employs a **chaining** optimization, allowing one CCF to jump directly to another without calling out of the **translation cache** (TC). These chaining jumps replace the translator-invoking jumps, which therefore are "execute once." Moreover, it is often possible to elide chaining jumps and fall through from one CCF into the next.

Interleaving of translation and execution continues for as long as the guest runs, with a decreasing proportion of translation as the TC gradually captures the guest's working set. A detailed explanation and examples of VMware's binary translation process can be found at [1].

3.3.2 Shadowing

Not only does binary translation need to de-privilege guest operating system code, BT needs to provide an execution environment that meets the guest's expectations, despite the fact that the guest's privileged state differs from that of the underlying hardware.

To provide an execution environment that meets the guest's expectations, the VMM derives **shadow structures** from guest-level **primary structures** [1]. For on-CPU privileged state, such as the page table pointer register (%cr3) or processor status register, the VMM simply maintains an image of the guest register and refers to that image in instruction emulation as guest operations trap. For off-CPU privileged state such as page tables, which may reside in memory, the VMM traces (write-protects or read-protects) the guest's primary structure. With traces, modifications to the primary structures will trap, allowing the VMM to update shadow structures.

To make use of primary and shadow structures, translations must frequently access both guest primary structures and the primary structures maintained by the VMM. One of the challenges of virtualization is to switch between accessing guest data structures and VMM data structures quickly. In order to do so, the VMM and guest must share an address space.

3.3.3 Protecting the VMM with segmentation

Protecting the VMM from guest accesses is one of the challenges of virtualization. This section explains how, before widening BT, VMware used segmentation to protect the VMM.

In order to quickly switch between accessing guest data structures and VMM data structures, the VMM and the guest share the same address space. Before widening BT, the guest and VMM address spaces always overlapped. VMM32 and 32-bit guest operating system overlapped while sharing 32-bit address space. VMM64 and 64-bit operating system overlapped while sharing 64-bit address space. The guest expects access to all addresses, so the VMM must be protected from guest memory accesses. An efficient way to protect the VMM is with hardware segmentation. Segment protection of the VMM allows the VMM to maximize the number of instructions translated identically; else the VMM must use software checks or page protection, both of which are expensive.

By mapping the guest low in memory and the VMM high in memory, segmentation can be used to segregate guest portions (low) and VMM portions (high) of memory. The VMM then **truncates** guest segments so they don't include the region of address space the VMM is in. Figure 3-3 shows how VMM32 truncates guest segments because VMM32 and 32-bit guest share the same 32-bit address space.

The guest's actual, un-truncated segment values are stored in the VCPU. VMM32 takes the guest segment values, truncates them, and loads these values onto hardware when translations are executed. However, if *all* segments in hardware are truncated to not include monitor address space, then there wouldn't be a way to access the VMM data structures.

In order to have access VMM data structures, GS was selected to be special. GS is the only segment that spans all of the linear address space and includes the VMM, providing an escape into VMM data structures. Other segments (DS, SS, ES, and FS)

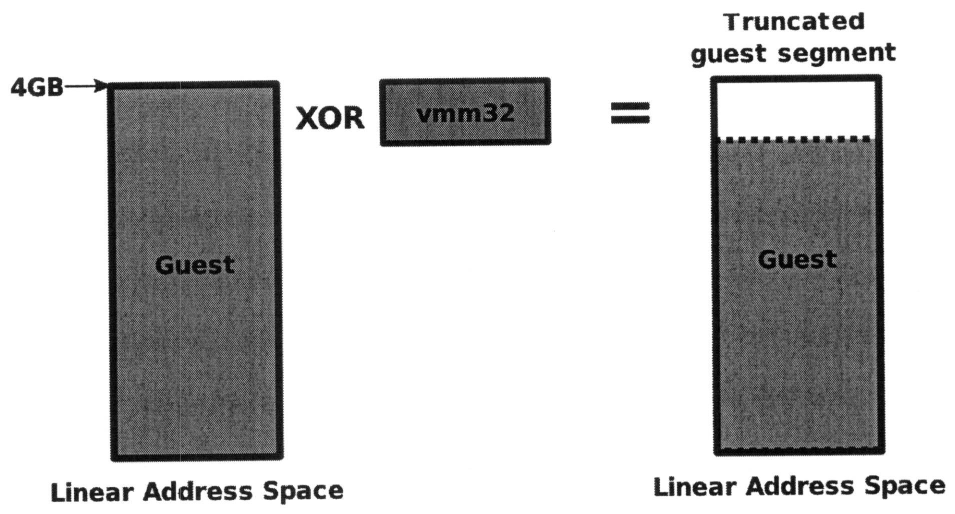

Figure 3-3: Guests segments are truncated to not include the region of memory where the VMM resides. Hardware segmentation mechanism protects the VMM from guest accesses.

are still loaded with truncated guest segments.

To emit instructions that access VMM data structures, the binary translator emits the instructions with GS segment prefixes. To prevent translations of guest instructions from accessing the VMM, the binary translator strips the GS prefix from translations of guest instructions and emits code to emulate GS memory access. The emitted code uses FS as a temporary segment, temporarily loading FS with truncated guest GS value, re-emits the guest instruction with the FS prefix (emulating the GS memory access), and lastly restores FS to contain truncated guest FS value.

When not using widening BT, segmentation is vital for protecting the VMM when using binary translation for virtualization. However, 64-bit mode on Intel has segment limits disabled. Therefore BT cannot be efficiently used to virtualize 64-bit guests on Intel machines. Hardware virtualization must be used to virtualize 64-bit guests on Intel machines. With widening BT, which allow the VMM to virtualize 32-bit guests in x86-64 mode, the VMM can be protected without segmentation. The VMM takes advantage of the address-size override hardware feature to protect the VMM.

Chapter 4

Widening binary translations

This chapter explains the widening binary translations of the most common instructions found in guest operating systems. All of these instructions are unprivileged and many of them can even be translated nearly identically. This chapter examines simple unprivileged instructions, stack operations and control flow. In doing so, it describes how BT preserves the appearance of instruction atomicity and optimizes indirect control flow.

4.1 Nearly IDENT instructions

Most source instructions are unprivileged, even in kernel code. When performing same-codesize binary translation, these unprivileged instructions can be translated identically (aka IDENT). However, when binary translating from x86-32 to x86-64, additional modifications need to be made to these otherwise IDENT instructions. Examples of nearly IDENT instructions include non-memory accessing instructions or simple mem-accessing instructions like MOV.

In order to translate nearly IDENT instructions, widening BT needs to preserve the operand size and address size. In x86-32 mode, the D flag in the code segment descriptor determines the default operand and address size (see Table 4.1).

Setting the D flag selects the 32-bit operand size and address size attributes. Clearing the D flag selects the 16-bit operand size and address size attributes.

When the processor is executing in real mode, the default operand size and address size attributes are always 16-bit (the D flag is clear). In x86-64 (64-bit mode), the default operand size is 32-bit and the default address size is 64-bit. The REX.W prefix in 64-bit mode forces a 64-bit operand size; when the REX.W prefix is present, the operand size prefix is ignored (see Table 4.2).

By comparing the effective operand size and address size encodings of x86-32 mode and x86-64 mode in table 4.1 and table 4.2, widening BT can determine when to add or strip operand size and address size prefixes.

First of all, widening BT must preserve operand size when translating to x86-64 mode. Widening BT preserves the operand size by adding or removing operand size prefix (0x66) as necessary. Simply preserving the operand size allows widening BT to translate most unprivileged and non-memory-accessing instructions from x86-32 to x86-64.

x86-64 translations of instructions with 32-bit operand size will not use an operand size prefix, while translations of instructions with 16-bit operand size will need an operand size prefix. If the D flag in the code segment descriptor is set, widening BT needs to toggle the operand size prefix on the translation. If the D flag is clear, widening BT doesn't need to alter the operand size prefix.

Next, widening BT needs to preserve address size for mem-accessing instructions. The address size prefix (0x67) allows widening BT to represent 32-bit address size in 64-bit mode, but there is no way to represent 16-bit address size in 64-bit mode. In order to calculate a 16-bit address, widening BT uses a LEA instruction with 16-bit operand size to compute the effective 16-bit address. In general, all 64-bit nearly-IDENT translations of memory-accessing instructions will have an address size prefix to force address size to 32-bit, while 16-bit address size instructions will depend on a LEA instead.

Table 4.1: Effective Operand- and Address Size Attributes in x86-32 mode (Intel Manual Volume 1 Section 3.6 Table 3-3)

D Flag in Code Segment Descriptor	0	0	0	0	1	1	1	1
Operand Size Prefix 0x66			op	op			op	op
Address size Prefix 0x67		ad		ad		ad		ad
Effective Operand Size	16	16	32	32	32	32	16	16
Effective Address Size	16	32	16	32	32	16	32	16

Table 4.2: Effective Operand- and Address Size Attributes in 64-bit mode (Intel Manual Volume 1 Section 3.6 Table 3-3)

L Flag in Code Segment Descriptor	1	1	1	1	1	1	1	1
REX.W Prefix	0	0	0	0	1	1	1	1
Operand Size Prefix 0x66			op	op			op	op
Address size Prefix 0x67		ad		ad		ad		ad
Effective Operand Size	32	32	16	16	64	64	64	64
Effective Address Size	64	32	64	32	64	32	64	32

4.2 Stack operations

After the nearly IDENT instructions, stack operations PUSH and POP are the most common instructions. Widening translation of stack operations is tricky because the stack size is forced to 64-bit in 64-bit mode (x86-64), while 32-bit guests want to use 32-bit or 16-bit stack size.

A guest 32-bit stack size PUSH or POP expects to use and modify `%esp` (which corresponds to the low 32 bits of `%rsp`), but since there can't be a 32-bit stack size in 64-bit mode, widening BT will have the wrong stack-wrapping behavior. The guest stack pointer is live in `%rsp` when executing translations. A POP could increment guest stack pointer to be above 4GB and a PUSH could decrement guest stack pointer such that it wraps around to the top of the address space. Widening BT has to make sure the high 32 bits of the stack pointer are clear when emulating a 32-bit stack size in 64-bit mode.

Furthermore, a guest 16-bit stack size PUSH or POP expects to modify only the low 16 bits of the stack pointer while leaving the high bits the same. There is no way to translate a 16-bit stack size PUSH or POP to a corresponding PUSH or POP in 64-bit mode such that only the low 16 bits of the stack pointer are modified.

Another perceived difficulty is that %rsp would not be 64-bit aligned under widening BT. However, this "stack misalignment" will not cause any performance degradation. Intel and AMD encourage data to be "naturally aligned"; 1-byte data is always aligned, 2-byte data is naturally aligned when 2-byte aligned, 4-byte data is naturally aligned when 4-byte aligned, etc. Therefore %rsp does not need to be 64-bit aligned for good performance; widening BT mostly works with 32-bit and 16-bit values, which will be naturally aligned to the 32-bit or 16-bit guest stack pointer loaded in %rsp.

Due to the difficulty of representing a 32-bit or 16-bit stack size in 64-bit mode, widening BT does not translate guest PUSHes and POPs to corresponding 64-bit mode PUSHes and POPs. Widening BT translates PUSH or POP respectively into a MOV to or from the stack and a LEA to decrement or increment the stack pointer.

A 32-bit operand size LEA adjusting the stack pointer will emulate a 32-bit stack size while a 16-bit operand size LEA will emulate a 16-bit stack size, keeping the upper 48 bits of the stack pointer %rsp register intact. Widening BT uses a LEA instruction instead of an arithmetic instruction (e.g. ADD or SUB, INC or DEC) because arithmetic instructions alter the flags register %rflags while LEA does not. When executing translations, virtual registers (including %rflags) are bound to their physical counterparts as much as possible to facilitate IDENT (or nearly IDENT) translation, so widening BT goes to great lengths to avoid altering registers with live guest values.

Let D flag in code segment descriptor be set (default operand size and address size are 32-bit). Let <ad> represent address size override prefix and <op> represent operand size override prefix.

The translations of 32-bit PUSH and POP are:

```
Source: push %eax                Source: pop %eax
Translation:                     Translation:
```

```
lea -4(%esp) -> %esp            <ad> mov (%esp) -> %eax
mov %eax -> (%esp)              lea 4(%esp) -> %esp
```

If widening BT knows that the high 32 bits of %rsp are clear, it doesn't need to emit an address size prefix on the MOV for POP %eax. The LEA for PUSH %eax always clears the high 32 bits of %rsp, so widening BT never needs an address size prefix on the MOV following the LEA.

The translations of 16-bit PUSH and POP are:

```
Source: <op> push %ax           Source: <op> pop %ax
Translation:                    Translation:
<op> lea -4(%sp) -> %r8w        <op> movzx %sp -> %r8d
movzx %r8w -> %r8d              <op> mov (%r8d) -> %ax
<op> mov %ax -> (%r8d)          <op> lea 2(%r8w) -> %sp
<op> mov %r8w -> %sp
```

16-bit PUSH and POP have more complicated translations. This is because 16-bit stack operations ignore but do not clear the high 48 bits of %rsp. The MOVZX (move with zero-extension) instructions are necessary, because 16-bit operand size operations do not clear the top bits of the registers and widening BT doesn't have a way to directly represent 16-bit address size in x86-64 mode. Widening BT stores the 16-bit stack address in a temporary register %r8, dereference and update it, then commit the updated address to %rsp.

For memory to memory operations like PUSH from memory and POP from memory, widening BT uses an intermediate temporary register %r12 to store the value. The memory to memory operation becomes memory to %r12, %r12 to memory.

4.3 Sync regions

Widening BT's translations for PUSH and POP contain multiple instructions—MOV and LEA. When widening BT translates a single source instruction into multiple target instructions, widening BT must preserve the appearance of atomicity. In order to preserve the appearance of atomicity, should widening BT take a fault, interrupt, or trap in the middle of executing the translation, widening BT needs to be able to roll forward or roll back the state of execution to the beginning or end of the instruction. Therefore each multi-instruction translation of a single guest instruction is associated with a **sync region** and **sync callback**, a function that is called should the execution of the translation be interrupted.

The sync region stores the addresses of the beginning and end of the compiled code fragment (CCF) in the translation cache, as well as what type of instruction it is a translation of (memory-accessing, stack operation, control flow, etc). Each sync region type is associated with a sync callback function, which figures out where in the translation the fault, interrupt, or trap occurred and how to roll execution of the translation forward or back. Widening BT rolls backwards on faults and forwards on traps; in x86, traps happen after the instruction. On interrupts, widening BT can choose whether to roll forward or back depending on whichever is convenient.

Translations are run much more frequently than their sync callbacks are invoked, so widening BT is motivated to optimize translations much more than sync callbacks. However, each bit of complexity an optimization introduces into a translation corresponds to far more complexity introduced in the corresponding sync callback. VMware wants the best performance for the lowest complexity, so while there are optimizations with lower complexity cost to be made, VMware defers complicating the VMM code. Let's take a look at how widening BT's PUSH translation influences the complexity of the PUSH sync callback.

There are two ways to translate PUSH. Widening BT could decrement the stack pointer, then move the operand to the stack, where the decremented stack pointer is pointing. This takes two instructions. Alternatively, widening BT could compute the effective destination address of operand, move the operand to that address, and then update the stack pointer. Here is assembly code for the the two methods:

```
Example: push %eax

Method 1:                   Method 2:
```

```
lea -4(%esp) -> %esp      lea -4(%esp) -> %r10
mov %eax -> (%esp)        mov %eax -> (%r10)
                          mov %r10 -> %rsp
```

Method 1 is shorter than method 2. If widening BT decrements the stack pointer last in the translation block (method 2), widening BT doesn't need to restore the stack pointer if the translation faults on the memory access; the translation hasn't clobbered the stack pointer yet when the fault is taken. However, committing the decrement of stack pointer last takes an extra instruction. PUSH is such a popular instruction that an extra instruction in its translation takes up considerable space in the translation cache and an extra few cycles per PUSH during runtime. In this case, making the sync callback function for PUSH smart (more complicated) to roll back `%rsp` is worthwhile because trimming PUSH's translation to be one instruction shorter results in great gains due to how much guest operating systems use the PUSH instruction.

When the PUSH sync callback is called to handle a faulting PUSH, the sync callback needs to know the address size and operand size in order to restore `%rsp`. The address size and operand size can be decoded from the LEA instruction at the beginning of any PUSH translation. If the LEA instruction has an operand size prefix, the sync callback knows that the stack size was 16-bit. The displacement of the LEA instruction reveals the size of the operand pushed onto the stack.

Instead of having the sync callback figure out the operand size and address size, widening BT could create different sync region types for each PUSH operand size and address size combination, but that would take up room in the sync type table, wasting memory.

4.4 Control Flow

Because translation does not preserve code layout, control flow instructions cannot be translated identically. Instead, control flow instructions terminate translation units and are translated into two translator-invoking continuations, one for the taken-branch and one for the fall-through:

```
Example source:
cmp %bl, $0xf00
jnz 0x5

Translation:
cmp %bl, $0xf00
jnz [takenAddr]  ; JCC
jmp [fallthrAddr]
```

Through the "chaining" optimization, the two translator-invoking continuations can be replaced with jumps to other compiled code fragments. The CCFs can also be laid out such that one CCF falls through to the next.

A direct control flow instruction is easy to chain. If one of the two translator-invoking continuations is taken, the translator maps the guest address to the TC address of the target CCF and replaces the continuation with the TC address.

Indirect control flow instructions cannot be chained in this way. The target of an indirect control flow instruction is not fixed; it must be computed dynamically, e.g. with a hash table lookup. To speed up indirect control flow target lookups, widening BT uses a guest address to TC address cache. A CALL translation also preemptively places the return address in a return target cache to speed up a corresponding RET's target lookup [3]. The cache is flushed whenever the code segment (CS) changes. 32-bit guests change the code segment quite often; in 64-bit mode, the code segment is defined to be flat with base zero and limit maximal and changes a lot less. To speed up indirect calls even further, widening BT has inlined calls; widening BT predicts the target of the indirect control flow instruction and bind to the prediction.

Chapter 5

Emulating segmentation

While x86-32 mode has full segmentation, x86-64 mode segmentation has no limits and only FS and GS have segment bases. AMD x86-64 mode has a special mode bit that turns on segment limits for DS, ES, SS and GS, but Intel x86-64 mode does not. Therefore emulating segmentation is the primary challenge in widening binary translation. There are two types of memory accesses: instruction fetches and data accesses. Instruction fetches come from code segments while data accesses come from data segments. Widening BT needs to emulate segmentation for these two types of memory accesses.

5.1 Code base and limit

All x86 instruction fetches come from code segment CS. Widening BT emulates segmentation by checking code segment base and limit when entering the translation cache to execute translations.

Widening BT tags each CCF (compiled code fragment, which is a basic block of translated code) in the translation cache with the code base and code limit; the code base and limit for each CCF is stored in a corresponding "sourcekey" (details in section 5.1.1). When attempting to run a translation, widening BT makes sure that the code base and limit of the translation matches the guest operating system's current base and limit. When "chaining" one CCF to another, widening BT verifies that the code base and limit of both CCFs match.

Because chained CCFs must have the same code base and limit, widening BT only needs to check the code base and limit of the first CCF it executes when entering the translation cache; all other CCFs that chain from the initial CCF have the same code base and limit as the initial CCF. By doing so, widening BT ensures that the appropriate code base and limit are respected without having to frequently pay the cost of dynamic checks.

The big picture:

1. At translation time: perform segment limit and access rights check and store the code segment base and limit (in a "sourcekey").
2. When entering translation cache (execution time): check whether the current code segment base and limit match the translation time base and limit.
3. If base and limit are the same, execute translation. Widening BT has already performed the segment checks at translation time.
4. If base or limit differ from translation time, retranslate.

CCFs translated under one set of code base and limit are not be run under a different code base and limit, even if the target code satisfies the conditions imposed by the new code base and limit. This design decision makes enforcing code base and limit when chaining blocks of translated code simpler. Otherwise chaining becomes very complicated; even if widening BT knew one CCF (block of translated target code) can be reused under a new code base and limit, widening BT wouldn't be sure that everything it chains to can be reused.

5.1.1 Sourcekeys

Widening BT can take advantage of **sourcekeys** for checking code base and limit each time before translations are run.

A sourcekey describes a block of translated code, including its position in memory, its privilege level, etc. Sourcekeys are identifiers for blocks of translated code.

Sourcekeys serve to uniquely identify the translation unit and also contains the "context" under which the translation unit (TU) was generated. Code translations generated under one context should not be executed under a different context; code translated under privileged mode should not be executed in a user mode context; code translated under one set of code base and limit should not be run under a context with a different code base and limit. Before translated code is run, the current execution context is checked against the sourcekey, which stores the translation-time context.

In order to perform code base and limit checks for widening binary translation, I extended the VMM64 source keys to hold code base and limit information. Having the code base and limit stored in the sourcekey allows the monitor to check all components of the context (segment bases and limits, code priviledge, etc.) before translated code is run.

5.2 Data base and limit

While a program can only have one code segment at a time, a program can have multiple data segments (SS, DS, ES, FS, GS) at the same time. Although code base and limit were taken care of with sourcekeys, data base and limit still need to be reckoned with.

5.2.1 Software segmentation

Widening BT emits software segment checks because it is the simplest way to emulate segmentation. Here are the steps taken in assembly to perform a software segment check:

1. Cache `%rax` in `%r11`. `%r11` will be used to restore `%rax` at the end of software segment check. Widening BT needs `%rax` to save the contents of `%rflags` (the status flags register) to memory.

2. Store status flags in `%rax` with LAHF ("load status flags into `%ah`").

3. The VMM keeps a reformatted copy of the guest descriptor tables (DT); because x86 segment descriptor formatting is inconvenient, reformatting the guest segment descriptors makes them easier to read. Load index which points to reformatted copy of guest segment descriptor `DS` into `%r8`.

4. Indexing off of `%r8`, load guest `DS` segment descriptor flags into `%r10`.

5. Check whether expand-down flag is set in `%r10`.

6. If segment is expand-down, bail to interpreter. Expand-down segments are uncommon, so bailing to interpreter is more efficient than adding more code to software segment check to deal with expand-down segments.

7. If segment is expand-up, load offset of logical address of memory access into `%r10`.

8. Load the maximum target address into `%r15`.
   `%r15` = offset + accessSize - 1.

9. If maximum target address (`%r15`) wraps around 4G address space, bail to interpreter. Wrapping around 4G address space implies that `%r15d` overflowed 32 bits, so the following limit check, which performs a comparison against the segment limit, would be invalid.

10. Check whether access exceeds segment limit; if so, bail from translation and deliver a general protection fault (GP).

    This check performs comparison between target maximum target address and segment limit. The comparison operation bashes the ALU flags, which is why the status flags had to be saved in `%rax` earlier.

11. If the access passes the limit check, load guest segment base into `%r8`.

12. Before finally executing guest memory access, restore `%rflags` and `%rax`.

13. Perform guest memory access. The target 32-bit linear address is the sum of the guest segment base and logical offset.
linear address = %r10 + %r8 = offset + base.

Here is an example of `MOV %eax <- 0x58(esi)` with emitted software segment check, with corresponding lines of assembly numbered to reflect what step they are.

```
<REXWB> MOV    %r11 <- %rax                     ; (1)
 LAHF  %ah                                      ; (2)
 SETO  %al
<REXRB> XOR    %r8d <- %r8d                     ; (3)
<OP><REXR> MOV    %r8w <- 0xfc0190d6
<REXB> SHR    %r8d <- $0x2
<REXRX> MOVZX %r10d <- 0xfc0789b0(r8)           ; (4)
<REXB> AND    %r10d <- $0x5                     ; (5)
<REXB> CMP    %r10d <- $0x5
 JZ     0x2 ;0xffffffffd8aa6d0                  ; (6)
 FAULTDISP 9 FAULTDISPATCH_WIDENSEGCHK_DS
<REXR> LEA    %r10d <- 0x58(rsi)                ; (7)
<REXRB> MOV    %r15d <- %r10d                   ; (8)
<REXB> ADD    %r15d <- $0x3
 JC     0xf1 ;0xffffffffd8aa6ce                 ; (9)
<REXRX> CMP    0xfc0589b4(8*r8) <- %r15d        ; (10)
 JC     0xe7 ;0xffffffffd8aa6ce
<REXRX> MOV    %r8d <- 0xfc0589b0(8*r8)         ; (11)
 CMP    %al <- $0x81                            ; (12)
 SAHF   %ah
<REXWR> MOV    %rax <- %r11
<A><REXXB> MOV    %eax <- (r10d + r8d)          ; (13)
```

Emitting code to do data segmentation limit checks takes considerable space in the TC and CPU cycles at runtime. For each memory-accessing instruction, about a dozen instructions are emitted for software limit checks. Many of the emitted instructions are themselves memory accessing. Although software limit checks are bulky, many software limit checks can be reduced.

5.2.2 Flat segments

Most software segment checks can be eliminated. Many modern 32-bit operating systems choose to just set up flat segmentation with zero base and a maximal limit, effectively disabling segmentation. This is called flat mode (see Figure 5-1 for a flat segment).

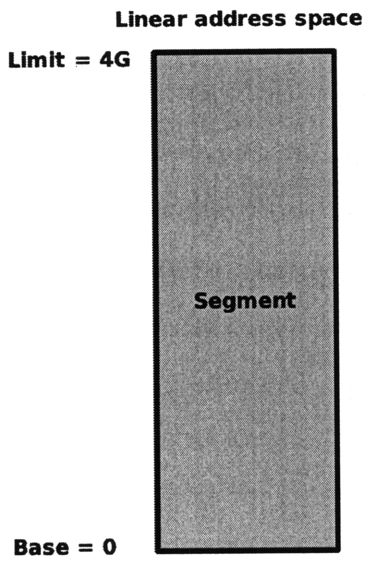

Figure 5-1: A flat 32-bit segment.

In flat mode, widening BT doesn't need to emit software segment checks. Because a flat segment has base 0, the logical address offset into a flag segment *is* the linear address. Because limit is maximal (4G), widening BT just needs to make sure guest memory addresses are 32-bit. For example, a 32-bit guest can generate large addresses with an a addressing mode involving two registers: `MOV %eax -> (%ebx + 4 * %ecx)`. By using the address-size prefix (0x67) on memory-accessing instructions, widening BT can make sure all guest memory addresses are 32-bit. Widening BT also needs to watch out for memory accesses that straddle the end of a flat segment. To check for accesses that straddle the 4G segment boundary, widening BT protects the $4G + 1$ page and deliver page faults (PFs) on the $4G + 1$ page as general protection faults (GPs) instead (see Figure 5-2).

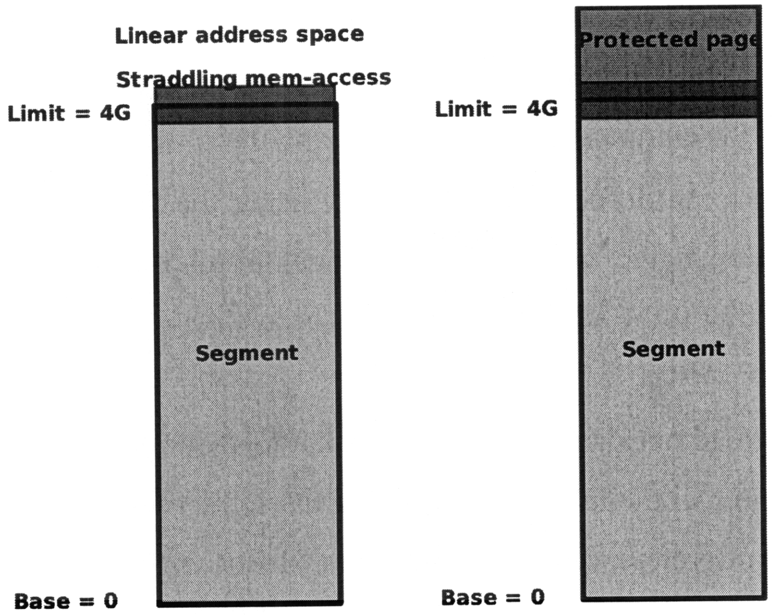

Figure 5-2: Although a 32-bit flat segment spans all of 32-bit address space, accesses straddling the 4G limit must cause a general protection fault (GP). Widening BT protect the 4G + 1 page and deliver page faults (PFs) there as general protection faults (GPs) instead.

Finally, widening BT needs to ensure that the context flat mode translations are executed under is also flat. Once again, widening BT can take advantage of sourcekeys to check translation context against execution context. Not only can sourcekeys keep track of the code base and limit, sourcekeys can also keep track of VCPU modes. If the VCPU mode for a CCF at execution time differs from the VCPU mode at translation time, widening BT retranslates. By creating a custom "flat mode" VCPU mode, widening BT uses sourcekeys to make sure its flat mode translations have flat execution contexts.

5.2.3 Fixed limits

Widening BT can further optimize segment checks by emitting a fast limit check for segments that are guaranteed to have the same segment limit at execution time as translation time.

Real mode cannot change its segment limits, so widening BT knows that the limit at translation time is the same as the limit at execution time. Knowing this, widening BT could avoid loading segment limit from memory and just compare against an immediate, but that would still require saving guest flags to avoid bashing them with the compare operation.

Widening BT can do better by effectively using the TLB to do the limit check without touching flags: widening BT arranges to load a byte from a region in memory such that if the address the guest is accessing is within the limit the load suceeds.

For example, to perform a fast limit check for 64K limit real mode segments (a common case in real mode) without bashing flags, the translation masks a pointer to 16 bits, then dereferences the pointer into 64K of virtual address space that is followed by a guard page. The 64K of virtual address space was chosen to be in the TC (translation cache), which is always mapped. Saving and restoring flags is an expensive little dance, so the fast real mode limit check saves a lot of time by avoiding a limit comparison. The "fast limit check" works for limits that aren't 64K. A fast limit check can be done for any limit that is guaranteed to be unchanged at execution time.

5.2.4 Sharing segment check code

Segment check code is bulky. Having translations share segment check code would save space. Making segment check code shared would also let widening BT easily choose between different types of segment checks dynamically; if widening BT wants to switch from a code block optimized for checking expand-up segments to a code block optimized for checking expand-down segments, widening BT merely needs to change which block of code the translation jumps to. In order to share code, widening BT needs to be able to jump to and return from the shared code block efficiently.

Widening BT could use indirect jumps to return from shared code. Although indirect jumps in general are relatively slow, CALLs and RETs are highly optimized indirect jumps. Using RETs to return from shared code would be efficient, but in order to use RETs, widening BT would have to push the return address onto a stack. However, when executing translations the guest stack is always live. Pushing the return value onto the guest's stack is a bad idea; the guest stack pointer might not even be pointing into valid memory. Widening BT wants to use a stack controlled by the monitor, in monitor memory. Widening BT could switch to using a monitor stack when sharing code, but setting the stack pointer `%rsp` is costly. Using an indirect jump would be cheaper than using RET with constant stack-switching between monitor and guest stacks.

Because widening BT translates all guest stack operations, it *is* possible to keep a live guest stack and a live monitor stack at the same time. Widening BT already translates all guest stack operations such as PUSH and POP into LEA and MOV based on `%rsp`. Widening BT could designate `%r14` as the guest stack pointer register and have stack operations manipulate memory based on `%r14`. Then widening BT could point `%rsp` at the monitor stack and perform CALLs and RETs to shared code with the monitor stack. Being able to stash the guest stack pointer in `%r14` demonstrates how useful it is to have those extra eight 64-bit registers at the VMM's disposal.

I didn't actually implement this code-sharing scheme because my focus was on modern operating systems, which have flat segments. This code-sharing scheme would make operating systems that use interesting segment features like expand-down run faster. OS/2, Windows 95 and 98 would benefit most from sharing segment check code.

Chapter 6

Guest system calls

Emulating guest system calls is the most overhead-intensive part of BT. A guest system call causes the VMM to switch between executing guest operating system code and guest user code. Because VMware's VMM binary translates guest operating system (privileged CPL0) code, but directly executes guest user (CPL3) code, any guest system call requires the VMM to switch from one mode of virtualizing to another.

The overhead of a guest system call is highly influenced by what CPL the *hardware* is running at when the VMM is running guest operating system (CPL0) code. With virtualization, the guest operating system may think it is running at CPL0, while the underlying hardware is running at a different CPL, such as CPL1. Let running the guest operating system with BT on hardware at CPL1 be called BT/CPL1. Let running the guest operating system with BT on hardware at CPL0 be called BT/CPL0.

Guest system calls under BT/CPL0 are almost twice as fast as under BT/cpl1. Before widening BT, VMware's VMM ran guest operating system (CPL0) code with BT/CPL1, running the translations on hardware at CPL1. After widening BT, the VMM runs guest operating system code translations with BT/CPL0. BT/CPL0 is only possible when performing widening BT.

This chapter explains how guest system calls are implemented, why BT/CPL0 has faster guest system calls than BT/CPL1, why BT/CPL0 is only possible with widening BT, and how BT/CPL0 is implemented in widening BT.

6.1 Guest system calls with BT/CPL0

A round-trip system call between guest operating system and guest user mode consists of a guest "system call" from guest CPL3 to guest CPL0 and a guest "system call return" from guest CPL0 to guest CPL3.

When running the guest under BT/CPL0, the hardware CPL is the same as the guest CPL. When the guest operating system thinks it is running at CPL0, the VMM is running in BT mode with CPL0 on hardware. When the guest user code thinks it is running at CPL3, the VMM is running in DE mode with CPL3 on hardware.

The VMM emulates guest system calls by transitioning between BT/CPL0 and DE at CPL3. When going from BT/CPL0 to DE at CPL3, the VMM uses *fast* IRET *to* DE. When going from DE at CPL3 to BT/CPL0, the VMM uses *fast fault-forwarding* or *fast syscalls*.

6.1.1 Fast IRET to DE

Fast IRET *to* DE is the path from BT/CPL0 to DE. When a guest attempts to change from CPL0 to CPL3 with an IRET the VMM checks whether conditions allow for the guest CPL3 code at the IRET destination to be directly executed. The VMM only IRETs to DE if segments are clean (the VMM's shadow segments match hardware segment values), IF is set and IOPL/VIP/VIF are clear, etc.

If DE conditions are met, the VMM performs the guest IRET to CPL3. The VMM executes a special block of assembly code that builds an IRET frame for transfering control to directly execute guest CPL3 code. When the VMM executes IRET with the frame, the processor begins to execute guest CPL3 code directly on hardware.

IRET isn't the only way to switch from CPL0 to CPL3; LRET, SYSRET and SYSEXIT are other ways, but their implementations are very similar to IRET's. I didn't bother implementing a fast SYSRET path for 32-bit code because in practice no 32-bit guest uses SYSRET.

6.1.2 Fast fault-forwarding

Older 32-bit operating systems such as Windows 2000 use INT*n*s to perform system calls. Newer 32-bit operating systems such as Windows XP use syscall instructions instead (see section 6.1.3). An INT*n* is basically a fault. The VMM uses the "fast fault-forwarding" path to handle both faults and INT*n* system calls from guest user mode (CPL3) to guest operating system.

When guest user code takes a fault/INT*n* in DE, the VMM needs to forward the fault to the guest by invoking the guest's fault handler. The VMM handles faults/INT*n*s from the guest with a special fault handler. The VMM's handler makes sure that the BT system is ready to run at CPL0 (BT/CPL0), that the guest CPL0 virtual CPU state is loaded, looks up the appropriate guest fault-handler, and invokes the guest's handler directly in BT/CPL0.

6.1.3 Fast syscalls

The INT*n* instruction generates a call to the interrupt or exception handler specified with the destination operand. Using INT*n* to perform system call is rather expensive because it invokes the interrupt handling mechanism. To optimize system calls, AMD developed SYSCALL and SYSRET and Intel developed SYSENTER and SYSEXIT. SYSCALL and SYSRET are companion instructions. SYSENTER and SYSEXIT are companion instructions. None of these instructions invoke the interrupt handling mechanism. They use MSRs (model specific registers) to switch to and from kernel CS, `%rip` and `%rsp`. Special handlers were made to handle guest syscalls.

6.2 BT/CPL0 is faster than BT/CPL1

Before widening BT, VMware's VMM translated and executed guest operating system code at CPL1, not CPL0. Before widening BT, guest system calls transitioned between BT/CPL1 and DE at CPL3. This section explains why guest system calls transitioning between BT/CPL0 and DE are almost twice as speedy as guest system calls transitioning between BT/CPL1 and DE.

6.2.1 Implementing virtual time is expensive with BT/CPL1

The first reason guest system calls with BT/CPL1 are slower than with BT/CPL0 is the way the VMM delivers virtual time to the guest. When running guest user code (CPL3) directly on hardware, the VMM needs to give the guest virtual time instead of real time. To give the guest virtual time, the VMM needs to intercept guest's RDTSC ("read time stamp") instruction and emulate it. The VMM can cause RDTSC instruction to fault for guest user code by setting a bit in the control register `%cr4`. This bit is called the `%cr4.tsc` bit. Setting `%cr4.tsc`=1 causes fault on RDTSC when CPL > 0, allowing the VMM to intercept guest RDTSCs.

Therefore, when entering DE from BT/CPL1, the VMM sets `%cr4.tsc` bit. Because the VMM directly executes guest code at CPL3 on hardware (DE at CPL3), setting `%cr4.tsc` causes guest RDTSCs to trap out, allowing the VMM to give the guest virtual time.

However, the VMM doesn't want to trap on BT's RDTSCs while using BT/CPL1, so the VMM clears `%cr4.tsc` when exiting DE. Setting and clearing `%cr4.tsc` is an expensive 25% of the system call cost. If the VMM BTs at CPL0, the VMM wouldn't have to toggle the `%cr4.tsc` bit. By always using BT/CPL0, the VMM can just leave `%cr4.tsc` set all the time because RDTSCs won't trap at CPL0.

6.2.2 VMM needs to perform privileged operations at CPL0

The second reason guest system calls with BT/CPL1 are slower is because somewhere between the BT to DE transition, the VMM must go to CPL0 to perform some privileged operations anyway; one such privileged operation is setting `%cr4.tsc`, mentioned in section 6.2.1. The transition between BT/CPL0 and DE involves only one hardware CPL transition between hardware CPL0 to CPL3. On the other hand,

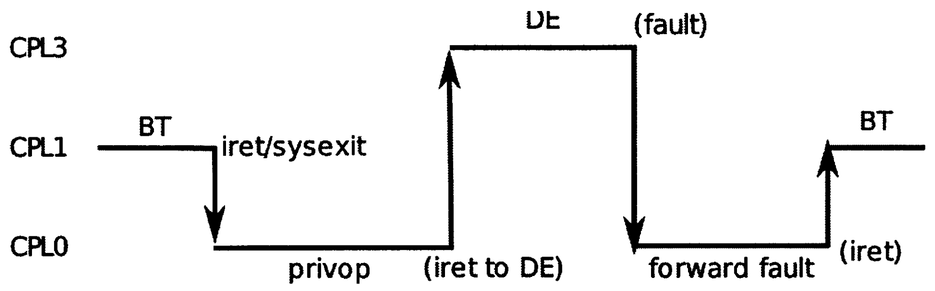

Figure 6-1: There are 4 vertical CPL transitions. If the VMM runs binary translations at CPL0, only the 2 transitions between CPL0 and CPL3 would be needed. The transitions between CPL1 and CPL0 probably account for 20% of guest system call cost.

the transition between BT/CPL1 and DEmust detour into hardware CPL0, which takes an extra CPL transition.

With BT/CPL1, entering and then exiting DE for a system call involves a total of four CPL transitions (see Figure 6-1). On the guest system call return, the VMM must leave BT/CPL1, drop to CPL0 to perform privileged operations to prepare for the fast IRET to DE. When the VMM receives the the guest system call from DE, the VMM is at CPL0, but must transition to CPL1 to run guest handler with BT/CPL1.

For a round-trip guest system call, the VMM goes through 4 CPL transitions, 2 of which are required because BT happens at CPL1. These 2 transitions between CPL1 and CPL0 are about 20% of the system call cost. The 2 transitions between CPL0 and CPL1 could be avoided if BT occurred at CPL0 (Figure 6-1).

With BT/CPL0, the VMM does not need to toggle `%cr4.tsc` bit (25% of system call cost) or transition between CPL1 and CPL0 (20% of system call cost). These estimates show that BT/CPL0 will save about 45% of system call costs, almost doubling the speed of guest system calls.

6.3 BT/CPL0 only possible with widening BT

Unfortunately, BT/CPL0 is only possible for widening BT which translates x86-32 to x86-64 code. In order to use BT/CPL0, widening BT must be able to protect the VMM and switch from the guest stack to the VMM stack on fault. Switching to

VMM stack on fault means the exception frame will be pushed onto the VMM stack. Switching to VMM stack is necessary because the VMM needs to intercept all faults and selectively forward faults to the guest; exception frames should not be pushed directly to guest stack. VMM state must never be visible to the guest.

VMware's VMM has to use BT/CPL1 for VMM32 (x86-32 to x86-32 BT) and 64-bit guest code in VMM64 (x86-64 to x86-64 BT) because in these situations, the VMM cannot both protect the VMM address space and switch to VMM stack on fault. The reasons the VMM currently uses BT/CPL1 for VMM32 and 64-bit guest operating system code running under VMM64 are completely different.

VMM32 cannot use BT/CPL0 because:

1. If an exception is taken during BT/CPL0, x86-32 (legacy/compatibility mode) does not switch stacks. If an exception is taken during BT/CPL1, x86-32 changes privilege level to CPL0 and switches stacks. Inter-CPL transitions cause stack switches in x86-32 [2].

2. On the other hand, VMM32 can use segmentation to protect VMM address space in x86-32 mode.

VMM64 cannot BT/CPL0 because:

1. AMD 64-bit mode segmentation does not work at CPL0; segmentation does work for CPLs 1 through 3. Without segmentation at CPL0, VMM64 cannot efficiently protect the VMM address space from 64-bit guest code.

2. Unlike VMM32, VMM64 can use IST field (Interrupt Stack Table field) to switch stacks even if it is running translations at CPL0. Gate descriptors have an IST field, which if set, always causes a stack switch on exception.

The VMM needs to protect the VMM address space and switch to a VMM stack on exception. Widening BT can use address-size prefixes to restrict the 32-bit guest to low 4G of 64-bit address space, protecting the VMM address space. Widening BT allows the VMM to set the IST field to switch stacks. Therefore only widening

BT running with x86-64 VMM can benefit from the BT/CPL0 guest system call optimizations. However, if AMD simplifies their x86-64 segmentation specifications to allow segment limits at CPL0, the VMM could use BT/CPL0 when translating 64-bit guest code, allowing 64-bit guests to run under BT/CPL0 to receive a similar performance boost.

6.4 Implementing BT/CPL0

Under widening BT, the VMM can always protect the VMM address space with address-size prefix, so protecting the VMM when running translated guest code at CPL0 is no problem. Stack-switching can be implemented with interrupt stack table (IST), available under 64-bit mode (x86-64). This section describes how to emulate inter-CPL stack-switching with the gate descriptors' IST field.

If the VMM sets the IST field for a gate descriptor, x86-64 will *always* switch stacks on exception. The VMM has to be particularly careful about handling nested exceptions. Setting the IST field causes x86-64 to always reset the stack, so the VMM must prevent new exception frames from clobbering the exception frame it is currently handling.

To simulate nested exceptions, the VMM lets the IST mechanism push the exception frame into a buffer above the VMM's CPL0 stack (see Figure 6-2). Before the VMM handles the exception, it copies the exception frame into the VMM's stack. If the exception is not nested, the VMM copies the exception frame to the beginning of VMM's stack. If the exception is nested, the VMM copies the exception frame to the end of the VMM's stack. Because the hardware IST mechanism only writes to the buffer, exceptions will never clobber frames on the VMM's stack.

The buffer is named VMM_STACK_TOP_64_HW. The VMM stack is named VMM_STACK_TOP. Between the buffer and the VMM stack is a region named "No Man's Land", which is full of zeros. Checking that No Man's Land is always blank (completely zeroed) allows the VMM to be sure that exception frames aren't accidentally overflowing the buffer VMM_STACK_TOP_64_HW.

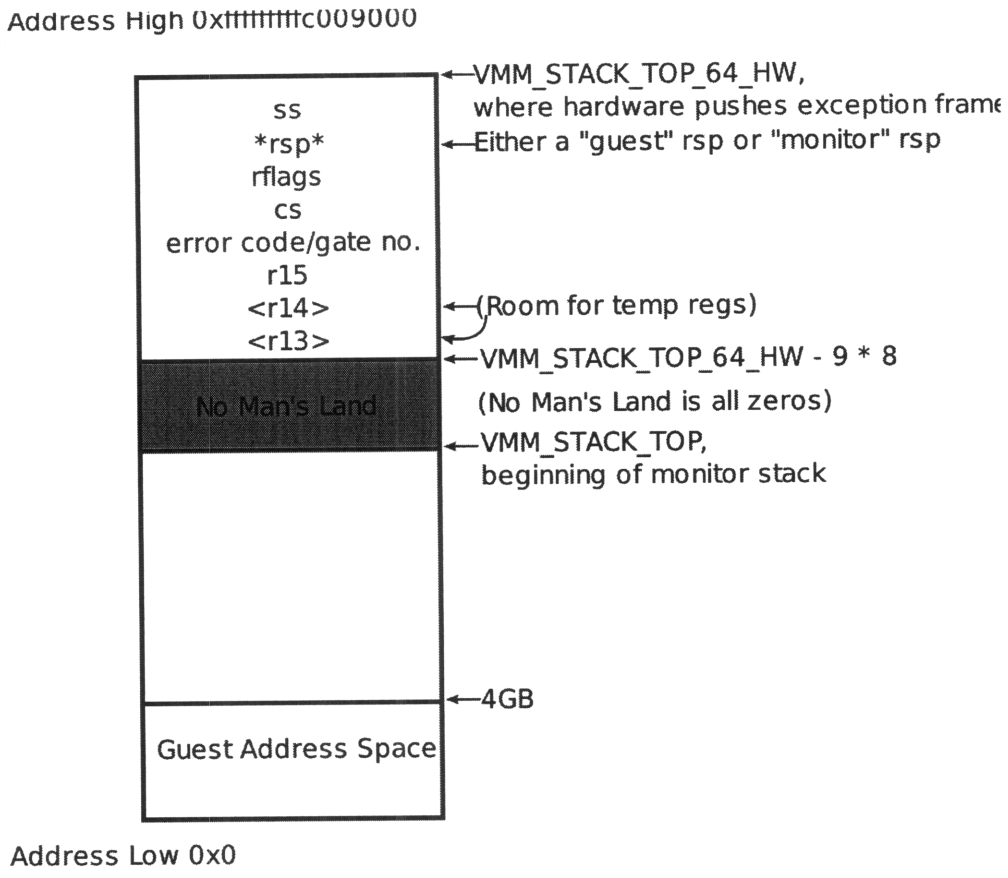
Address High 0xffffffffc009000
ss
rsp
rflags
cs
error code/gate no.
r15
<r14>
<r13>
No Man's Land
Guest Address Space
Address Low 0x0
VMM_STACK_TOP_64_HW,
where hardware pushes exception frame
Either a "guest" rsp or "monitor" rsp
(Room for temp regs)
VMM_STACK_TOP_64_HW - 9 * 8
(No Man's Land is all zeros)
VMM_STACK_TOP,
beginning of monitor stack
4GB

Figure 6-2: Address space right after interrupt causes hardware to push an exception frame into buffer at `VMM_STACK_TOP_64_HW`.

There are two cases, nested and unnested, when handling exceptions (see Figure 6-3):

1. The exception happened while running on a guest stack. The guest stack is live in `%rsp` only if the VMM is in BT or DE mode. One way to distinguish between guest stacks and VMM stacks is that 32-bit guest stacks are all under 4GB, while VMM stacks are all above 4GB. However, there is no way to IRET from 64-bit kernel mode (with 64-bit stack size) to 16-bit mode without leaking the top 48 bits of the kernel stack pointer; in this case, the VMM may be running on a 16-bit guest stack but actually have a 64-bit value greater than 4GB in `%rsp`. To address this problem, the VMM uses `%ss` (stack segment register) to tag whether it is running in BT or DE. If the VMM is running with a guest stack, `%ss` will either contain 0x4028, the BT/CPL0 SS, or select a CPL3

SS, which implies the VMM is running in DE. Once the VMM has determined that it is running with the guest stack, it copies `ExcFrame` (exception frame) to the beginning of the VMM CPL0 stack (`VMM_STACK_TOP_64`). This emulates a stack switch to the VMM CPL0 stack.

2. The exception happened while the VMM is running on a VMM stack. The VMM knows it is running on a VMM stack if SS is a VMM SS (either 0x4018 or 0). The VMM copies `ExcFrame` to the end of the VMM stack it was running on; it copies the `ExcFrame` to just below `ExcFrame` rsp value. Since the `ExcFrame` is copied to the end of the stack the VMM was running on, to the exception handler, it is as if the VMM didn't switch stacks. This means that any `%ss` other than `MONITOR_CPL0_64BIT_SS` (0x4018) or null SS (0) will clobber the beginning of `VMM_STACK_TOP_64`, so the VMM needs to be careful to track what possible `%ss` values can exist in its code.

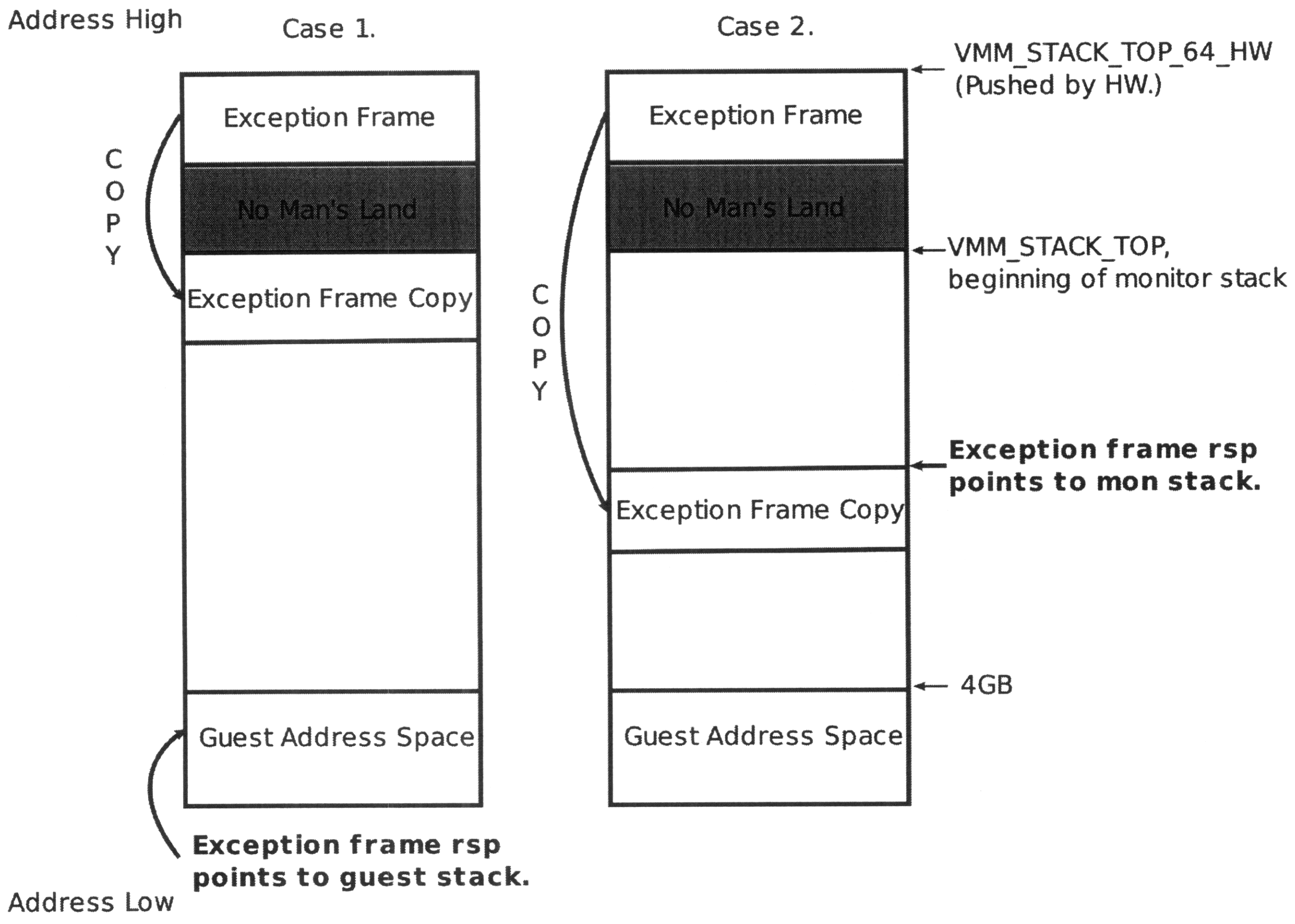

Figure 6-3: Case 1: Exception is not nested; this implies the VMM is running on guest stack and needs to emulate stack-switch to VMM stack. Case 2: Exception is nested; the VMM needs to emulate staying on the VMM stack by copying the new exception frame to the bottom of the VMM stack.

Chapter 7

Performance results

While the main focus of widening BT is to simplify the virtual machine monitor, widening BT should enhance performance overall. I ran OS install, system-call-intensive micro-benchmarks, and PassMark benchmarks to determine whether the performance of widening BT's x86-64 translations is good enough for VMware to run all guest code in x86-64 mode (long mode) all the time.

I compare the performance of 32-bit guest operating systems running on an x86 64-bit CPU:

1. On x86-64 mode hardware with x86-64 monitor (VMM64).

2. On x86-32 mode hardware with x86-32 monitor (VMM32).

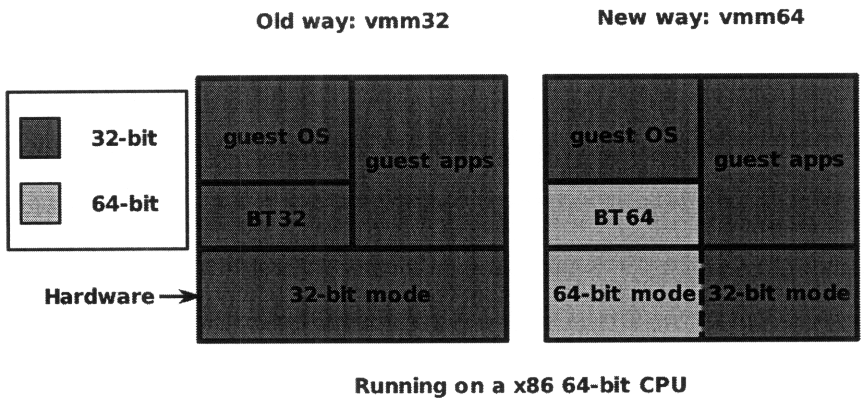

Figure 7-1: I compare the performance of BT to x86-32 target code under VMM32 with the performance of BT to x86-64 target code under VMM64.

7.1 OS installs

The OS install benchmark times how long it takes to install a particular OS followed by booting and shutting down the OS at least once. It contains a mix of I/O and computation, is easy to run for different OSes, and is fairly consistent run to run. OS installs model real workloads more closely than purely CPU-intensive or system-call-intensive benchmarks.

7.1.1 Understanding results

The OS install benchmark was run on an AMD Quad-core Opteron Processor 2356 at 2.3GHz. The OS install benchmark results are presented as bar charts:

- Y axis: different OSes
- X axis: the time ratio of VMM64/VMM32; bars shorter than 1 mean VMM64 is faster, while bars longer than 1 mean VMM32 is faster.
- The OSes are sorted with those that run faster under VMM64 near the top and those that run faster in VMM32 near the bottom.
- Modern OSes are expected to be near the top while the older OSes, which use non-flat segmentation, are expected to be near the bottom.

7.1.2 Windows installs

Windows ME, 95 and 98 all use non-flat segmentation; the cost of emulating non-flat segmentation for them is considerable. They appear to use non-flat segment FS for thread local storage [9]. Other Windows operating systems, especially the more modern ones such as XP, use flat segments and run faster when virtualized under VMM64 than VMM32.

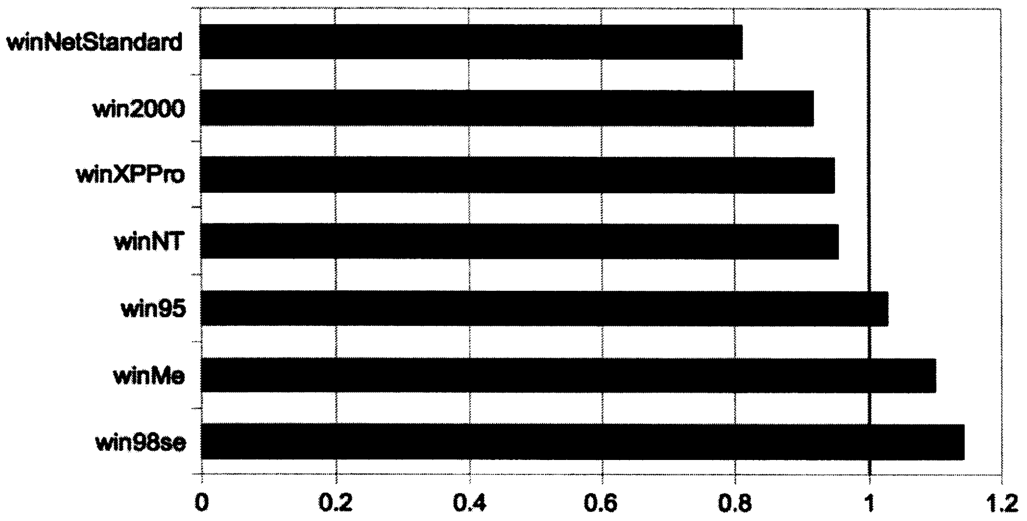

Figure 7-2: OS install benchmark for Windows operating systems. Shorter bar means VMM64 runs faster than VMM32.

7.1.3 Linux installs

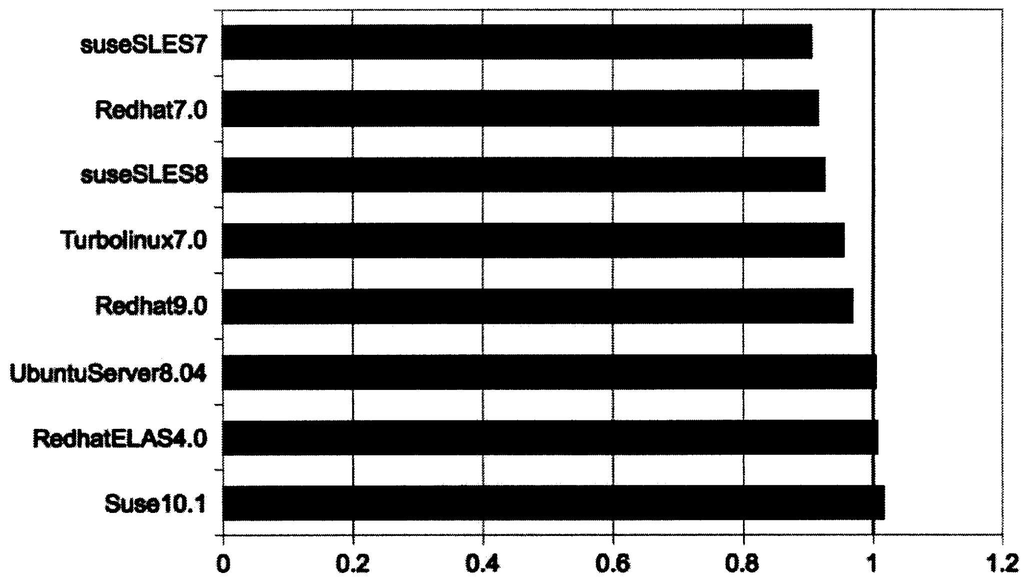

Figure 7-3: OS install benchmark for Linux operating systems. Shorter bar means VMM64 runs faster than VMM32.

Linux guests don't use non-flat segmentation, so there is no overhead for virtualizing 32-bit Linux guests under x86-64 mode.

7.1.4 Corner case: UnixWare

For most operating systems, the performance of VMM64 is close to that of VMM32 under OS install benchmark. However, by having disjoint guest and monitor address spaces, VMM64 can avoid performance cliffs that trouble VMM32.

UnixWare, a Unix operating system maintained by The SCO Group [8], is an operating system that suffers from frequent guest accesses to the monitor address space. UnixWare keeps the stack high up in the address space. When UnixWare is run as a guest operating system under BT with VMM32, its stack overlaps with VMM32's address space, causing many faults and degrading performance. In fact, UnixWare faults so many times under BT in VMM32 that it hangs while installing. However, if run under VMM64 with widening BT, the UnixWare guest and monitor address spaces don't overlap, avoiding address-space contention. UnixWare installs normally under VMM64.

Other operating systems with a high stack that overlaps monitor address space include OS/2 and Mac OS X.

7.2 Micro-benchmarks

While OS installs approximate real workloads, micro-benchmarks such as forkwait, getpid and switch test workloads that BT is particularly bad at. All of these micro-benchmarks make system calls. In general, system calls are the most overhead-intensive operations when virtualizing with BT(emulating non-flat segmentation in x86-64 is also costly, but few operating systems use non-flat segmentation). Most of software virtualization overhead comes from system calls. Most other instructions are non-privileged and can be executed directly on hardware or binary translated identically, incurring very little overhead.

Here are descriptions of each of the micro-benchmarks:

Forkwait: Executes a fork system call followed by a wait system call. The fork system call copies the process state; the new process is called the "child" process and the old process is called the "parent" process. The wait system call blocks the parent process until the child dies. Unix shells often perform forkwait; much of the time spent in shell scripts is in fork. Forkwait's performance depends on the time it takes to set up the new process's page tables.

Getpid: Fetches the process ID. This benchmark makes one system call which immediately returns to the application with the process ID. This benchmark most closely measures the time it takes to perform a system call.

Switch: This benchmark measures Java thread switching overhead. Switching Java threads causes the operating system to switch threads. The threads form a ring of size 2. One thread signals the next thread to form a context switch.

Forkwait and getpid are the most important benchmarks because they matter more in real workloads.

7.2.1 Understanding results

I measured and compared performance of micro-benchmarks between VMM64 and VMM32 on Windows 2000 and Windows XP Pro. I compared the performance under Windows 2000 with the performance under Windows XP Pro because they use different instructions and mechanisms to perform system calls. Older operating systems such as Windows 2000 tend to use INT*n*, while newer operating systems such as Windows XP Pro use SYSENTER/SYSEXIT syscall instructions.

I also ran each set of micro-benchmarks on an Intel machine and an AMD machine. I did this because AMD and Intel have different implementations of syscall instructions. The benchmarks are run on an AMD Quad-core Opteron Processor 2356 at 2.3GHz and an Intel iCore 7 at 3.2GHz.

New Windows (such as XP Pro) and Linuxes use SYSENTER/SYSEXIT syscall instructions, which are available for both Intel and AMD CPUs in 32-bit mode. SYSCALL/SYSRET syscall instructions only work in 32-bit mode on AMD CPUs.

Figure 7-4 shows the micro-benchmark results as bar charts:

- Y axis: different micro-benchmarks (forkwait, getpid, switch).
- X axis: the time ratio of VMM64/VMM32; bars shorter than 1 mean VMM64 is faster, while bars longer than 1 mean VMM32 faster.

7.2.2 Windows XP Pro on AMD machine

The getpid and switch benchmarks under VMM64 are slower on the AMD machine because 64-bit AMD does not implement the SYSENTER and SYSEXIT syscall instructions. Emulating SYSENTER and SYSEXIT with widening binary translation results in some overhead; this affects getpid micro-benchmark the most.

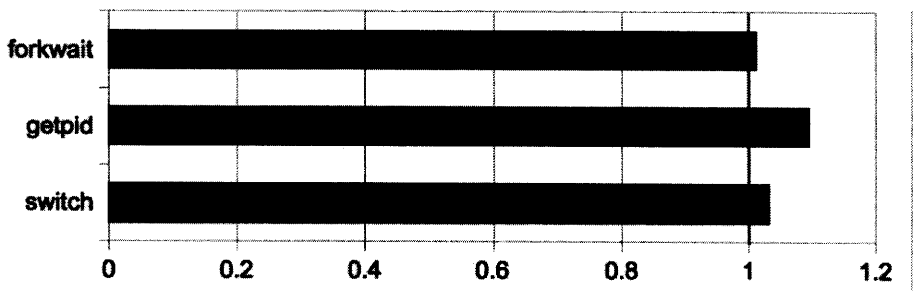

Figure 7-4: Micro-benchmarks for Windows XP Pro on AMD machine. A bar shorter than 1 means VMM64 runs faster than VMM32.

7.2.3 Windows XP Pro on Intel machine

The getpid and switch benchmarks under VMM64 are faster on the Intel machine than the AMD machine because widening BT can use the SYSENTER and SYSEXIT instructions in Intel 64-bit mode; widening BT doesn't incur overhead from having to emulate SYSENTER and SYSEXIT as it does on AMD machine.

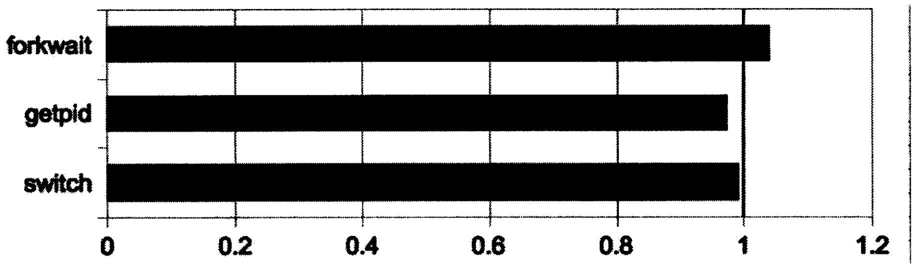

Figure 7-5: Micro-benchmarks for Windows XP Pro on Intel machine. Shorter bar means VMM64 runs faster than VMM32.

7.2.4 Windows 2000 on AMD machine

Windows 2000 uses INT*n* instructions to perform system calls. The disparity between benchmark performance on Intel machine and AMD machine should be less pronounced for Windows 2000.

When running the getpid micro-benchmark under VMM64, there is higher overhead. The overhead does not come from emulating SYSENTER and SYSEXIT (Windows 2000 uses neither instruction), but rather from the fact that VMM64's fault/INT*n* handling path is simple C while VMM32's fault/INT*n* handling path is highly optimized assembly.

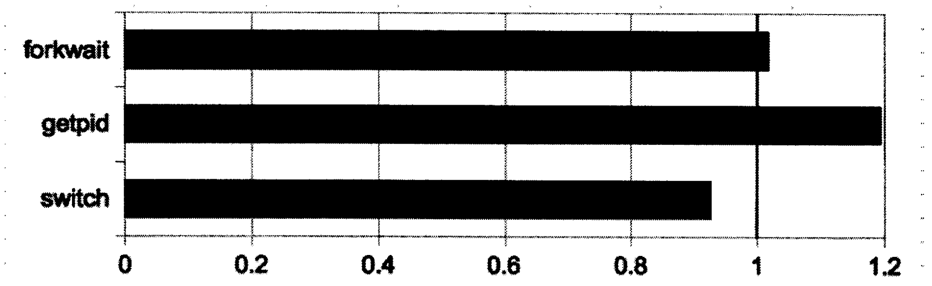

Figure 7-6: Micro-benchmarks for Windows 2000 on AMD machine. Shorter bar means VMM64 runs faster than VMM32.

7.2.5 Windows 2000 on Intel machine

For Windows 2000 micro-benchmarks, VMM64's performance is worse on the Intel machine than on the AMD machine.

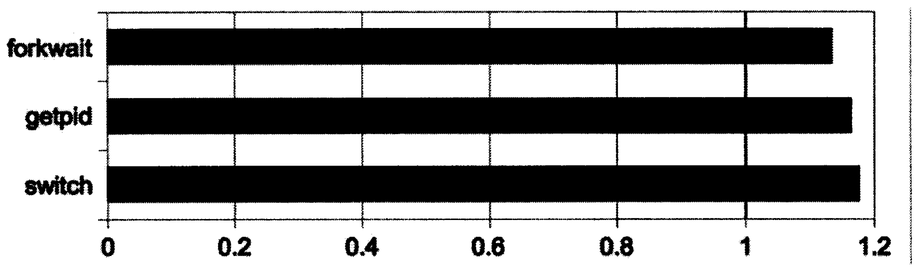

Figure 7-7: Micro-benchmarks for Windows 2000 on Intel machine. Shorter bar means VMM64 runs faster than VMM32.

7.3 PassMark

PassMark PerformanceTest is a hardware benchmark utility. I ran the PassMark CPU benchmarks under Windows XP Pro on AMD Quad-core Opteron Processor 2356 at 2.3GHz. For the PassMark CPU benchmarks, the performance of VMM64 and VMM32 are very similar (see Table 7.1). This is because for non-privileged instructions, the performance is close to native and whether the code is run under VMM64 or VMM32 doesn't matter.

While PassMark CPU benchmarks run non-privileged code, PassMark Graphics 2D benchmarks execute system calls. There is greater disparity between the performance of VMM64 and VMM32 on PassMark Graphics 2D benchmarks (see Table 7.2).

Table 7.1: PassMark CPU benchmark run under Windows XP Pro on AMD Quad-core Opteron Processor 2356 at 2.3GHz.

CPU benchmark	VMM64	VMM32	Unit
Integer Math	71.4	70.8	Millions of operations per second
Floating Point Math	408.3	400.2	Millions of operations per second
Find Prime Numbers	241.1	241.2	Operations per second
SSE/3DNow!	1838.7	1816.3	Million matrices per second
Compression	1965.2	1962.6	Kbytes processed per second
Encryption	17.1	17	Mbytes transferred per second
Image Rotation	311.9	312.2	Image rotations per second
String Sorting	1246.7	1256	Thousand Strings per second
Mark	535.8	532.1	Composite average of other results
PassMark Rating	144.7	143.7	Composite average of other results

Table 7.2: Graphics 2D PassMark run under Windows XP Pro on AMD Quad-core Opteron Processor 2356 at 2.3GHz.

2D Benchmark	VMM64	VMM32	Units
Lines	26.1	31.1	Thousands of lines drawn per second
Rectangles	4.19	1.3	Thousands of images drawn per second
Shapes	5.3	7.1	Thousands of shapes drawn per second
Fonts and Text	56.2	84.3	Operations per second
GUI	97.8	161.1	Operations per second
Graphics Mark	105.3	147.8	Composite average of other results
PassMark Rating	14.7	20.7	Composite average of other results

7.4 Evaluation of performance

The purpose of running the performance benchmarks is to make sure that widening BT performs well enough for VMware to run guest code from all modes under x86-64 mode with VMM64 all the time. Based on OS install results, the verdict is that widening BT performs well enough, even though widening BT incurs some overhead for system calls and emulating segmentation. Widening BT will allow VMware to save developing time by discarding the VMM32 monitor mode once 64-bit CPUs are ubiquitous.

Chapter 8

Conclusion

The goal of running 32-bit x86 guests in x86-64 mode is accomplished by extending VMware's virtual machine monitor's binary translation infrastructure to translate x86-32 code to x86-64 code via a process called "widening BT". Widening BT allows the VMM to take advantage of 64-bit CPUs for virtualizing x86-32 code. Obvious x86-64 mode benefits include the large 64-bit address space and twice as many general purpose registers. After this change, VMware's VMM runs 32-bit guest operating systems in x86-64 mode under a x86-64 monitor VMM64 instead of a x86-32 monitor VMM32.

Translating 32-bit guest operating systems to x86-64 mode simplifies the virtual machine monitor and enhances overall performance. A performance gain can be seen in the OS install benchmark (Section 7.1), although VMM64 performs worse on system calls than VMM32 (Section 7.2). The OS install benchmark more closely resembles real workloads than the system call micro-benchmarks; system calls take up a relatively small fraction of real workloads. Widening BT also needs to emulate x86-32 segmentation in x86-64 mode, which incurs some overhead, especially for old operating systems like OS/2. The performance checks were important to make sure that the performance of 32-bit guest operating systems in x86-64 mode under VMM64 is *good enough* to switch over to running x86-32 code solely under VMM64. The human cost savings of only having to maintain one virtual machine monitor (just VMM64) instead of two (VMM64 *and* VMM32) is much more important than the performance gains provided by VMM64. With widening BT, VMware can discard VMM32 once 64-bit CPUs are ubiquitous.

Widening BT shows hardware support is unnecessary for running 32-bit guests. Widening BT shows how to emulate 32-bit legacy segmentation in 64-bit mode; segmentation hardware is used infrequently enough in modern operating systems that should it be dropped, software segmentation can replace it. By translating x86-32 code to x86-64 code, widening BT reduces hardware's burden of backwards-compatibility. Future work can explore more consequences of using binary translation as an abstraction layer between software and hardware.

Bibliography

[1] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 2–13, New York, NY, USA, 2006. ACM.

[2] Advance Micro Devices. *AMD64 Architecture Programmer's Manual*, September 2007. Volume 2, Chapter 8.7 Legacy Protected-Mode Interrupt Control Transfers.

[3] Ole Agesen. Binary translation of returns. In *WBIA 2006: Workshop on Binary Instrumentation and Application*, pages 1–7, 2006.

[4] Apple. Rosetta. In *Universal Binary Programming Guidelines, Second Edition.*, 2007.

[5] Fabrice Bellard. Qemu, Jul 2008. `http://bellard.org/qemu/`.

[6] Intel, Intel Corporation, P.O. Box 5937, Denver, CO 80217-9808. *Intel 64 and IA-32 Architectures Software Developer's Manual*, April 2008. Volume 1, section 3.4.3.

[7] Intel, Intel Corporation, P.O. Box 5937, Denver, CO 80217-9808. *Intel 64 and IA-32 Architectures Software Developer's Manual*, April 2008. Volume 3A, Chapter 3 Memory management.

[8] Santa Cruz Operation. Introducing SCO UnixWare 7.1.4, April 2009. http://www.sco.com/products/unixware714/.

[9] Matt Pietrek. Under the hood. *Microsoft Systems Journal*, May 1996. http://www.microsoft.com/msj/archive/s2ce.aspx.

[10] Byung sun Yang, Soo mook Moon, Seongbae Park, Junpyo Lee, Seungil Lee, Jinpyo Park, Yoo C. Chung, Suhyun Kim, Kemal Ebcioglu, and Erik Altman. Latte: A java vm just-in-time compiler with fast and efficient register allocation. In *Parallel Architectures and Compilation Techniques, 1999. Proceedings. 1999 International Conference on*, pages 128–138, Newport Beach, CA, USA, 1999.

[11] VMware. Virtualization Basics. http://www.vmware.com/technology/virtual-machine.html, 2009.