VMM Based OS Profiler

BITS ZG629T: Dissertation

By

Ajay Harikumar
2005 HZ13069

Dissertation work carried out at
Intel Technology India Pvt. Ltd
Dissertation submitted in partial fulfillment of the requirement of
M.S. (Software Systems) degree programme
Under the Supervision of
Ravindra Bodhe
Engineering Manager, System Research Centre
Intel Corporation

BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE
PILANI (RAJASTHAN)

September 2007

CERTIFICATE

This is to certify that the Dissertation entitled “VMM based OS profiler” and submitted by Ajay Harikumar having ID-No. 2005 HZ13069 for the partial fulfillment of the requirements of M.S. Software Systems degree of BITS, embodies the bonafide work done by him under my supervision.

_____

Signature of Supervisor

Place: Bangalore

Date: _____

Ravindra B Bodhe
Engineering Manager
System Research Centre
Intel Technology India Pvt. Ltd.
Bangalore
India

Birla Institute of Technology & Science, Pilani

Distance Learning Programmes Division

First Semester 2007-2008

BITS ZG629T: Dissertation Outline

BITS ID No.	**:**	**2005 HZ13069**
NAME	**:**	**Ajay Harikumar**
EMAIL ADDRESS	**:**	**Ajay.Harikumar@Intel.com**
EMPLOYING ORGANIZATION	**:**	**Intel Technology India Pvt. Ltd.**
SUPERVISOR'S NAME	**:**	**Ravindra B Bodhe**
DISSERTATION TITLE	**:**	**VMM Based OS Profiler**

ABSTRACT

Dissertation Title: **VMM Based OS Profiler** (Design and implementation of a light-weight Virtual Machine Monitor (VMM) using Virtualization Technology for the purpose of profiling the OS running on top of the VMM)

Supervisor: Ravindra Bodhe

Name of student: Ajay Harikumar
Semester: Fourth
ID No. : 2005HZ13069

Virtualization (or more specifically Platform Virtualization) is a technology that is gaining ground in servers and PCs as the preferred mechanism for running multiple Operating Systems concurrently. Processor Vendors like Intel and AMD have provided hardware features designed to facilitate Virtual Machine Monitors and allow them to run multiple Guest OS

The purpose of this project is to design and develop a light weight Virtual Machine Monitor that runs underneath the Guest OS and collect statistics on the Guest OS and workloads using hardware provided performance monitoring registers, Virtual Machine enters and exits, page tables and IO calls traps. The VMM then collects and provide information that can be used to identify Guest OS and Workload hot spots, bottlenecks and inefficient implementations. This information would allow the System Architects, OS developer and Application Developers to modify the System Architecture, OS and the Workloads respectively for better performance.

The advantage of this is that OS and Workload profiling can be done by an independent software stack that does not depend on the OS for execution. Thus any Operating System and the accompanying Workload can be profiled using this light-weight VMM without having to port the profiling stack to each OS environment and also without having to modify the OS or the Workload. The VMM profiling approach also avoids the overheads of running the profiling tool on top of the OS and in turn coloring the OS characteristics.

Broad Academic Area of Work:
Operating Systems

Key words (Specify the keywords in alphabetical order)

- Binary Translation
- BIOS
- Chipset
- EFI
- Guest Operating System
- Hosted Architecture
- Hypervisor
- IA32
- Instrumentation
- Intel VT Technology
- Operating Systems
- Para-virtualization
- Performance Monitoring Registers
- Processor

- Profiling
- Statistical Profilers
- System Architecture
- Trace
- Virtual Machine
- Virtual Machine Monitor
- Virtualization
- VMM
- VMWare
- Xen

ACKNOWLEDGEMENT

I take this opportunity to express my gratitude to my guide, Ravindra Bodhe, Engineering Manager, Intel, for his exemplary guidance, monitoring and constant encouragement throughout the course of this Dissertation work. This project and my course would not have been possible without his support and help.

Special thanks to Sasikanth Avancha for bringing in his unique perspective to the project, broadening the scope of the work and for all the long hours of discussion on how to make the LWVMM do more and be more useful. Without his guidance the LWVMM would have been just another unused tool.

Sincere appreciation is extended to Philip Lantz, Sebastian Schoenberg for their suggestions and help in guiding this project.

Table of Contents

Table of Figures

Chapter 1: Introduction

Virtualization (or more specifically Platform Virtualization) is the creation of multiple virtual machines on a single physical platform allowing the ability to run multiple Operating Systems concurrently. A Virtual Machine Monitor is a software layer that sits between the hardware and the Operating Systems (also known as Guest OS) and emulates an entire platform allowing the guest OSs to run concurrently in an isolated manner.

Virtualization has been around in Computer Systems since the 1960s but has become mainstream only since early this century with the introduction of commercial VMM (Virtual Machine Monitors) for x86 architecture and the subsequent support in x86 hardware by companies like Intel and AMD allowing for virtualization to be achieved in Desktops and Servers alike. Intel processors have started supporting VT (Virtualization Technology) as a Processor feature allowing for fully virtualized (unmodified) Operating Systems to run on Intel VT enabled systems. Gartner has described virtualization as the single most important technology this decade it is estimated that around 10% of the x86 servers (which themselves constitute around 90% of the server market) will use virtualization. Considering that all future x86 processors will ship with Virtualization Technology enabled, it is estimated that around 70% of all servers will make use of virtualization by 2010.

As mentioned above, Intel Processors have started supporting VT in Processors since 2005. As will be seen in subsequent chapters, Virtualization Technology allows a thin layer of software (the VMM) to sit between the hardware and the Operating System. The VMM can thus intercept any interaction that happens between the underlying hardware and the Operating System.

The objective of this project is to design and develop a lightweight VMM that sits on an Intel VT supported System, underneath the fully virtualized Operating System and collects statistics of the behavior of the OS. The VMM can thus provide information that can be used to profile the OS and identify hot spots, bottlenecks and inefficient implementations. This information would allow the OS developer to tune the OS for better performance.

The advantage of this approach to profiling is that OS profiling can be done by an independent software stack that does not sit on and thus depend on the OS for execution. Thus any Operating System can be profiled using this light-weight VMM without having to port the profiling stack to each OS environment. The VMM profiling approach also avoids the overheads of running the profiling tool on top of the OS and in turn coloring the OS characteristics.

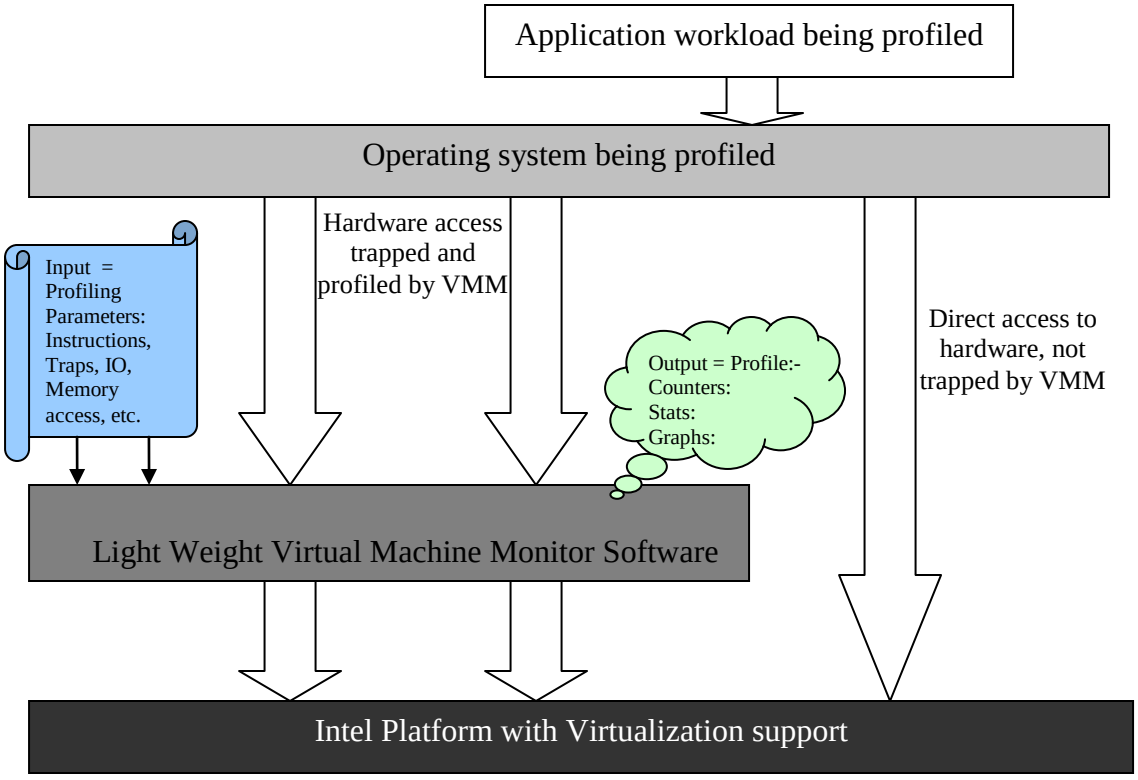

Figure 1: Architecture being proposed for Profiling

The belief is also that this lightweight VMM Framework can be used for other functionalities like implementing RAS (Reliability, Accessibility, Serviceability) and system management features like memory and IO sparing, can act as a virus scanner (network packet analysis etc.) to protect the system and also for implementing OS checkers (safe trusted place to stand for OS measuring agents)

The remaining of this document is as follows

Chapter 2 gives a brief introduction to Intel processors and covers details of Intel Architecture that is important for this project

Chapter 3 introduces Virtualization Technology and explains the implementation of virtualization in Intel Processors

Chapter 4 introduces profiling and the different methods of profiling possible

Chapter 5 introduces EFI which would be used as the build and runtime environment to create the Lightweight VMM

Chapter 6 talks about the Lightweight VMM, what it can and cannot do. It will include details of the flow

Chapter 7 describes the hardware setup used for developing and running the LWVMM

Chapter 8 goes into details of the actual implementation including details of the PC boot flow, EFI boot flow, the LWVMM boot flow and the VMM details

Chapter 9 summarizes the results of running the LWVMM

Chapter 10 is on Conclusions and Recommendations

Chapter 11 gives directions for future work on this project

Chapter 2: Introduction to Intel Processors

Intel Virtualization Technology is a set of hardware features designed to facilitate Virtual Machine Monitors and allow them to run Guest OS without having to do either para-virtualization or Binary Translation.

Intel has two main processor lines, the Intel Core based/Pentium and the Intel Itanium processor lines. The Intel Core/Pentium refers to processors based on x86 architecture (8086, 80286, 80486, Pentium, Core 2 Duo etc.). These are used in a majority of the laptops, desktops and servers. The Itanium line is EPIC (Explicitly Parallel Instruction Computing) VLIW like, systems that are mostly used in the high end server systems. Here we concentrate on Intel's brand of x86 processors Core 2 Duo (Core2 refers to the Processor Architecture, while the Duo refers to two cores in a single Processor package). These processors are 64bit and support Virtualization Technology.

Intel Processors support four levels of privileges called as Rings. The Ring 0 level has the highest privilege where usually the OS kernel executes, while Ring 3 level is where the applications execute. Ring 1 is usually meant for driver code to execute, while services are meant to run on Ring 2. However for most Operating Systems, the OS, the drivers and the services run at Ring 0 while the applications run at Ring 3.

For a complete introduction of Intel Core 2 Duo (EM64T) processors, refer to [12].

IA-32 architecture (beginning with the Intel386 processor family) provides extensive support for operating-system and system-development software. This support offers multiple modes of operation, which include:

- Real mode
- Protected mode
- Virtual 8086 mode
- System management mode.

These are sometimes referred to as legacy modes.

Intel 64 architecture supports almost all the system programming facilities available in IA-32 architecture and extends them to a new operating mode (IA-32e mode) that supports a 64-bit programming environment. IA-32e mode allows software to operate in one of two sub-modes:

- 64-bit mode supports 64-bit OS and 64-bit applications
- Compatibility mode allows most legacy software to run; it co-exists with 64-bit applications under a 64-bit OS.

This chapter describes the system registers that are used to set up and control the processor at the system level and gives a brief overview of the processor's system-level (operating system) instructions.

All Intel 64 and IA-32 processors enter real-address mode following a power-up or reset. Software then initiates the switch from real-address mode to protected mode. If IA-32e mode operation is desired, software also initiates a switch from protected mode to IA-32e mode.

Overview of the System-Level Architecture

System-level architecture consists of a set of registers, data structures, and instructions designed to support basic system-level operations such as memory management, interrupt and exception handling, task management, and control of multiple processors. Figure 2 provides a summary of system registers and data structures that applies to IA-32e mode.

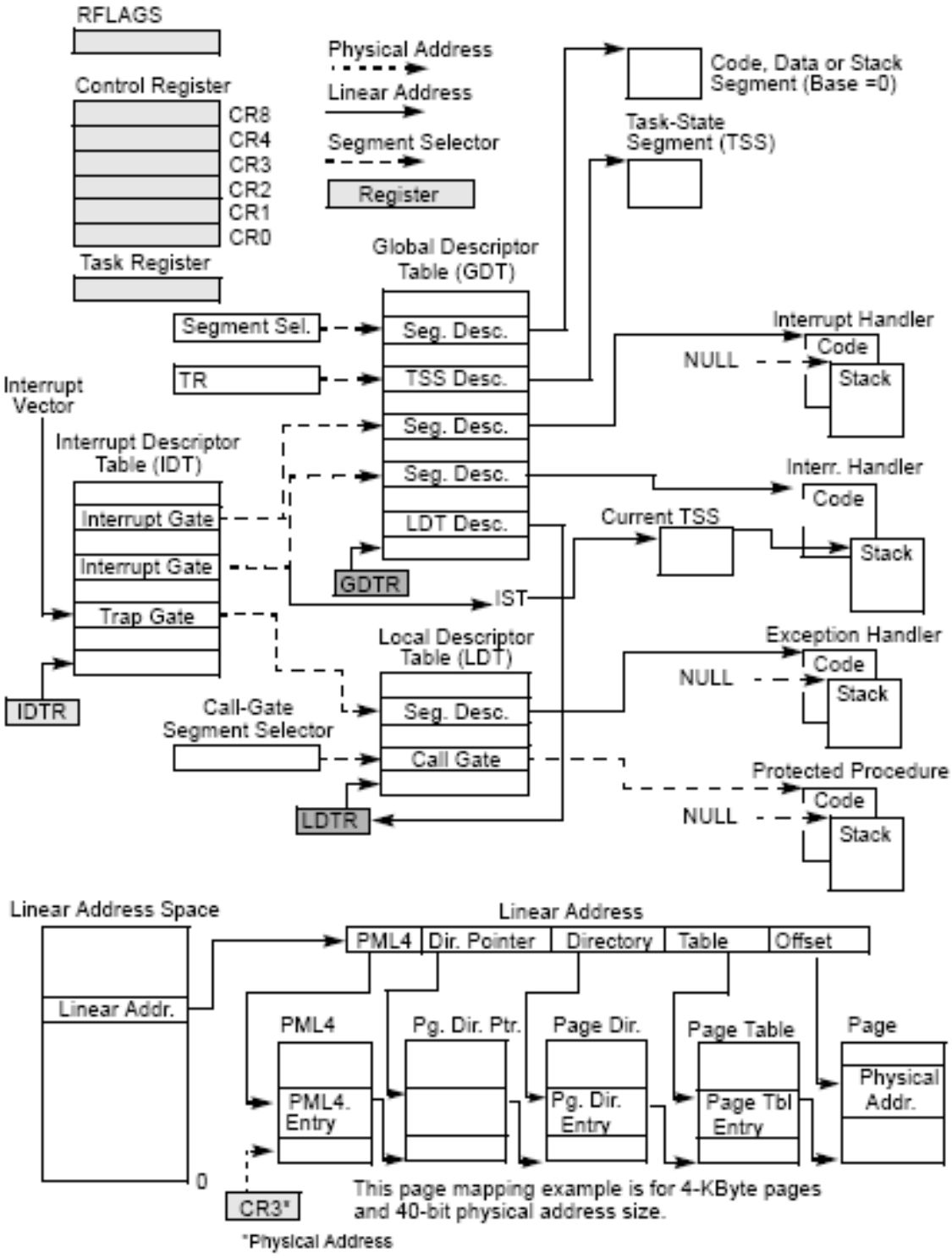

Figure 2: Intel IA-32e mode System Level Registers and Data Structures

Global and Local Descriptor Tables

When operating in protected mode, all memory accesses pass through either the global descriptor table (GDT) or an optional local descriptor table (LDT) as shown in Figure 2. These tables contain entries called segment descriptors. Segment descriptors provide the base address of segments well as access rights, type, and usage information.

Each segment descriptor has an associated segment selector. A segment selector provides the software that uses it with an index into the GDT or LDT (the offset of its associated segment descriptor), a global/local flag (determines whether the selector points to the GDT or the LDT), and access rights information. To access a byte in a segment, a segment selector and an offset must be supplied. The segment selector provides access to the segment descriptor for the segment (in the GDT or LDT). From the segment descriptor, the processor obtains the base address of the segment in the linear address space. The offset then provides the location of the byte relative to the base address. This mechanism can be used to access any valid code, data, or stack segment, provided the segment is accessible from the current privilege level (CPL) at which the processor is operating. The CPL is defined as the protection level of the currently executing code segment.

The linear address of the base of the GDT is contained in the GDT register (GDTR); the linear address of the LDT is contained in the LDT register (LDTR).

GDTR and LDTR registers are expanded to 64-bits wide in both IA-32e sub-modes (64-bit mode and compatibility mode). Global and local descriptor tables are expanded in 64-bit mode to support 64-bit base addresses, (16-byte LDT descriptors hold a 64-bit base address and various attributes). In compatibility mode, descriptors are not expanded.

Interrupt and Exception Handling

External interrupts, software interrupts and exceptions are handled through the interrupt descriptor table (IDT). The IDT stores a collection of gate descriptors that provide access to interrupt and exception handlers. Like the GDT, the IDT is not a segment. The linear address for the base of the IDT is contained in the IDT register (IDTR).

Gate descriptors in the IDT can be interrupt, trap, or task gate descriptors. To access an interrupt or exception handler, the processor first receives an interrupt vector (interrupt number) from internal hardware, an external interrupt controller, or from software by means of an INT, INTO, INT 3, or BOUND instruction. The interrupt vector provides an index into the IDT. If the selected gate descriptor is an interrupt gate or a trap gate, the associated handler procedure is accessed in a manner similar to calling a procedure through a call gate. If the descriptor is a task gate, the handler is accessed through a task switch.

Memory Management

System architecture supports either direct physical addressing of memory or virtual memory (through paging). When physical addressing is used, a linear address is treated as a physical address. When paging is used: all code, data, stack, and system segments (including the GDT and IDT) can be paged with only the most recently accessed pages being held in physical memory. The location of pages (sometimes called page frames) in physical memory is contained in two types of system data structures: page directories and page tables. Both structures reside in physical memory.

The base physical address of the page directory is contained in control register CR3. An entry in a page directory contains the physical address of the base of a page table, access rights and memory management information. An entry in a page table contains the physical address of a page frame, access rights and memory management information.

To use this paging mechanism, a linear address is broken into three parts. The parts provide separate offsets into the page directory, the page table, and the page frame. A system can have a single page directory or several. For example, each task can have its own page directory.

System Registers

To assist in initializing the processor and controlling system operations, the system architecture provides system flags in the EFLAGS register and several system registers:

- The system flags and IOPL field in the EFLAGS register control task and mode switching, interrupt handling, instruction tracing, and access rights.
- The control registers (CR0, CR2, CR3, and CR4) contain a variety of flags and data fields for controlling system-level operations. Other flags in these registers are used to indicate support for specific processor capabilities within the operating system or executive.
- The debug registers allow the setting of breakpoints for use in debugging programs and systems software.
- The GDTR, LDTR, and IDTR registers contain the linear addresses and sizes (limits) of their respective tables.
- The task register contains the linear address and size of the TSS for the current task.
- Model-specific registers: The model-specific registers (MSRs) are a group of registers available primarily to operating-system or executive procedures (that is, code running at privilege level 0). These registers control items such as the debug extensions, the performance-monitoring counters, the machine- check architecture, and the memory type ranges (MTRRs).

Most systems restrict access to system registers (other than the EFLAGS register) by application programs. Systems can be designed, however, where all programs and procedures run at the most privileged level (privilege level 0). In such a case, application programs would be allowed to modify the system registers.

Modes of Operation

The IA-32 supports three operating modes and one quasi-operating mode:

- **Protected mode** - This is the native operating mode of the processor. It provides a rich set of architectural features, flexibility, high performance and backward compatibility to existing software base.

- **Real-address mode -** This operating mode provides the programming environment of the Intel 8086 processor, with a few extensions (such as the ability to switch to protected or system management mode).
- **System management mode (SMM)** — SMM is a standard architectural feature in all IA-32 processors, beginning with the Intel386 SL processor. This mode provides an operating system or executive with a transparent mechanism for implementing power management and OEM differentiation features. SMM is entered through activation of an external system interrupt pin (SMI#), which generates a system management interrupt (SMI). In SMM, the processor switches to a separate address space while saving the context of the currently program or task. SMM-specific code may then be executed transparently. Upon returning from SMM, the processor is placed back into its state prior to the SMI.
- **Virtual-8086 mode** — In protected mode, the processor supports a quasioperating mode known as virtual-8086 mode. This mode allows the processor execute 8086 software in a protected, multitasking environment.

Intel 64 architecture supports all operating modes of IA-32 architecture and IA-32e modes:

- **IA-32e mode -** In IA-32e mode, the processor supports two sub-modes: compatibility mode and 64-bit mode. 64-bit mode provides 64-bit linear addressing and support for physical address space larger than 64 GBytes. Compatibility mode allows most legacy protected-mode applications to run unchanged.

Figure 3 shows how the processor moves between operating modes. The processor is placed in real-address mode following power-up or a reset. The PE flag in control register CR0 then controls whether the processor is operating in real address or protected mode. The VM flag in the EFLAGS register determines whether the processor is operating in protected mode or virtual-8086 mode. Transitions between protected mode and virtual-8086 mode are generally carried out as part of a task switch or a return from an interrupt or exception handler. The LMA bit (IA32_EFER.LMA.LMA[bit 10]) determines whether the processor is operating in IA-32e mode. When running in IA-32e mode, 64-bit or compatibility sub-mode operation is determined by CS.L bit of the code segment. The processor enters into IA-32e mode from protected mode by enabling paging and setting the LME bit (IA32_EFER.LME[bit 8]).

The processor switches to SMM whenever it receives an SMI while the processor is in real-address, protected, virtual-8086, or IA-32e modes. Upon execution of the RSM instruction, the processor always returns to the mode it was in when the SMI occurred.

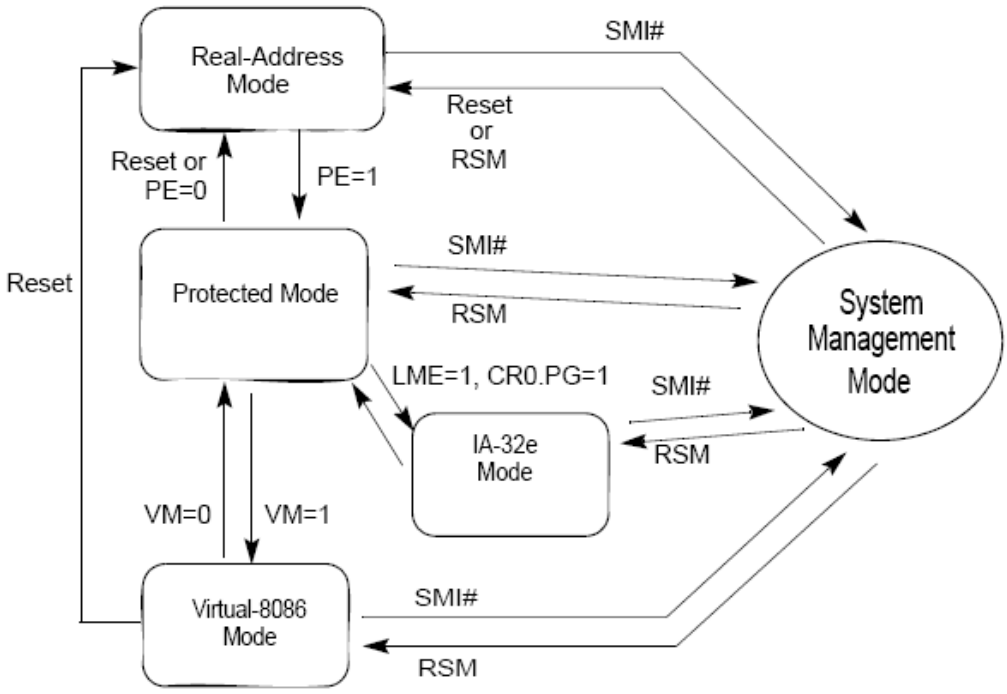

Figure 3: IA32e Modes of operation

Control Registers

Control registers (CR0, CR1, CR2, CR3, and CR4) determine operating mode of the processor and the characteristics of the currently executing task. These registers are 32 bits in all 32-bit modes and compatibility mode. In 64-bit mode, control registers are expanded to 64 bits. The MOV CRn instructions are used to manipulate the register bits. Operand-size prefixes for these instructions are ignored.

- **CR0** - Contains system control flags that control operating mode and states of the processor.
- **CR1** - Reserved.
- **CR2** - Contains the page-fault linear address (the linear address that caused a page fault).
- **CR3** - Contains the physical address of the base of the page directory and two flags (PCD and PWT). This register is also known as the page-directory base register (PDBR). Only the most-significant bits (less the lower 12 bits) of the base address are specified; the lower 12 bits of the address are assumed to be 0. The page directory must thus be aligned to a page (4-KByte) boundary. The PCD and PWT flags control caching of the page directory in the processor's internal data caches. When using the physical address extension, the CR3 register contains the base address of the page-directory-pointer table In IA-32e mode, the CR3 register contains the base address of the PML4 table.

- **CR4 -** Contains a group of flags that enable several architectural extensions, and indicate operating system or executive support for specific processor capabilities. The control registers can be read and loaded (or modified) using the moveto-or-from-control-registers forms of the MOV instruction. In protected mode, the MOV instructions allow the control registers to be read or loaded (at privilege level 0 only). This restriction means that application programs or operating system procedures (running at privilege levels 1, 2, or 3) are prevented from reading or loading the control registers.

- **CR8 -** Provides read and write access to the Task Priority Register (TPR). It specifies the priority threshold value that operating systems use to control the priority class of external interrupts allowed to interrupt the processor. This register is available only in 64-bit mode. However, interrupt filtering continues to apply in compatibility mode.

Of special interest to us is the VMXE bit in CR4

VMX-Enable Bit (bit 13 of CR4) - Enables VMX operation when set.

Chapter 3: Virtualization

Virtualization has been around since the late 1960s. For a good introduction on the principles behind Virtualization it is important to read the seminal work by Goldberg [1]. Taking a step back lets look at what virtualization technology really is.

Virtualization (or more specifically Platform Virtualization) is the creation of multiple virtual machines on a single physical platform allowing the ability to run multiple Operating Systems concurrently.
A Virtual Machine Monitor is a software layer that sits between the hardware and the Operating Systems (also known as Guest OS) and emulates an entire platform allowing the guest OSs to run concurrently in an isolated manner. There are two types of VMMs

1. Type 1 VMM (also known as Hypervisor architecture) runs directly on hardware and hosts guest Operating Systems. E.g. VMware Workstation and GSX Server
2. Type 2 VMM (Hosted architecture) that runs within an Operating System environment and hosts guest Operating Systems on top of this VMM. E.g. Xen and VMware ESX Server

On platforms that do not provide hardware hooks for supporting virtualization, there are two mechanisms for supporting virtualization

1. Para-virtualization which involves modifying the source code of the Guest OS so that it runs on a particular VMM
2. Binary Translation/Patching involves the VMM making modifications to the OS as it gets loaded to the memory or during run-time

For a good introduction on the current technology and future trends, refer [6].

Intel Virtualization Technology

Intel Virtualization Technology is a set of hardware features designed to facilitate Virtual Machine Monitors and allow them to run Guest OS without having to do either para-virtualization or Binary Translation.

VT-x is the technology used for Intel IA32/64 systems (EM64T systems) and VT-I is used for Intel Itanium systems. Here we talk only about VT-x and further references to VT refers to Intel VT-x

VT-x augments IA with two new modes of operations which are the VMX Root Mode and the VMX non-root Mode. The VMM runs in VMX Root Mode while the guest Operating Systems runs in VMX non-root mode. The transition from VMX root mode to non-root mode is known as VM Entry and the transition from VMX non-root mode to VMX root mode is known as a VM Exit.

Processor in VMX Root Mode allows the executing software all privileges while in VMX non-root mode, is restricted and modified to facilitate virtualization. These restrictions include the inability to execute certain privileged instructions and executing these instructions in VMX non-root mode results in a VM exit into the VMX root mode consequently giving control to the VMM.

Figure 4 illustrates the life cycle of a VMM and its guest software by illustrating the interactions between them.

- Software enters VMX operation through execution of the VMXON instruction.
- The VMM can then enter its guests into virtual machines (one at a time) using VM entries.

The VMM effects a VM entry using the VMX instructions VMLAUNCH and VMRESUME; it regains control using VM exits. VM exits transfer control to an entry point specified by the VMM. The VMM can take action appropriate to the cause of the VM exit and can then return to the virtual machine via a VM entry. Eventually, the VMM may decide to shut itself down and leave VMX operation. It does so by executing the VMXOFF instruction. For a complete list of instructions and events that cause VM Exit, refer to the Appendix A

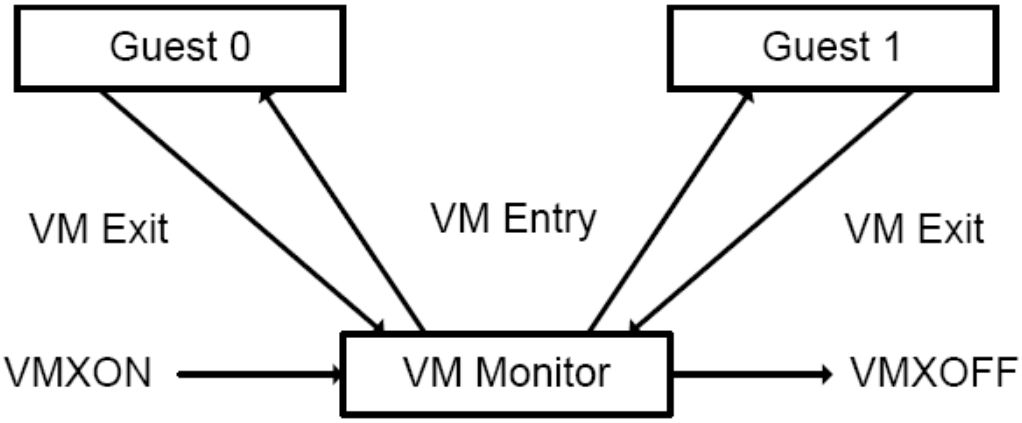

Figure 4: VM transitions

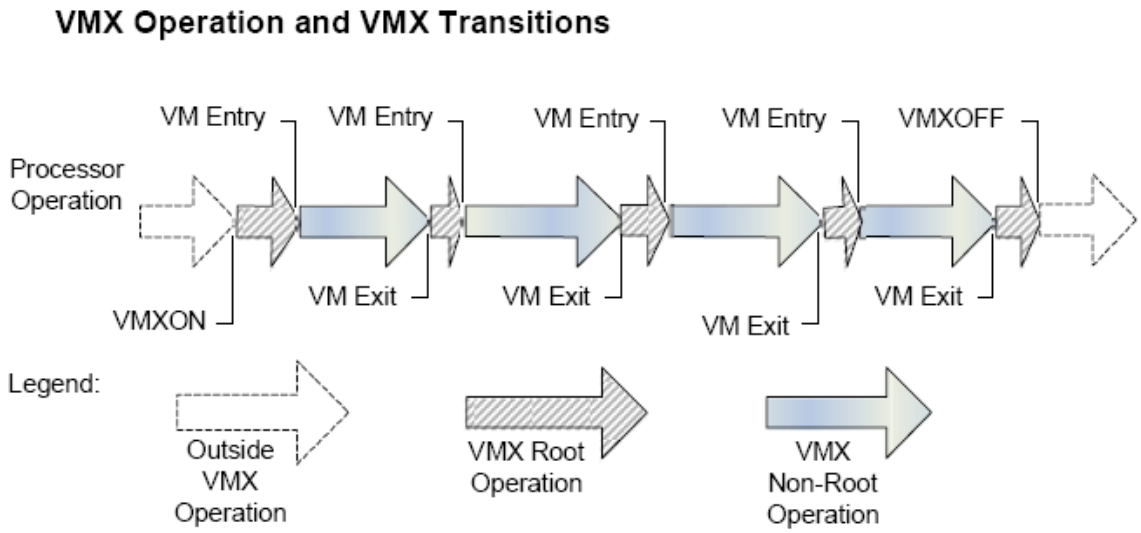

Figure 5: VMX Transitions

VMX non-root operation and VMX transitions are controlled by a data structure called a Virtual Machine Control structure (VMCS). Access to the VMCS is managed through a component of processor state called the VMCS pointer (one per logical processor). The value of the VMCS pointer is the 64-bit address of the VMCS. The VMCS pointer can be read and written using the instructions VMPTRST and VMPTRLD. The VMM configures a VMCS using other instructions: VMREAD, VMWRITE, and VMCLEAR.

A VMM could use a different VMCS for each virtual machine that it supports. For a virtual machine with multiple logical processors (virtual processors), the VMM could use a different VMCS for each virtual processor.

System software can determine whether a processor supports VMX operation using CPUID. If CPUID.1:ECX.VMX[bit 5] =1, then VMX operation is supported. Before system software can enter VMX operation, it must enable it by setting CR4.VMXE[bit 13] = 1. VMX operation can then be entered by executing the VMXON instruction. VMXON causes an invalid-opcode exception (#UD) if executed with CR4.VMXE = 0. Once in VMX operation, it is not possible to clear CR4.VMXE. VMXON is also controlled by the IA32_FEATURE_CONTROL MSR (MSR address 0000003AH). This MSR is cleared to zero when a logical processor is reset. The relevant bits of the MSR are described below:

• Bit 0 is the lock bit. If this bit is clear, VMXON causes a general-protection exception. If the lock bit is set, WRMSR to this MSR causes a general-protection exception. Once the lock bit is set, the MSR cannot be modified until a power-up reset condition.

• Bit 2 enables VMXON. If this bit is clear, VMXON causes a general-protection exception. Before executing VMXON, software should allocate a naturally aligned 4KB region of memory that a logical processor may use to support VMX operation.[1] This region is called the **VMXON region**. The physical address of the VMXON region (called the **VMXON pointer**) is provided in an operand to VMXON.

VMCS

The virtual-machine control data structure (VMCS) is defined for VMX operation. The VMCS manages transitions in and out of VMX non-root operation (VM entries and VM exits) as well as processor behavior in VMX non-root operation. A VMCS can be manipulated by the new instructions VMCLEAR, VMPTRLD, VMREAD, and VMWRITE. A VMM could use a different VMCS for each virtual machine that it supports. For a virtual machine with multiple logical processors (virtual processors), the VMM could use a different VMCS for each virtual processor.

A logical processor associates with each VMCS a 4KB region in memory called the **VMCS region**. Software references a VMCS by using the 64-bit physical address of this region; such an address is called a **VMCS pointer**. Every VMCS pointer must be 4KB-aligned (bits 11:0 must be zero). In addition, the pointer must not set bits beyond the processor's physical-address width.

A logical processor may maintain any number of active VMCSs, at most one of which is the current VMCS:

- Software makes a VMCS active by executing VMPTRLD with the address of the VMCS. The processor may optimize VMX operation by maintaining the state of an active VMCS in memory, on the processor, or both. Software should not make a VMCS active on more than one logical processor. Software makes a VMCS inactive by executing VMCLEAR with the address of the VMCS. A logical processor does not use an inactive VMCS or maintain its state on the processor. If VMXOFF is executed while a VMCS is active, the VMCS data in the corresponding VMCS region are undefined after execution of VMXOFF.

Software can avoid this problem by avoiding execution of VMXOFF while a VMCS is active.

- Software makes a VMCS current by executing VMPTRLD with the address of the VMCS; that address is loaded into the current-VMCS pointer. VMX instructions VMLAUNCH, VMPTRST, VMREAD, VMRESUME, and VMWRITE operate on the current VMCS. In particular, the VMPTRST instruction stores the current VMCS pointer into a specified memory location (it stores the value FFFFFFFF_FFFFFFFFH if there is no current VMCS). A VMCS remains current until either software executes VMPTRLD with the address of a different VMCS (which then becomes the current VMCS) or software executes VMCLEAR with the address of the current VMCS (after which there is no current VMCS).

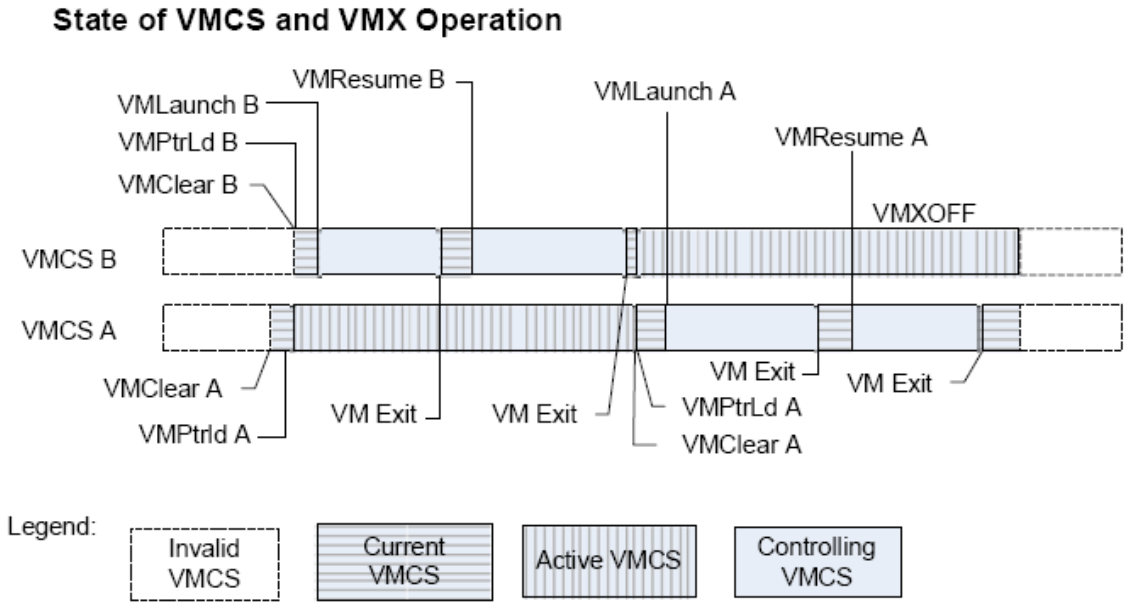

Figure 6: VMCS and VMX Transitions

Software can enter VMX non-root operation using either of the VM-entry instructions VMLAUNCH and VMRESUME. VMLAUNCH can be used only with a VMCS whose launch state is clear and VMRESUME can be used only with a VMCS whose the launch state is launched. VMLAUNCH should be used for the first VM entry after VMCLEAR; VMRESUME should be used for subsequent VM entries with the same VMCS.

The VMX_BASIC MSR (index 1152) consists of the following fields:

- Bits 31:0 contain the 32-bit VMCS revision identifier used by the processor.
- Bits 44:32 report the number of bytes that software should allocate for the VMXON region and any VMCS region. It is a value greater than 0 and at most 4096 (bit 44 is set if and only if bits 43:32 are all clear).
- Bits 53:50 report the memory type that the processor uses to access the VMCS for VMREAD and VMWRITE and to access the VMCS and data structures

referenced by pointers in the VMCS (for example, I/O bitmaps, TPR shadow, etc.) during VM entries, VM exits, and in VMX non-root operation. The first processors to support VMX operation use the write-back type. The values used are given in

Software should map all VMCS regions and referenced data structures with the indicated memory type.

- The values of bits 49:45 and bits 63:54 are reserved and are read as 0.

VMCS Entries:

The VMCS data are organized into six logical groups:

- **Guest-state area.** Processor state including control registers, Instruction Pointer (IP), Stack Pointer (SP), Flags etc are saved into the guest-state area on VM exits and loaded from there on VM entries.
- **Host-state area.** Processor state which is basically the state of the VMM, is loaded from the host-state area on VM exits.
- **VM-execution control fields.** These fields control processor behavior in VMX non-root operation. They determine in part the causes of VM exits. These include
    - **IO Bitmap Address:** IO bitmap A contains one bit for each IO port in the range 0000H to 7FFFH and IO bitmap B contains bits for ports in the range 8000H to FFFFH. Execution by the Guest OS of an IO instruction (io/out port) causes a VM exit if the bit corresponding to the port address in the IO bitmap is set to 1.
    - **CR3-**Target controls: Execution of MOV to CR3 in VMX non-root operation do not cause a VM Exit if the source operand matches one of these values
    - Pin based VM Execution Controls: Control whether external interrupts and NMI cause VM Exits
    - Processor based VM Execution Controls
- **VM-exit control fields.** These fields control VM exits.
    - **VM Exit Controls:** Control state of Logical Processor after VM Exit
    - **VM Exit Controls for MSRs:** Contain values for Model Specific Registers on VM Exit
- **VM-entry control fields.** These fields control VM entries.
- **VM-exit information fields.** These fields receive information on VM exits and describe the cause and the nature of VM exits. They are read-only.

The VM-execution control fields, the VM-exit control fields, and the VM-entry control fields are referred to collectively as VMX controls.

Certain limitations and features exist for software access to VMCS.

VMX Non-Root Operation

Execution in non-root mode is similar to execution in root mode except that certain instructions and events cause a VM Exit into non-root mode.

- Instructions: The instructions causing the VM exit do not execute and no processor state is updated by the instruction. Certain instructions cause VM Exit unconditionally while others cause conditional exits. The following instructions cause VM exits whenever they are executed in VMX non-root operation: CPUID, INVD, MOV from CR3, RDMSR, WRMSR, and the new instructions VMCALL, VMCLEAR, VMLAUNCH, VMPTRLD, VMPTRST, VMREAD, VMRESUME, VMWRITE, VMXOFF, and VMXON. Instructions causing conditional VM exits include INLVPG, IN, OUT etc.
- Events: The events causing VM Exits include Task switches, External Interrupts, INIT, Exceptions and SIPIs (Startup Interrupt Processor Interrupts)

VMX Root Operation

Software can enter VMX non-root operation using either of the VM-entry instructions VMLAUNCH and VMRESUME. VMLAUNCH can be used only with a VMCS whose launch state is clear and VMRESUME can be used only with a VMCS whose the launch state is launched. VMLAUNCH should be used for the first VM entry after VMCLEAR; VMRESUME should be used for subsequent VM entries with the same VMCS.

VM exits perform the following operations:

1. Information about the cause of the VM exit is recorded in the VM-exit information fields and the valid bit (bit 31) is cleared in the VM-entry interruption-information field.
2. Processor state is saved into the guest-state area.
3. MSRs may be saved into the VM-exit MSR-store area
4. The following may be performed in parallel and in any order
    a. Processor state is loaded based in part on the host-state area and some VM-exit controls.
    b. Address-range monitoring is cleared.
5. MSRs may be loaded from the VM-exit MSR-load area (Section 5.6).

VMX Instruction Set Reference

The virtual-machine extensions (VMX) includes five instructions that manage the virtual machine control structure (VMCS) and five instruction that manage VMX operation.

The behavior of the VMCS maintenance instructions are summarized below:

- VMPTRLD. This instruction takes a single 64-bit source operand that is in memory. It makes the referenced VMCS active and current, loading the current-VMCS pointer with this operand and establishes the current VMCS based on the contents of VMCS-data area in the referenced VMCS region. Because this makes

the referenced VMCS active, a logical processor may start maintaining on the processor some of the VMCS data for the VMCS.

- VMPTRST. This instruction takes a single 64-bit destination operand that is in memory. The current-VMCS pointer is stored into the destination operand.
- VMCLEAR. This instruction takes a single 64-bit operand that is in memory. The instruction sets the launch state of the VMCS referenced by the operand to "clear", renders that VMCS inactive, and ensures that data for the VMCS have been written to the VMCS data area in the referenced VMCS region. If the operand is the same as the current-VMCS pointer, that pointer is made invalid.
- VMREAD This instruction reads a component from the VMCS (the encoding of that field is given in a register operand) and stores it into a destination operand that may be a register or in memory.
- VMWRITE. This instruction writes a component to the VMCS (the encoding of that field is given in a register operand) from a source operand that may be a register or in memory.

The behaviors of the rest of the VMX instructions are summarized below:

- VMCALL. This instruction allows a guest in VMX non-root operation to call the VMM for service. A VM exit occurs, transferring control to the VMM.
- VMLAUNCH. This instruction launches a virtual machine managed by the VMCS. A VM entry occurs, transferring control to the VM.
- VMRESUME. This instruction resumes a virtual machine managed by the VMCS. A VM entry occurs, transferring control to the VM.
- VMXOFF. This instruction leaves VMX operation.
- VMXON. This instruction takes a single 64-bit source operand that is in memory. It causes a logical processor to enter VMX root operation and to use the memory referenced by the operand to support VMX operation.

For further details on the VT instructions refer to [9]

Chapter 4: Profiling

A profiler is a performance analysis tool that measures the behavior of a program as it runs, particularly the frequency and duration of function calls. The output is a stream of recorded events (a trace) or a statistical summary of the events observed (a profile). Profilers use a wide variety of techniques to collect data, including hardware interrupts, code instrumentation, operating system hooks, and performance counters.

Profile measurement data is linear to the program size, while the trace follows the actual path including loops and hence follows the runtime path. Program analysis tools are extremely important for understanding program behavior. Computer architects need such tools to evaluate how well programs will perform on new architectures. Software writers need tools to analyze their programs and identify critical pieces of code. Compiler writers often use such tools to find out how well their instruction scheduling or branch prediction algorithm is performing.

Profiler maybe of several types based on Output, Sampling, Instrumentation type etc.

Types based on Output

1. Flat profiler: Flat profiler's compute the average call times, from the calls, and do not breakdown the call times based on the callee or the context.
2. Call-Graph profiler: Call Graph profilers show the call times, and frequencies of the functions, and also the call-chains involved based on the callee. However context is not preserved.

Statistical profilers

Some profilers operate by sampling. A sampling profiler probes the target program's program counter at regular intervals using operating system interrupts. Sampling profiles are typically less accurate and specific, but allow the target program to run at near full speed.

Some profilers instrument the target program with additional instructions to collect the required information. Instrumenting the program can cause changes in the performance of the program, causing inaccurate results and heisenbugs. Instrumenting can potentially be very specific but slows down the target program as more specific information is collected.

The resulting data are not exact, but a statistical approximation. The actual amount of error is usually more than one sampling period. In fact, if a value is n times the sampling period, the expected error in it is the square-root of n sampling periods.

Instrumentation profilers

The technique where compilers that inserts output analysis data code at compile time into the program to be profiled are known as instrumentation profilers. Instrumentation profilers can be classified as

1. Manual: Done by the programmer, e.g. by adding instructions to explicitly calculate runtimes.
2. Compiler assisted: Example: "gcc -pg ..." for gprof, "quantify g++ ..." for Quantify

3. Binary translation: The tool adds instrumentation to a compiled binary. Example: ATOM
4. Runtime instrumentation: Directly before execution the code is instrumented. The program run is fully supervised and controlled by the tool. Examples: PIN, Valgrind
5. Runtime injection: More lightweight than runtime instrumentation. Code is modified at runtime to have jumps to helper functions. Example: DynInst
6. **Hypervisor: Data are collected by running the (usually) unmodified program on a hypervisor. Example: SIMMON. The LWVMM will fall under this category as the hypervisor acts as the profiler and the Guest OS that is being profiled runs on top of the hypervisor and is not modified**
7. Simulator: Data are collected by running under an Instruction Set Simulator. Example: SIMMON

Chapter 5: EFI - Extensible Firmware Interface

EFI refers to the specification that is meant to replace BIOS (Basic Input Output Software) in PCs. Tiano is the implementation of the EFI specification

EFI Reference Code
To facilitate development of EFI based drivers and applications, Intel has provided a development environment that includes an implementation of the EFI specifications. This implementation is available for free download at Intel's site at [21]

EFI for usage as the VMM
This build environment contains the ability to create a boot floppy or a boot USB which can boot on any regular PC and provide an EFI environment including the EFI shell. This allows for the development of EFI drivers and applications. EFI drivers and applications are executed in an IA Protected Flat Mode. The Protected Flat Mode is a model in which all of the memory is available for direct access by the software. In the Protected Flat mode Linear Address is the same as Logical Address and is the same as the Physical Address. The EFI environment provides a great environment for implementation of our Virtual Machine Monitor for the following reasons

a. EFI provides a pre-OS build environment.
b. EFI provides a pre-OS boot environment which is needed for the VMM to load before loading the OS.
c. EFI provides a Protected Flat Mode which can be used by the VMM to set itself up and to launch the Guest OS.
d. EFI provides a USB boot disk environment which enables us to plug in the EFI-VMM boot disk into a USB port and enable VMM operations instead of having to partition the hard disk and needing a place to stand for the VMM

The EFI specification defines a model for the interface between operating systems and platform firmware. The interface consists of data tables that contain platform-related information, plus boot and runtime service calls that are available to the operating system and its loader. Together, these provide a standard environment for booting an operating system and running pre-boot applications.

The EFI specification was primarily intended for the next generation of IA architecture-based computers and began in 1998. One of the goals of the EFI group was to enable the usage of EFI in non-IA architectures. With this goal in mind, in 2005 the Unified EFI Forum was formed to enable the industry to use this specification to implement the initial boot environment for other architectures. Currently, the UEFI Specification is becoming accepted throughout the industry to replace the EFI Specifications.

Why EFI
The "PC-AT" boot environment presents significant challenges to innovation within the industry. Each new platform capability or hardware innovation requires firmware developers to craft increasingly complex solutions, and often requires OS developers to

make changes to their boot code before customers can benefit from the innovation. This can be a time-consuming process requiring a significant investment of resources.

The primary goal of the EFI specification is to define an alternative boot environment that can alleviate some of these considerations. In this goal, the specification is similar to other existing boot specifications. The main properties of this specification and similar solutions can be summarized by these attributes:

- *Coherent, scalable platform environment*. The specification defines a complete solution for the firmware to completely describe platform features and surface platform capabilities to the OS during the boot process. The definitions are rich enough to cover the full range of contemporary Intel architecture-based system designs.
- *Abstraction of the OS from the firmware*. The specification defines interfaces to platform capabilities. Through the use of abstract interfaces, the specification allows the OS loader to be constructed with far less knowledge of the platform and firmware that underlie those interfaces.
- *Reasonable device abstraction free of legacy interfaces*. "PC-AT" BIOS interfaces require the OS loader to have specific knowledge of the workings of certain hardware devices. This specification provides OS loader developers with something different abstract interfaces that make it possible to build code that works on a range of underlying hardware devices without having explicit knowledge of the specifics for each device in the range.
- *Abstraction of Option ROMs from the firmware*. This specification defines interfaces to platform capabilities including standard bus types such as PCI, USB, and SCSI. The list of supported bus types may grow over time, so a mechanism to extend to future bus types is included.

EFI Architecture

Figure 7 shows the principal components of EFI and their relationship to platform hardware and OS software.

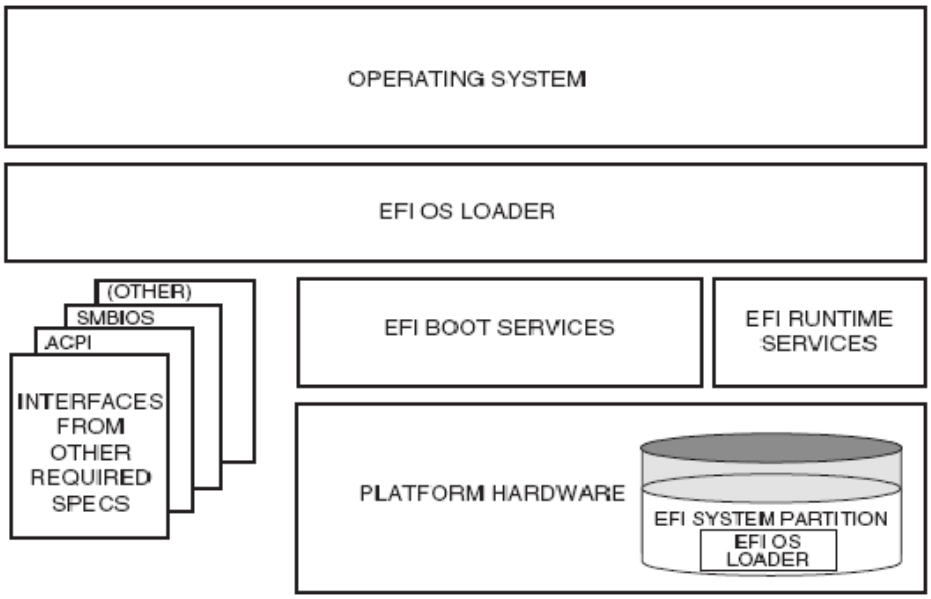

Figure 7: EFI components

The platform firmware is able to retrieve the OS loader image from the EFI System Partition. The specification provides for a variety of mass storage device types including disk, CD-ROM and DVD as well as remote boot via a network. Once started, the OS loader continues to boot the complete operating system. To do so, it may use the EFI boot services and interfaces defined by this or other required specifications to initialize the various platform components and the OS software that manages them. EFI services may be booted into two, EFI boot time services and EFI runtime services. EFI boot services are available only at boot time when the OS Loader is loaded. EFI runtime services are available even when the OS is up and running.

EFI Boot Flow

EFI allows the extension of platform firmware by loading EFI driver and EFI application images once the initial platform init is done as shown in Figure 8. When EFI drivers and EFI applications are loaded they have access to all EFI defined runtime and boot services. EFI allows the consolidation of boot menus from the OS loader and platform firmware into a single platform firmware menu. These platform firmware menus will allow the selection of any EFI OS loader from any partition on any boot medium that is supported by EFI boot services. An EFI OS loader can support multiple options that can appear on the user interface. It is also possible to include legacy boot options, such as booting from the A: or C: drive in the platform firmware boot menus.

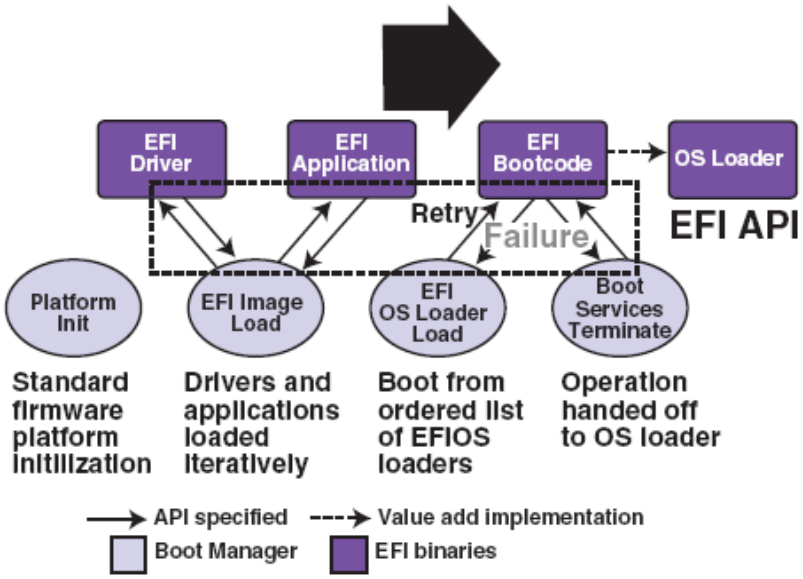

Figure 8: EFI Boot Flow

Chapter 6: LWVMM – Light-Weight VMM

The previous chapters provided the essential ingredients needed for us to begin our implementation of the Lightweight VMM. As mentioned in the introduction, the plan is to use the virtualization feature in Intel VT enabled systems to implement a light-weight and low overhead VMM that can run fully virtualized COTS Guest Operating Systems

The lightweight VMM can be passed various parameters to collect statistics of the behavior of the Guest OS using Virtual Machine Enters and Exits (VMEnter/VMExit) which are triggered by instruction executions and by events in the Guest OS and the Workloads. This helps to profile the different aspects of the Guest OS and the Workloads on different platforms.

The VMM can collect and provide information that can be used to identify Guest OS hot spots, bottlenecks and inefficient implementations. This information would allow the System Architects to understand which aspects of the System Architecture would need to be looked at to improve performance, would help OS developer to understand aspects of the OS that need to be modified for better performance and would help Workload designers to understand what changes are required for the workload to run them faster on the system.

The advantage of this is that OS profiling can be done by an independent software stack that does not depend on the OS for execution. Thus any Operating System and any Workload can be profiled using this light-weight VMM without having to port the profiling stack to each OS environment. Thus a direct comparison can be done between different OS on similar workloads to understand the relative advantages and disadvantages of each OS. For example, a web transaction on a Linux server can be compared to the same web transaction on a Windows server and the corresponding number of IO transactions needed can be measured to see where the bottlenecks in each system are. Similarly, a workload can be run and analyzed to understand which are the most commonly used instructions and then the Processor architect can identify if some of the commonly used instructions can be optimized or the Software Architect can decide to improve certain portions of the algorithm used. Thus profiling can be done without modifying any portions of either the OS or the Workload. This is useful in cases where only the binary is available like in the case of Windows OS.

The VMM profiling approach also avoids the overheads of running the profiling tool on top of the OS and in turn coloring the OS characteristics.

LWVMM Architecture:

As shown in Figure 1, the LWVMM sits between the OS and the VT supported hardware. There are multiple ways of loading the LWVMM in a system.

1. Load the LWVMM before the OS gets control:
   Here, the LWVMM may be loaded once the BIOS has finished initializing the hardware and is ready to load the OS loader. The LWVMM can be set up to load

itself, enable VT, set itself up and then load the OS loader using the VM Launch instruction after setting up the VM Control Structure for the OS

2. Load the LWVMM during OS run:
   Here the LWVMM would get loaded manually as a driver once the OS has finished booting up, it obtains memory from the OS to run and to set up the VM structures and then enables VT. It then sets up the VM structures and then while handing over control back to the OS, does so by doing a VM Launch instead of a return.
3. Load the LWVMM during the OS load:
   This would be similar to the above. Here the LWVMM would get loaded as a driver during the OS driver load, it obtains memory from the OS to run and to set up the VM structures and then enables VT. It then sets up the VM structures and then while handing over control back to the OS, does so by doing a VM Launch instead of a return. The advantage of (3) over (2) is that this can profile the OS during the load stage too.

Both (2) and (3) need the OS driver wrappers since they execute in the context of the OS, and thus are OS dependant. This would defeat one of the main advantages of having OS independent software for profiling the OS. Once set up, (2) and (3) will not color the OS execution as now they are executing outside of the OS execution mode. They however need to execute at Ring 0 privilege as virtualization hooks are available for execution only at CPL = 0 (Ring 0).

(2) and (3) and potentially easier to implement due to two main reasons.

1. Hardware has been set up with the required environment for enabling Virtualization. This includes Paging and Segmentation. For (1), the VMM has the task of enabling segmentation and paging and also that of loading the OS loader before launching it.
2. Doing a VM launch to launch back the OS is easy as parameters in the VM Control Structure which consist of the registers and the system state can be filled up from what the state was when the VMM got control from the OS. This state is what was essentially stored when the VMM got control from the OS at the VMM startup. However for (1), the system state has to be set up such that it is conducive for loading the OS loader

Examples of (2) and (3) include Rutkowska's Blue Pill and Shawn Embleton's VMM. See References for more details.

However, it has been decided to implement (1) due to the above mentioned limitation of OS dependency.

Thus the LWVMM gets control immediately after the BIOS initialization instead of the OS Loader. The LWVMM checks and enables VT, sets up the VT tables, loads the Virtual Machine Control Structure (VMCS) for the OS and then launches the Virtual Machine which is the OS.

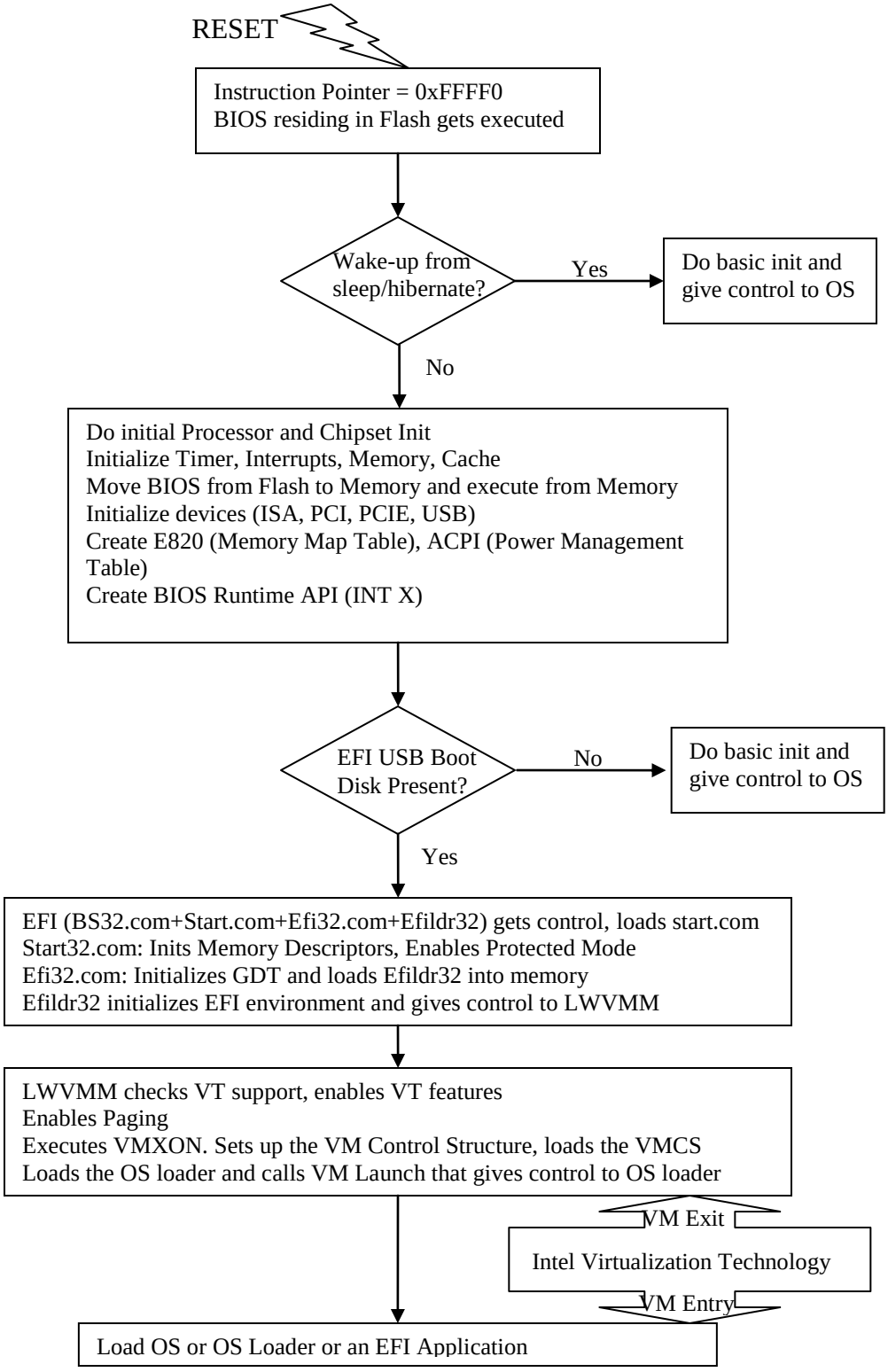

Figure 9: The BIOS-EFI-VMM-OS flow

The LWVMM implementation details and enumerated in Chapter 8.

Chapter 7: Hardware Setup

The Hardware Platform setup consists of an Intel Core 2 Duo system with a 965 chipset Motherboard. A USB Flash disk is used to create a bootable EFI based VMM.

VT Support:

The BIOS has to be set up to enable Intel VT support. Enter BIOS Setup using the F2 key. Go to Security page, and enable VT Technology

Processor Setup

In the Main Page, Disable Core Multiplexing Technology. This enables only one processor of the Core 2 Duo processor. This is done to essentially simplify the process of setting up the VMM. With two or more Processor Cores, multiple VMCS have to be setup and enabled. Since the purpose of this project is to show a conceptual VMM, it is best left to further work to enable the VMM on multicore systems.

USB Boot:

The BIOS setup has to be changed to enable USB Boot devices and to make the USB the first boot device so that the USB device can get control at BIOS exit and boot up the LWVMM. To do this, go to Advanced→USB Configuration and verify that USB Ports are enabled, USB 2.0 is enabled and USB Legacy is also enabled. Next go to Boot Menu and change the

Boot to Removable Devices → Enable

USB Boot → Enable

Boot USB Device First → Enable

USB Mass Storage Emulation Type → All Fixed Disc

Hard Drive Order to set the USB device as the first boot device

Chapter 8: Implementation

The Lightweight VMM is created as a bootable image on a USB disk device. This way, the System Analyzer only needs to plug in the USB boot disk into any regular system to be able to profile that system. Thus it would require no changes to the BIOS (re-flashing of the BIOS), no changes to the disk image or disk partitions to host the VMM, no changes to the OS and neither the need to install any profiling agent in the OS. To stop profiling, all that would be needed would be to boot without the USB device.

The PC boot flow

On any PC system the BIOS is the first component that gets control once the reset button has been pressed or the system has been powered up. The reset vector is the first address that is executed when the system is powered on. On a PC (x86 systems) that address is 0xFFFF-FFF0. On all PC architectures, this address is mapped to the System Flash. The system Flash contains the BIOS. The BIOS is responsible for initializing and bringing up the system. This includes initializing the processor, chipset and the remaining components of the system including the hard disks, USB, CD ROM, network etc. The BIOS also provides a common interface (exposed as INT instructions) that the OS loader can use to boot the rest of the OS up. Thus the BIOS provides a common interface on all different kinds of PCs so that the OS does not need any change on different PC platforms and different PC architectures.

Once the BIOS has finished initialization and has set up the common interfaces, the BIOS loads the Boot Manager that depending on the user input identifies where the OS resides and then does an INT 19 to load the first 64KB of the OS, which usually is the OS loader up. The OS loader with the help of other BIOS interfaces like INT 13 for disk services, INT10 for video etc. loads the rest of the OS up from the persistent storage like hard disks and CD ROMs or from the Network.

The EFI implementation sits between the BIOS and the OS and when an INT 19 is done, gets loaded and loads up the EFI drivers, applications and the EFI shell. The EFI environment also has a Boot Manager that the user can use to load the OS. Operating Systems that are EFI compatible are known as EFI aware OS while those that were written with traditional BIOS interfaces are known in EFI jargon as Legacy OS. EFI compatible OS are loaded in a different manner from Legacy OS. EFI compatible OS are loaded by loading an EFI OS loader which basically is an EFI application. The EFI OS loader can use the EFI boot time interfaces till it is in a position to load up the rest of the OS up. Once it no longer needs the EFI Boot Services, it calls a function called ExitEFIBootServices ( ) which tears down the EFI boot environment. After this, only the EFI Runtime Services are available.
EFI loads Legacy OS as traditional BIOS do by using INT19. For that purpose, EFI has to bring back the legacy BIOS environment that existed before which includes the INT services, the ACPI, SMBIOS, MPS tables and the Real Mode state of the processor.

As was mentioned before, EFI provides a good build and execution environment for our VMM development and implementation. It also provides a build environment for creating a USB boot disk. Below is a description of the various components that constitute an EFI boot disk

EFI Image Components

The bootable EFI firmware consists of two components:

a. **BootSect.com**

b. **Efildr**

BootSect.com
This component is the boot loader code that resides on the first sector of the EFI USB disk and is copied to location **0x7c00** during the boot process. Its main function is to locate the **Efildr** file in the FAT file system on the USB Boot Disk and start that image.

Efildr
This binary contains all the executable code required to initialize and start the EFI firmware. It is copied to the FAT file system on the USB using the **nmake usb32** command. It is, therefore, a normal file on the USB's file system, but internally contains three separate binaries/executables:

a. **start.com**

b. **efi32.com**

c. **Efildr32**

start.com:

This is a small executable that performs the following operations:

a. Reads the rest of the file system into memory.

b. Enables Gate A20, moves into protected mode.

c. Retrieves memory map descriptors from the BIOS (INT 15, Function E820h), and stores these descriptors in a temporary buffer.

d. Issues a far jump to **cs:21000**.

efi32.com
This executable is loaded at location **cs:21000** and performs the following operations:

a. Initializes the GDTs.

b. Retrieves the header information of the **Efildr** image.

c. Using the header information obtained in step b above, identifies the base address of the EFI loader entry point, **EfiLoader**.

d. Places the memory descriptors on the stack and issues a far jump to **EfiLoader**.

Efildr32
Contains the EFI loader. The VMM is implemented here

EFI Booting Sequence

Refer the diagram Figure 10. The BIOS identifies the EFI USB disk as bootable, loads boot sector code, B**ootSect.com**, into memory, and transfers control to this code.

BootSect.com searches the FAT file system on the USB disk to locate the **Efildr** file, copies this file into memory at 2000:0000 and transfers control to this latter location.

The usual components of **Efildr** shall then execute in the following order:

a. **Start.com** – Initialization of memory descriptors, protected mode operation.

b. **Efi32.com** - Initialization of GDTs and IVTs, identification of the entry point for the EFI loader. Jump to EFI loader and passes the memory descriptors in the input parameter to the EFI loader entry point, **EfiLoader**, in **Efildr.efi**.

c. **EfilLoader** – Copies memory descriptors from the input parameter and initializes the EFI environment. We begin the VMM implementation in this file

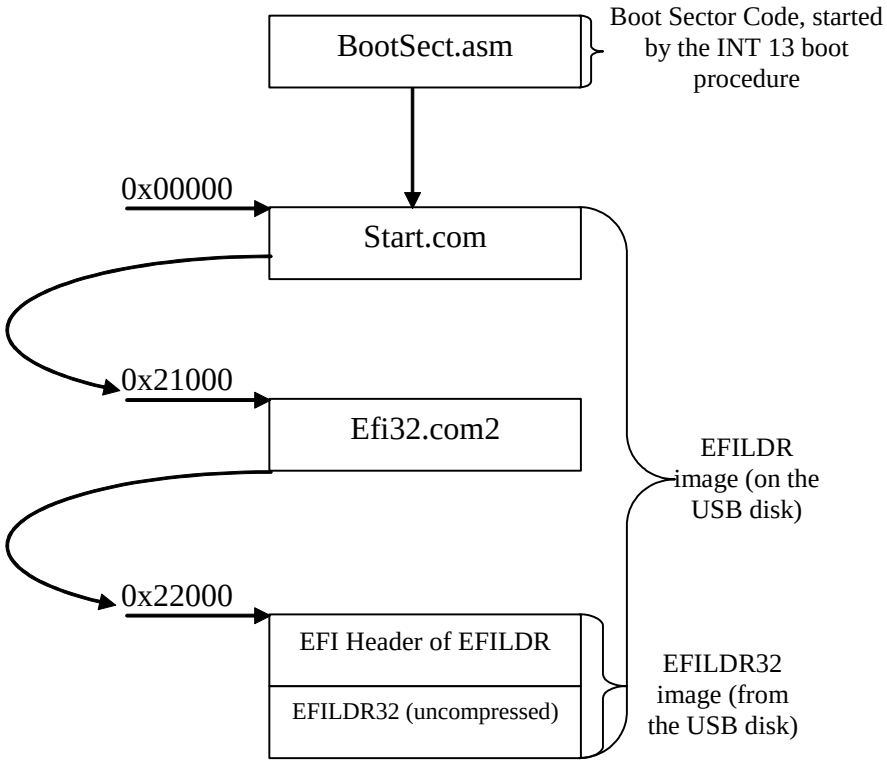

Figure 10: EFI Initial Boot Sequence

VMM Implementation

Once EFI has finished initialization of the system, the VMM can take over and set up the virtualization support in the system and boot an OS

The following are the conditions to be ensured before the system can got o VT Mode

1. VMMs need to ensure that the processor is running in protected mode with paging before entering VMX operation. The following list describes the minimal steps required to enter VMX root operation with a VMM running at CPL = 0.
2. Check VMX support in processor using CPUID.
3. Determine the VMX capabilities supported by the processor through the VMX capability MSRs.
4. Create a VMXON region in non-pageable memory of a size specified by IA32_VMX_BASIC MSR and aligned to a 4-KByte boundary. Software should read the capability MSRs to determine width of the physical addresses that may be used for the VMXON region and ensure the entire VMXON region can be addressed by addresses with that width. Also, software must ensure that the VMXON region is hosted in cache-coherent memory.
5. Initialize the version identifier in the VMXON region (the first 32 bits) with the VMCS revision identifier reported by capability MSRs.
6. Ensure the current processor operating mode meets the required CR0 fixed bits (CR0.PE = 1, CR0.PG = 1). Other required CR0 fixed bits can be detected through the IA32_VMX_CR0_FIXED0 and IA32_VMX_CR0_FIXED1 MSRs.
7. Enable VMX operation by setting CR4.VMXE = 1. Ensure the resultant CR4 value supports all the CR4 fixed bits reported in the IA32_VMX_CR4_FIXED0 and IA32_VMX_CR4_FIXED1 MSRs.
8. Ensure that the IA32_FEATURE_CONTROL MSR (MSR index 3AH) has been properly programmed and that its lock bit is set (Bit 0 = 1). This MSR is generally configured by the BIOS using WRMSR.
9. Execute VMXON with the physical address of the VMXON region as the operand.
10. Check successful execution of VMXON by checking if RFLAGS.CF = 0.
11. Upon successful execution of the steps above, the processor is in VMX root operation.

Enumerated below are the steps for setting up the VMM.

1. Identify if the processor supports VT
   This can be done by executing the CPUID instruction by setting EAX = 1 and

after the execution of the CPUID instruction checking bit 20. If bit 20 is enabled, Virtualization is supported. Else exit with an error code.

2. Enable Segmentation
   This requires setting up of the Global Descriptor Tables and then setting the GDTR (GDT Register) with the address of the beginning of the GDT. The GDT requires a minimum of three entries. The first entry has to be null as per Intel Specifications. The second entry is filled with the CS field. Since the system is going to operate in Flat Mode, the Segment Base is set as zero and the Segment limit is set as 0xFFFFFFFF (4 GB). Similarly for the third entry, which is used for all other segments including the Data Segment and the Extended Segments (DS, ES, FS, SS), the values are set up as Base Address = 0 and the Limit = 0xFFFFFFFF. EFI already sets up Protected Flat Mode and so all the VMM has to do is to check that Protected Mode is enabled by checking bit 0 of Control Register 0 (CR0)

3. Enable Paging and set up the Page Tables
   Since VT only operates when Paging is enabled (possibly because page fault traps are the only way to monitor the memory used by the OS), the VMM has to set up paging as neither the BIOS nor EFI enable Paging.
   Paging is controlled by the PG flag in control register CR0. When this flag is clear, the paging mechanism is turned off; when it is set, paging is enabled. However before enabling paging in the system, the Page directories and Page Tables have to be set up. First add a memory descriptor to set up the page directories and page tables. In this project, the Page Directories have been set up at 256MB and the Page Tables have been set up at 260MB. Systems with 32 bits addressability can access 2^32 bytes. This translates to 4 GB of memory. This is possible in the following way. Each Page Table maps a 4 KB of memory. Each Page Directory maps 1K of Page Tables. Hence 1024 Page Directories map, 1024*1024 Page Tables. Since each Page table maps 4KB of memory, the total memory accessible becomes (1024 PD * 1024 PT * 4K pages = 4 GB). PAE is a mechanism of extending this to 2^36 (64 GB) of memory and IA32E extends this to be able to theoretically access up to 2^64 bytes of memory. Since the system used for enabling the VMM in this project has 1 GB of memory it does not require either PAE or IA32E enabled and would need only 256 Page Directories for accessing the 1 GB of memory (256 PD * 1024 PT * 4K = 1 GB). The Page Directories and Page Tables are set up linearly such that the address before enabling and after enabling paging are the same. This way, there is no need of relocating the VMM code to a different address so that it is accessible after paging is enabled. Once the Page Directories and Page Tables are set up, CR3 (also known as PDBR – Page Directory Base Register) has to be filled up with the address of the start of the Page Directories (0x10000000). Next enable paging by setting bit 31 (PG) of the CR0 register. A jmp instruction is executed immediately after paging is enabled to flush the pre-fetch queue and the EIP is set to the new value again using a jmp routine.

4. Enable VT on the system
   Now that the two preconditions for enabling VT (Segmentation and Paging) have been minimally set up, we can go ahead and enable VT. This is done by enabling the VMXE bit in CR4 (Control Register 4).
5. Next the VMM features are checked by reading the VMX_BASIC_MSR Model Specific Register (MSR address = 0x480).
6. VMXON
   VMXON instruction is used to get into VMM Mode. Before executing VMXON, software allocates a region of memory (called the VMXON region) that the logical processor may use to support VMX operation. The physical address of this region (the VMXON pointer) is provided in an operand to VMXON. The VMXON pointer must be 4KB-aligned (bits 11:0 must be zero); in addition, the pointer must not set any bits beyond the processor's physical-address width. For the LWVMM, we set this to 0x0A000000 (160 MB). The size of the VMXON region and the VMX revision ID are obtained from the above mentioned VMX_BASIC_MSR.
7. Once VMXON is successfully executed, the system is now in VMX Root Mode, which means that the code now executing is the VMM.
8. The next step would be to set up the VMCS structure for the OS to be loaded. Initialize the version identifier in the VMCS (first 32 bits) with the VMCS revision identifier reported by the VMX capability MSR IA32_VMX_BASIC. Next execute the VMCLEAR instruction by supplying the guest-VMCS address. This will initialize the new VMCS region in memory and set the launch state of the VMCS to "clear". This action also invalidates the working VMCS pointer register to FFFFFFFF_FFFFFFFFH. Software should verify successful execution of VMCLEAR by checking if RFLAGS.CF = 0 and RFLAGS.ZF = 0.
9. Next load this VMCS to be the current VMCS. Execute the VMPTRLD instruction by supplying the guest-VMCS address. This initializes the working-VMCS pointer with the new VMCS region's physical address.
10. Issue a sequence of VMWRITEs to initialize various host-state area fields in the working VMCS. The initialization sets up the context and entry points to the VMM upon subsequent VM exits from the guest. Host-state fields include control registers (CR0, CR3 and CR4), selector fields for the segment registers (CS, SS, DS, ES, FS, GS and TR), and base-address fields (for FS, GS, TR, GDTR and IDTR; RSP, RIP and the MSRs that control fast system calls).
11. Use VMWRITEs to set up the various VM-exit control fields, VM-entry control fields, and VM-execution control fields in the VMCS. Care should be taken to make sure the settings of individual fields match the allowed 0 and 1 setting for the respective controls as reported by the VMX capability MSRs. Any settings inconsistent with the settings reported by the capability MSRs will cause VM entries to fail.

12. Use VMWRITE to initialize various guest-state area fields in the working VMCS. This sets up the context and entry-point for guest execution upon VM entry. This has to be the value expected by the OS loader when it initially gets control from the BIOS in the normal scenario. The various conditions are that Segmentation and Paging are disabled, and the system is in Real Addressing Mode. The Control registers are initialized accordingly.
13. Clear the VMX Abort Error Code (address = VMCS Region + 4) prior to VMLAUNCH
14. Execute the VMLAUNCH instruction to give control back to the OS.
15. This boot straps the Guest OS and the Guest OS gets control. If the VMM wants control back, it needs to enable the bit maps accordingly

Chapter 9: Project Summary

The hardware was set up. The EFI development environment including the build and makefiles were created. A USB boot disk was created that boots EFI.
The VMM was completely implemented including enabling VT support, setting up the required VMCS and VMXON regions and then launching the OS/Application using VMLAUNCH.

The profiling was done on an EFI application that was doing a number of hardware accesses.

Chapter 10: Conclusions and Recommendations

Conclusions:
Though it was a bit difficult to implement the VMM, having to enable Segmentation and Paging, in the end, compared to other methods of profiling mentioned in Chapter 4, this mechanism has proven to be much simpler in multiple ways.

1. Enabling various profiling parameters are much easier as they involve only enabling a bit to invoke a VM Exit. This is in contrast to having to put in profiling hooks in possibly multiple libraries and kernel functions and having the possibility of having missed certain points.
2. The profiler can be implemented without modifications to the OS. Thus it is easier to implement and the OS measurements do not get colored either.
3. The profiler is OS independent and can be used with different OS and for different workloads
4. Since VT is going to be ubiquitous in Intel and AMD based platforms, using VT enabled systems would become easy to find
5. Fine-grained profiling is possible which would not be possible in the case of other methods of profiling. This method is especially useful in profiling instruction set usages. For example, if a processor architect has an option of optimizing between two different instructions, running a few OS and workloads on the LWVMM Profiler would definitely point to which of the instructions need optimization based on the number of times it is used.

Issues with LWVMM:

1. Since VT technology is going to be ubiquitous in most platforms, VMMs and OS that take advantage of VT technology would also follow and soon become ubiquitous themselves. Since VT does not support multiple layers of virtualization, it is today not possible to have multiple VMMs in the same system. Hence if the OS already has a VMM inbuilt, then it would not be possible to run the LWVMM underneath it. One mechanism of overcoming this restriction is to virtualize the VMM itself. In this mechanism, the LWVMM would enable VT and run as the VMM. Any other VMM which tries to run will get virtualized and will get a set of virtualized registers. This however is complicated and would mean implementing a full set of VMM features.
2. Physical proximity required: The VMM is implemented as a USB boot device and thus would require the user to plug in the USB device into the USB port and he would not be remotely be able to turn on or turn off the VMM. This remote turn-on feature would be extremely useful in servers which may not be located in physical proximity of the user
3. Currently no mechanism has been provided for recording the different events and for storing the statistics. One potential area where they could be gathered would be in the VMM memory. However, that does not provide any mechanism for the user to download and decipher them. The other mechanism would be to store it in the USB key from where the VMM was booted. However that would mean that the VMM has to sequester off the USB port on which the USB key is plugged in.

Other mechanisms include sending over the network to the user and the like, but all of these mechanisms would mean additional effort to implement

Usages of the LWVMM:

The LWVMM can also be used as a framework for enabling other functionalities like implementing RAS (Reliability, Accessibility, and Serviceability) and system management features like memory and IO sparing. It can act as a virus scanner (network packet analysis etc.) to protect the system and also for implementing OS checkers (safe trusted place to stand for OS measuring agents)

Root Kit attacks:

On the harmful side this LWVMM can be used to launch root-kit attacks against users and their systems. For example, plugging a USB key with the LWVMM in an adversary's system could potentially be used for Root Kit attacks which could modify the OS such as to reside permanently on the adversary's system and on subsequent boots load the LWVMM without needing to boot from the USB disk. This way the attacker could potentially get control of the adversary's computer. The attacker could then be able to retrieve secret keys from registers, instructions, scans of memory etc. The LWVMM could also potentially change the OS flow and thus can be used for bypassing password entries, checking for licenses etc. However the threat of this being used for such malicious intent is discarded because there already exist a number of such root kit programs that can find vulnerabilities in Ring 0 code (OS kernel/driver) and load itself as a VMM and sit underneath the OS. These are easier to be used since they can be launched remotely and do not need physical access to stick in a USB stick as is needed in the LWVMM case. Rutkowska's Blue Pill is a good and well known example of the same and was done on an AMD based system.

Chapter 11: Directions for Future Work

Once the LWVMM is completed, it can further boot up a COTS OS like Linux or Windows. On successful load of the OS, the OS that now boots on top of the LWVMM can be profiled.

To do so, certain bits have to be enabled corresponding to instructions and events that would cause VM Exits. The LWVMM can then be used to gather statistics of how many of those events occurred during the OS boot or run. This can be analyzed to find out optimizations in the System and in the OS.

Once the project is done, the plan is to make the LWVMM user friendly and to provide invocation options and for storing the events in a file in a format like CSV (Comma Separated Values) so that it can be used later to graph the run.

One plan is also to generalize the LWVMM so that different groups can easily use the LWVMM to implement features that are important to them like System Management, OS Characterization Agents etc.

It is planned to write a paper titled "LWVMM – A lightweight VMM for profiling purposes" and provide more results of profiling Operating Systems and Workloads with the LWVMM

The LWVMM would also be open-source licensed since it is based on the EFI Reference Code which is BSD licensed.

References

Literature References <Convert to Alphabetical Listing>

1. R.P. Goldberg, "Survey of Virtual Machine Research," Computer, June 1974, pp. 34-45.
2. "Xen and the Art of Virtualization", Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt and Andrew Warfield, Proceedings of the ACM Symposium on Operating Systems Principles (SOSP), October 2003
3. Running multiple operating systems concurrently on an IA32 PC using virtualization techniques, Kevin Lawton, 1999 (technical article, unpublished)
4. Introducing VMware Virtual Platform, Technical White Paper, VMware Inc, February 1999
5. Architecture of Virtual Machines, Robert P. Goldberg, Proc. Workshop on Virtual Computer Systems, Cambridge, MA, 1973, pp 74-112.
6. Virtual Machine Monitors: Current Technology and Future Trends
   Mendel Rosenblum VMware Inc. and Tal Garfinkel Stanford University
   IEEE Computer May 2005
7. An Overview of Virtual Machine Architectures, James E. Smith, White Paper, University of Wisconsin, October 2001.
8. T. C. Bressoud , F. B. Schneider, Hypervisor-based fault tolerance, Proceedings of the fifteenth ACM symposium on Operating systems principles, p.1-11, December 03-06, 1995, Copper Mountain, Colorado, United States
9. Shawn Embleton – Virtual Machine Monitor for Windows. **www.rootkit.com**
10. Rutkowska - Blue Pill. **www.rootkit.com**
11. Intel Corp., "Intel Virtualization Technology Specification for the IA-32 Architecture" C97063-002 April 2005
12. Intel Virtualization Technology - Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C. M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kagi, Felix H. Leung, Larry Smith, May 2005 Computer, Volume 38 Issue 5
13. **http://www.intel.com/technology/virtualization/index.htm**
14. **http://www.intel.com/technology/efi**
15. **http://www.intel.com/technology/itj/2006/v10i3/index.htm**
16. Intel® 64 and IA-32 Architectures Software Developer's Manual
    Volume 1: Basic Architecture
    Volume 2A: Instruction Set Reference, A-M
    Volume 2B: Instruction Set Reference, N-Z
    Volume 3A: System Programming Guide
    Volume 3B: System Programming Guide
17. Wikipedia.org

Appendix A

Conditions causing VM Exits

It is important to identify what conditions in the Guest OS can cause VM Exits. This would help to identify what parameters in the Guest OS can be profiled and what useful data can be gathered from the profiling. In this Appendix, is given the conditions under which the system will cause a VM Exit. Some of these are conditional, while others will unconditionally cause a VM Exit.

Instructions That Cause VM Exits Unconditionally:

The following instructions cause VM exits when they are executed in VMX non-root operation: CPUID, INVD, MOV from CR3. This is also true of instructions introduced with VMX, which include: VMCALL,2 VMCLEAR, VMLAUNCH, VMPTRLD, VMPTRST,
VMREAD, VMRESUME, VMWRITE, VMXOFF, and VMXON.

Instructions That Cause VM Exits Conditionally:

Certain instructions cause VM exits in VMX non-root operation depending on the setting of the VM-execution controls. The following instructions can cause "fault-like" VM exits based on the conditions described:

• **CLTS**. The CLTS instruction causes a VM exit if the bits in position 3 (corresponding to CR0.TS) are set in both the CR0 guest/host mask and the CR0 read shadow.

• **HLT.** The HLT instruction causes a VM exit if the "HLT exiting" VM-execution control is 1.

• **IN, INS/INSB/INSW/INSD, OUT, OUTS/OUTSB/OUTSW/OUTSD.** The behavior of each of these instructions is determined by the settings of the "unconditional I/O exiting" and "use I/O bitmaps" VM-execution controls:

— If both controls are 0, the instruction executes normally.

— If the "unconditional I/O exiting" VM-execution control is 1 and the "use I/O bitmaps" VM-execution control is 0, the instruction causes a VM exit.

— If the "use I/O bitmaps" VM-execution control is 1, the instruction causes a VM exit if it attempts to access an I/O port corresponding to a bit set to 1 in the appropriate I/O bitmap. If an I/O operation "wraps around" the 16-bit I/O-port space (accesses ports FFFFH and 0000H), the I/O instruction causes a VM exit (the "unconditional I/O exiting" VM-execution control is ignored if the "use I/O bitmaps" VM-execution control is 1).

• **INLVPG.** The INLVPG instruction causes a VM exit if the "INLVPG exiting" VM-execution control is 1.

• **LMSW.** In general, the LMSW instruction causes a VM exit if it would write, for any bit set in the low 4 bits of the CR0 guest/host mask, a value different than the corresponding bit in the CR0 read shadow. Note that LMSW never clears bit 0 of CR0 (CR0.PE). Thus, LMSW causes a VM exit if either of the following are true:

— The bits in position 0 (corresponding to CR0.PE) are set in both the CR0 guest/mask and the source operand, and the bit in position 0 is clear in the CR0 read shadow.

— For any bit position in the range 3:1, the bit in that position is set in the CR0 guest/mask and the values of the corresponding bits in the source operand and the CR0 read shadow differ.
• **MONITOR.** The MONITOR instruction causes a VM exit if the “MONITOR exiting” VM-execution control is 1.
• **MOV from CR8.** The MOV from CR8 instruction (which can be executed only in 64-bit mode) causes a VM exit if the “CR8-store exiting” VM-execution control is 1. Note that, if this control is 0, the behavior of the MOV from CR8 instruction is modified if the “use TPR shadow” VM-execution control is 1
• **MOV to CR0.** The MOV to CR0 instruction causes a VM exit unless the value of its source operand matches, for the position of each bit set in the CR0 guest/host mask, the corresponding bit in the CR0 read shadow. (If every bit is clear in the CR0 guest/host mask, MOV to CR0 cannot cause a VM exit.)
• **MOV to CR3.** The MOV to CR3 instruction causes a VM exit unless the value of its source operand is equal to one of the CR3-target values specified in the VMCS. Note that, if the CR3-target count in *n*, only the first *n* CR3-target values are considered; if the CR3-target count is 0, MOV to CR3 always causes a VM exit.
• **MOV to CR4.** The MOV to CR4 instruction causes a VM exit unless the value of its source operand matches, for the position of each bit set in the CR4 guest/host mask, the corresponding bit in the CR4 read shadow.
• **MOV to CR8.** The MOV to CR8 instruction (which can be executed only in 64-bit mode) causes a VM exit if the “CR8-load exiting” VM-execution control is 1. Note that, if this control is 0, the behavior of the MOV to CR8 instruction is modified if the “use TPR shadow” VM-execution control is 1 and it may cause a trap-like VM exit.
• **MOV DR.** The MOV DR instruction causes a VM exit if the “MOV-DR exiting” VM-execution control is 1.
• **MWAIT.** The MWAIT instruction causes a VM exit if the “MWAIT exiting” VM-execution control is 1.
• **PAUSE.** The PAUSE instruction causes a VM exit if the “PAUSE exiting” VM-execution control is 1.
• **RDMSR.** The RDMSR instruction causes a VM exit if any of the following are true:
— The “use MSR bitmaps” VM-execution control is 0.
— The value of RCX is not in the range 00000000H – 00001FFFH or C0000000H – C0001FFFH.
— The value of RCX is in the range 00000000H – 00001FFFH and the *n*th bit in read bitmap for low MSRs is 1, where *n* is the value of RCX.
— The value of RCX is in the range C0000000H – C0001FFFH and the *n*th bit in read bitmap for high MSRs is 1, where *n* is the value of RCX & 00001FFFH.
• **RDPMC.** The RDPMC instruction causes a VM exit if the “RDPMC exiting” VM-execution control is 1.
• **RDTSC.** The RDTSC instruction causes a VM exit if the “RDTSC exiting” VM-execution control is 1.
• **RSM.** The RSM instruction causes a VM exit if executed in system-management mode (SMM)
• **WRMSR.** The WRMSR instruction causes a VM exit if any of the following are true:
— The “use MSR bitmaps” VM-execution control is 0.

— The value of RCX is not in the range 00000000H – 00001FFFH or C0000000H – C0001FFFH.
— The value of RCX is in the range 00000000H – 00001FFFH and the *n*th bit in write bitmap for low MSRs is 1, where *n* is the value of RCX.
— The value of RCX is in the range C0000000H – C0001FFFH and the *n*th bit in write bitmap for high MSRs is 1, where *n* is the value of RCX & 00001FFFH.
The MOV to CR8 instruction (which can be executed only in 64-bit mode) may cause a "trap-like" VM exit. This means that the instruction completes before the VM exit occurs and that processor state is updated by the instruction (for example, the value of RIP saved in the guest-state area of the VMCS references the next instruction).
Specifically, a VM exit occurs after execution of MOV to CR8 if the following are true:
• The "CR8-load exiting" VM-execution control is 0.
• The "use TPR shadow" VM-execution control is 1.
• The execution of MOV to CR8 reduces the value of the TPR shadow below that of the TPR threshold VM-execution control field

APIC-Access VM EXITS:
If the "virtualize APIC accesses" VM-execution control is 1, an attempt to access memory using a physical address on the APIC-access page causes a VM exit. Such a VM exit is called an **APIC-access VM exit**.
In general, an operation that attempts to access memory with a physical address on the APIC-access page causes an APIC-access VM exit.

Instructions That May Cause Page Faults Without Accessing Memory:
APIC-access VM exits may occur as a result of executing an instruction that can cause a page fault even if that instruction would not access the APIC-access page.
The following are some examples:
• The CLFLUSH instruction is considered to read from the linear address in its source operand. If that address translates to one on the APIC-access page, the instruction causes an APIC-access VM exit.
• The ENTER instruction causes a page fault if the byte referenced by the final value of the stack pointer is not writable (even though ENTER does not write to that byte if its size operand is non-zero). If that byte is writable but is on the APIC-access page, ENTER causes an APIC-access VM exit.1
• An execution of the MASKMOVQ or MASKMOVDQU instructions with a zero mask may or may not cause a page fault if the destination page is unwritable (the behavior is implementation-specific). An execution with a zero mask causes an APIC-access VM exit only on processors for which it could cause a page fault.
• The MONITOR instruction is considered to read from the effective address in EAX. If the linear address corresponding to that address translates to one on the APIC access page, the instruction causes an APIC-access VM exit.
• An execution of the PREFETCH instruction that would result in an access to the APIC-access page does not cause an APIC-access VM exit.

Other causes of VM Exits:
In addition to VM exits caused by instruction execution, the following events can cause VM exits:
• **Exceptions.** Exceptions (faults, traps, and aborts) cause VM exits based on the exception bitmap (see Section 20.6.3). If an exception occurs, its vector (in the range 0–31) is used to select a bit in the exception bitmap. If the bit is 1, a VM exit occurs; if the bit is 0, the exception is delivered normally through the guest IDT. This use of the exception bitmap applies also to exceptions generated by the instructions INT3, INTO, BOUND, and UD2.
Page faults (exceptions with vector 14) are specially treated. When a page fault occurs, a logical processor consults (1) bit 14 of the exception bitmap; (2) the error code produced with the page fault [PFEC]; (3) the page-fault error-code mask field [PFEC_MASK]; and (4) the page-fault error-code match field [PFEC_MATCH]. It checks if PFEC & PFEC_MASK = PFEC_MATCH. If there is equality, the specification of bit 14 in the exception bitmap is followed (for example, a VM exit occurs if that bit is set). If there is inequality, the meaning of that bit is reversed (for example, a VM exit occurs if that bit is clear).
Thus, if the design requires VM exits on all page faults, software can set bit 14 in the exception bitmap to 1 and set the page-fault error-code mask and match fields each to 00000000H. If the design does not require VM exits on page faults, software could set bit 14 in the exception bitmap to 1, set the page-fault errorcode mask field to 00000000H, and set the page-fault error-code match field to FFFFFFFFH.
• **External interrupts.** An external interrupt causes a VM exit if the "externalinterruptexiting" VM-execution control is 1. Otherwise, the interrupt is delivered normally through the IDT. (If a logical processor is in the shutdown state or the wait-for-SIPI state, external interrupts are blocked. The interrupt is not delivered through the IDT and no VM exit occurs.)
• **Non-maskable interrupts (NMIs).** An NMI causes a VM exit if the "NMI exiting" VM-execution control is 1. Otherwise, it is delivered using descriptor 2 of the IDT. (If a logical processor is in the wait-for-SIPI state, NMIs are blocked. The NMI is not delivered through the IDT and no VM exit occurs.)
• **INIT signals.** INIT signals cause VM exits. A logical processor performs none of the operations normally associated with these events. Such exits do not modify register state or clear pending events as they would outside of VMX operation. (If a logical processor is in the wait-for-SIPI state, INIT signals are blocked. They do not cause VM exits in this case.)
• **Start-up IPIs (SIPIs). SIPIs cause VM exits.** If a logical processor is not in the wait-for-SIPI activity state when a SIPI arrives, no VM exit occurs and the SIPI is discarded. VM exits due to SIPIs do not perform any of the normal operations associated with those events: they do not modify register state as they would outside of VMX operation. (If a logical processor is not in the wait-for- SIPI state, SIPIs are blocked. They do not cause VM exits in this case.)
• **Task switches.** Task switches are not allowed in VMX non-root operation. Any attempt to effect a task switch in VMX non-root operation causes a VM exit.

• **System-management interrupts (SMIs).** If the logical processor is using the dual-monitor treatment of SMIs and system-management mode (SMM), SMIs cause SMM VM exits.

In addition, there are controls that cause VM exits based on the readiness of guest software to receive interrupts:

• If the "interrupt-window exiting" VM-execution control is 1, a VM exit occurs before execution of any instruction if RFLAGS.IF = 11 and there is no blocking of events by STI or by MOV SS. Such a VM exit occurs immediately after VM entry if the above conditions are true.

Non-maskable interrupts (NMIs) and higher priority events take priority over VM exits caused by this control. VM exits caused by this control take priority over external interrupts and lower priority events.

These VM exits wake a logical processor from the same inactive states as would an external interrupt. Specifically, they wake a logical processor from the states entered using the HLT and MWAIT instructions. These VM exits do not occur if the logical processor is in the shutdown state or the wait-for-SIPI state.

• If the "NMI-window exiting" VM-execution control is 1, a VM exit occurs before execution of any instruction if there is no virtual-NMI blocking and there is no blocking of events by MOV SS. (A logical processor may also prevent such a VM exit if there is blocking of events by STI.) Such a VM exit occurs immediately after VM entry if the above conditions are true.

Debug-trap exceptions and higher priority events take priority over VM exits caused by this control. VM exits caused by this control take priority over non maskable interrupts (NMIs) and lower priority events.

These VM exits wake a logical processor from the same inactive states as would an NMI. Specifically, they wake a logical processor from the shutdown state and from the states entered using the HLT and MWAIT instructions. These VM exits do not occur if the logical processor is in the wait-for-SIPI state.

BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI
DISTANCE LEARNING PROGRAMMES DIVISION
First Semester 2007-2008
Fax No. 01596-244183

BITS ZG629T: Dissertation EC-2 Mid-Semester Progress Evaluation Sheet

Softcopy of Mid-Semester Report to be uploaded by July 31, 2007

Signed hard copy scheduled date to reach Pilani: August 7, 2007

ID No. **: 2005HZ13069**

NAME OF THE STUDENT **: Ajay Harikumar**

EMAIL ADDRESS **: Ajay.Harikumar@Intel.com**

SUPERVISOR'S NAME **: Ravindra B Bodhe**

DISSERTATION TITLE **: VMM Based OS Profiler (**Design and implementation of a light-weight Virtual Machine Monitor (VMM) using Virtualization Technology for the purpose of profiling the OS running on top of the VMM)

EVALUATION

DISSERTATION PROGRESS EVALUATION *(Please put a tick (✔) mark in the appropriate box)*

EC No.	Component	Excellent	Good	Fair	Poor
1.	Dissertation Outline				
2.	Mid-Semester Progress Report				

Remarks of the Supervisor:

	Supervisor	Additional Examiner
Name	Ravindra B Bodhe	Sasikanth Avancha
Qualification	M Tech (Computer Science), IIT Bombay	Ph. D. UMBC
Designation & Address	Engineering Manager c/o Intel Technologies India Pvt. Ltd. 28, Cubbon Road, Salarpuria Chambers, Bangalore - 560001	Research Scientist c/o Intel Technologies India Pvt. Ltd. 28, Cubbon Road, Salarpuria Chambers, Bangalore - 560001
Phone No.	+91 80 2262 3091	+91 80 2262 3461
Email Address	Ravindra.Bodhe@intel.com	Sasikanth.Avancha@intel.com
Signature		
Date		

Checklist of items for the Final Dissertation Report

This checklist is to be attached as the last page of the report.

This checklist is to be duly completed, verified and signed by the student.

1.	Is the report properly **hard bound? Spiral bound, softbound reports are not acceptable.**	Yes / No
2.	Is the Cover page in proper format as given in Annexure A?	Yes / No
3.	Is the Title page (Inner cover page) in proper format?	Yes / No
4.	(a) Is the Certificate from the Supervisor in proper format? (b) Has it been signed by the Supervisor?	Yes / No Yes / No
5.	Is the Abstract included in the report properly written within one page? Have the keywords been specified properly?	Yes / No Yes / No
6.	Is the title of your report appropriate? The title should be adequately descriptive, precise and must reflect scope of the actual work done.	Yes / No
7.	Have you included the List of abbreviations / Acronyms? Uncommon abbreviations / Acronyms should not be used in the title.	Yes / No
8.	Does the Report contain a summary of the literature survey?	Yes / No
9.	Does the Table of Contents include page numbers? (i). Are the Pages numbered properly? (ii). Are the Figures numbered properly? (Figure Numbers and Figure Titles at the bottom of the figures) (iii). Are the Tables numbered properly? (Table Numbers and Table Titles at the top of the tables) (iv). Are the Captions for the Figures and Tables proper? (v). Are the Appendices numbered properly?	 Yes / No Yes / No Yes / No Yes / No Yes / No
10.	Is the conclusion of the Report based on discussion of the work?	Yes / No
11.	Are References or Bibliography given at the end of the Report? Have the References been cited properly inside the text of the Report? Is the citation of References in proper format?	Yes / No Yes / No Yes / No
12.	Have you written your report according to the guidelines? The report should not be a mere printout of a Power Point Presentation. Source code need not be included in the report.	Yes / No
13.	A Compact Disk (CD) containing the softcopy of the Final Report and a copy of the Final Seminar Presentation made to the Supervisor / Examiner (both preferably in PDF format only) has been placed in a protective jacket securely fastened to the inner back cover of the Final Report. **Please write your name and ID No with a marker on the CD as well as the CD Jacket.**	Yes / No

Declaration by Student:

I certify that I have properly verified all the items in this checklist and ensure that the report is in proper format as specified in the course handout.

Place: Bangalore

Date: _____

_____

Signature of the Student

Name: Ajay Harikumar

ID No.: 2005 HZ13069