**Abstract**

The Xen Virtual Machine Monitor has proven to achieve higher efficiency in virtualizing the x86 architecture than competing x86 virtualization technologies. This makes virtualization on the x86 platform more feasible in High-Performance and mainframe computing, where virtualization can offer attractive solutions for managing resources between users. Virtualization is also attractive on the Itanium architecture. Future x86 and Itanium computer architectures include extensions which make virtualization more efficient. Moving to virtualizing resources through Xen may ready computer centers for the possibilities offered by these extensions.

The Itanium architecture is "uncooperative" in terms of virtualization. Privilege-sensitive instructions make full virtualization inefficient and impose the need for *para-virtualization*. Para-virtualizing Linux involves changing certain native operations in the guest kernel in order to adapt it to the Xen virtual architecture. Minimum para-virtualizing impact on Linux is achieved by, instead of replacing illegal instructions, trapping them by the hypervisor, which then emulates them. *Transparent para-virtualization* allows the same Linux kernel binary to run on top of Xen and on physical hardware.

Itanium *region registers* allow more graceful distribution of memory between guest operating systems, while not disturbing the Translation Lookaside Buffer. The *Extensible Firmware Interface* provides a standardized interface to hardware functions, and is easier to virtualize than legacy hardware interfaces.

The overhead of running para-virtualized Linux on Itanium is reasonably small and measured to be around 4.9 %. Also, the overhead of running transparently para-virtualized Linux on physical hardware is reasonably small compared to non-virtualized Linux.

## Preface

This master's thesis was written at the European Organization for Nuclear Research (CERN), while working for the CERN OpenLab for DataGrid Applications collaboration. The work is a continuation of previous research at CERN and NTNU, which has been conducted with fellow student Rune Andresen. We have looked into the application of virtualization with Xen in CERN's computational resources. CERN's computational Grid is being developed mainly for the purpose of collecting and analyzing data which will be produced by detectors in the Large Hadron Collider (LHC), after its experiments commence in 2007.

Most of the work described in this thesis has been conducted by Dan Magenheimer at HP Labs Fort Collins, the Xen Team at the University of Cambridge and affiliates of Intel and SGI. My work at the CERN OpenLab has been in assisting Dan Magenheimer in porting Xen to the Itanium architecture. This has involved testing, debugging and analyzing Xen and developing support for multiple domains on Itanium.

I thank Sverre Jarp, Anne C. Elster and Dan Magenheimer for giving me the opportunity to work on this project; the developers on the Xen mailing list and, again, Dan, for long-distance, but helpful, e-mail discussions; Rune Andresen and Dan for helpful feedback on this thesis; and Andreas Hirstius for helping with hardware and fixing things that break.

# Contents


**1 Introduction** **1**
1.1 Virtualization in the Grid . . . . . . . . . . . . . . . . . . . . . . . 2

**2 Background** **6**
2.1 Commodity Computer Architecture . . . . . . . . . . . . . . . . . 7
2.1.1 Interrupts . . . . . . . . . . . . . . . . . . . . . . . . . . 8
2.1.2 Privilege Separation . . . . . . . . . . . . . . . . . . . . . 9
2.1.3 Memory Management . . . . . . . . . . . . . . . . . . . . 9
2.2 The x86 Architecture . . . . . . . . . . . . . . . . . . . . . . . . 11
2.2.1 Memory Management . . . . . . . . . . . . . . . . . . . . 11
2.2.2 Process Switching . . . . . . . . . . . . . . . . . . . . . . 13
2.2.3 The Floating Point Unit . . . . . . . . . . . . . . . . . . . 13
2.2.4 Interrupts . . . . . . . . . . . . . . . . . . . . . . . . . . 14
2.3 The IA-64 Architecture . . . . . . . . . . . . . . . . . . . . . . . 15
2.3.1 Key Features . . . . . . . . . . . . . . . . . . . . . . . . . 15
2.3.2 Registers . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
2.3.3 Hardware Abstraction . . . . . . . . . . . . . . . . . . . . 17
2.3.4 Interrupts . . . . . . . . . . . . . . . . . . . . . . . . . . 18
2.3.5 Memory Management . . . . . . . . . . . . . . . . . . . . 18
2.3.6 The Performance Monitoring Unit . . . . . . . . . . . . . 19
2.4 Linux . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
2.4.1 Interrupts . . . . . . . . . . . . . . . . . . . . . . . . . . 21
2.4.2 Time . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
2.4.3 Processes . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
2.4.4 Kernel Entry & Exit . . . . . . . . . . . . . . . . . . . . . 24
2.4.5 Performance Monitoring . . . . . . . . . . . . . . . . . . . 25
2.5 Virtualization . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
2.5.1 Virtual Machine Monitors . . . . . . . . . . . . . . . . . . 27
2.6 Xen Virtual Machine Monitor . . . . . . . . . . . . . . . . . . . . 28
2.6.1 Paravirtualization . . . . . . . . . . . . . . . . . . . . . . 29
2.6.2 Event Handling . . . . . . . . . . . . . . . . . . . . . . . 30
2.6.3 Memory Management . . . . . . . . . . . . . . . . . . . . 30
2.6.4 Time . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31

**3 Analysis** **32**
3.1 Virtualization on Intel Architecture . . . . . . . . . . . . . . . . . 32
3.2 The Xen Hypervisor . . . . . . . . . . . . . . . . . . . . . . . . . 35
3.2.1 Para-virtualization . . . . . . . . . . . . . . . . . . . . . . 36

3.2.2 System Startup . . . 36
3.2.3 Memory Management . . . 37
3.2.4 Hypercalls . . . 38
3.2.5 Event Channels . . . 42
3.2.6 Time and Scheduling . . . 43

**4 IA-64 Implementation** **46**
4.1 Analysis . . . 46
4.1.1 Privilege-Sensitive Instructions . . . 47
4.1.2 Optimized Para-virtualization . . . 47
4.1.3 Transparent Para-virtualization . . . 48
4.1.4 Memory Management . . . 48
4.1.5 Hypercalls . . . 49
4.1.6 Virtual Hash Page Table . . . 52
4.1.7 Performance Monitoring . . . 52
4.2 Para-virtualization . . . 53
4.3 Optimized Para-virtualization . . . 55
4.4 Memory Management . . . 56
4.5 Firmware Interface Emulation . . . 56
4.6 Events . . . 57
4.7 Further Work . . . 57

**5 Performance Analysis** **59**
5.1 Experimental Environment . . . 59
5.2 Methodology . . . 59
5.2.1 Build Benchmark . . . 60
5.2.2 Instrumentation . . . 61
5.3 Results . . . 61
5.3.1 Build Benchmark . . . 61
5.3.2 Instrumentation . . . 62

**6 Related Work** **64**
6.1 Intel Virtualization Technology . . . 65

**7 Conclusion** **67**

**A Registers** **68**
A.1 Some Essential x86 Registers . . . 68
A.2 Some Essential IA-64 Registers . . . 69

**B Instruction Set Architectures** **70**
B.1 Some Essential x86 Instructions . . . 70
B.2 Some Essential IA-64 Instructions . . . 70

**C Source Code** **71**
C.1 Event Channel Hypercall . . . 71
C.2 Hypercall Handlers in the Hypervisor . . . 72
C.3 Privileged Operation Counters . . . 73
C.4 Paravirtualization . . . 73

# List of Figures


1.1 The grid layer stack. . . . . . . . . . . . . . . . . . . . . . . . . . 4
1.2 An alternative grid implementation. . . . . . . . . . . . . . . . . . 4

2.1 Virtual address to physical address translation. . . . . . . . . . . 10
2.2 Privilege separation in Linux on the x86 architecture. . . . . . . 12
2.3 The format of a bundle. . . . . . . . . . . . . . . . . . . . . . . . . 16
2.4 Flowchart showing how interrupts are handled in Linux. . . . . . 22
2.5 Illustration of a VMM hosted on a host OS, running two guest OSs. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
2.6 The Xen hypervisor, hosting several domains. . . . . . . . . . . . 29
2.7 Device virtualization in Xen. . . . . . . . . . . . . . . . . . . . . . 31

3.1 Different methods of virtualization. . . . . . . . . . . . . . . . . . . 34
3.2 The `domain` and `vcpu` data structures. . . . . . . . . . . . . . . . 36
3.3 Virtual block device initialization phase. . . . . . . . . . . . . . . . 44
3.4 The states and transitions of a frontend block device. . . . . . . . 44

4.1 Architecture specific parts of the `domain` and `vcpu` data structures. 55

5.1 Bar chart showing the times measured in the build benchmark. . 61
5.2 A chart over the number of para-virtualized operations executed. 63

6.1 VT's VMX Root and Nonroot execution environments. . . . . . . 66

# List of Tables

3.1 The hypercalls of Xen/x86. . . . . . . . . . . . . . . . . . . . . . . . 39

5.1 The configuration of the OpenLab GRID testbed cluster. . . . . 59
5.2 Measured time when compiling Linux. . . . . . . . . . . . . . . . . 61
5.3 Measured number of para-virtualized operations executed. . . . . 62

# Chapter 1

# Introduction

The Xen Virtual Machine Monitor is a virtualization project from the university of Cambridge. A presentation of the software is given in the paper "Xen and the Art of Virtualization" by Paul Barham et al. [B+03]. Xen has recently gained much popularity partly due to its efficiency and free and open source licensing. The software developer and user community around the software has grown significantly over the last few months, due not only to its inherent benefits, but also to the attention it has received through popular computer journalism, which has further boosted its development and given momentum to the project. Powerful computer technology companies such as Sun Microsystems, Hewlett-Packard, Novell, Red Hat, Intel, Advanced Micro Devices (AMD), Voltaire and IBM, have also recognized the potential in the software and have become involved with the project [Sha05].

Experiences show that Xen is able to utilize commodity PC resources better than most VMMs, yielding close to native OS (Operating System) resource utilization [And04, B+03, BA04, F+04b]. Compared to similar technologies such as VMWare and User Mode Linux, the performance benefits prove to be significant in most applications. Xen achieves this performance advantage through an unconventional approach to virtualizing a computer, called *para-virtualization* [W+02]. One significant difference from traditional approaches is that this one allows only modified OSs to run on the virtual machine.

Traditional Virtual Machine Monitors (VMM) have aimed to be able to let any, or at least a few, OSs run unmodified in their VMM execution environment, that is, to provide a true virtualization of the architecture. This is necessary in order to run, for example, Windows XP on the VMM, since most regular users and software devolpers are not allowed nor able to modify the OS in order to have it run on custom architectures, be they virtual or real. With the emergence of main-stream open source licensed OSs, such as Linux and OpenBSD, software developers are given more freedom to modify the OSs so that they may possibly run on a given computer architecture. The Xen developers have taken advantage of this freedom by, instead of completely virtualizing the PC architecture, providing a more benificial *Xen/x86* virtual architecture on which a modified OS may run.

Many novel applications for virtualization have emerged over the development of Xen as, more so than before, researchers and users now have access to a free, open, and efficient virtualization platform on cheap and powerful comput-

ers. Applications such as checkpointing, server consolidation and self-migrating OSs have become more feasible. Virtualization of computing resources is particularly attractive in HPC (High-Performance Computing), since this usually involves the sharing of computing resources among several users. By virtualizing resources, they may be shared between users more dynamically and securely.

Major computer chip manufacturers such as Intel and AMD, have also recognized the benefits of having multiple OS instances run simultaneously on a single processor. Recently, specifications for their Vanderpool (VT) and Pacifica processors have been released, which include facilities for enabling efficient true virtualization. Support for the Vanderpool virtualization architecture has already been implemented into Xen.

Xen has already shown to provide an efficient virtualization environment for x86 machines. The objective of this project is to attain and analyze a virtualization environment for the Itanium platform. This is important because grid computing resources consisting of Itanium machines can benefit from virtualization, as discussed in Section 1.1. The project involves assisting in the development of the Xen VMM for Itanium.

This master's thesis presents the work related to porting Xen to the Itanium architecture. Chapter 2 gives an introduction to some of the concepts used in further discussions in the thesis. Topics that are relevant for analysis of the virtualization of Itanium include the existing Xen implementation, current virtualization challenges on Intel architectures and specifics of the IA-64 architecture. These topics are discussed in Chapter 3. The implementation of Xen on Itanium is presented in Chapter 4. In order to evaluate the progress in terms of performance, Chapter 5 analyzes the performance of Xen/ia64. In Chapter 6, a few interesting projects that have relevance to Xen are discussed. Finally, Chapter 7 concludes the thesis.

## 1.1 Virtualization in the Grid

The construction of the Large Hadron Collider (LHC) at the European Organisation for Nuclear Research (CERN) is scheduled to complete in 2007. Many different high energy physics experiments will be performed in the LHC, by scientists from many parts of the world. An introduction to the LHC and its experiments can be found at [lhc05]. The particle detectors of the different experiments will generate massive amounts of data that need to be analyzed and stored on local and remote computing sites. It is estimated that roughly 15 PetaBytes of data will need to be processed and stored per year when the experiments start.

Computational grids are set to satisfy the demand for computing and storage resources, and many grid projects will be involved in the LHC experiments. Ian Foster and Carl Kesselman [FK] give a good introduction to grid computing. Also, previous work [Bje04] gives an overview of some important grid computing projects. One example grid project is the LHC Computing Grid project, which is being coordinated at CERN.

A computational grid can benefit from virtualization in a variety of ways. Xen is considered as a virtualization solution and has been evaluated for application in HPC in previous work with Rune Andresen [BA04]. The core parts of the LCG software stack has been ported to the Itanium architecture, which

makes the LCG ready to employ Itanium machines. At this moment, three computing centers are currently contributing Itanium computers to the LCG [HC05]. By porting Xen to the IA-64 architecture, it will be one step closer to being realized in grid environments consisting of Itanium machines, such as the Itanium computers of the LCG and the OpenLab testbed at CERN.

Amit Singh [Sin04] presents an extensive list of virtualization benefits, which probably can be extended to include many more. Staying within the scope of HPC, a shorter list, focusing on HPC virtualization benefits, is presented in the following.

- A general trend in integrated circuit design is that the number of transistors that can fit per die area increases more rapidly than processor companies' capacity to produce new logic. Thus, copying existing logic from one CPU core into a multi-core CPU, allows new die area to be utilized. Multi-core CPUs, however, face the same limitations as symmetric multiprocessing (SMP) systems with regards to programming paradigm, as they essentially are shared memory parallel machines. Virtualization allows computing resources to be utilized with more flexibility. Different OSs may be run at the same time, and on a SMP system, OSs may be pinned to individual CPUs. With the arrival of multi-core CPUs, virtualization will allow these resources to be shared among users on a virtualized hardware level.

- Parties that have interest in grid computing resources, such as research institutions or commercial organizations, are organized into *Virtual Organizations (VO)*. It is important that computations of one VO do not interfere with computations of another VO. Also, commercial VOs might want to keep their computations secret from competing VOs. Computations running in a virtualized environment may be *isolated* (see Section 2.5) while running on the same physical resources.

  Nancy L Kelem and Richard J. Feiertag [KF91] argue that the isolation provided by a hardware virtualization mechanism can be more easily mathematically abstracted than the isolation provided by a traditional time-sharing OS's security model. The sufficiency of the hardware virtualization security model can be evaluated by looking at the isolation at the hardware level. Since hardware virtualization is a simpler mechanism than an OS and has fewer lines of code, its isolation can be more easily proven.

- When physicists submit a job to a grid, there is a probability that the job will fail. On a given grid resource, this probability increases with the amount of time the computation runs. The physicist may waste a lot of time or be delayed when after a while into the computation, finding that the computation has failed due to grid failure. Virtualization can allow the state of the entire OS to be stored in *checkpoints*, since the virtualization environment *encapsulates* the OS (see Section 2.5). In the case of a failure, the OS can then be brought back to the state it was in before the failure and continue from there.

- Also, because of encapsulation, a running OS with a computation can be *migrated* from one node in the grid to another and continue to run. This allows the load on the nodes to be balanced more dynamically, and

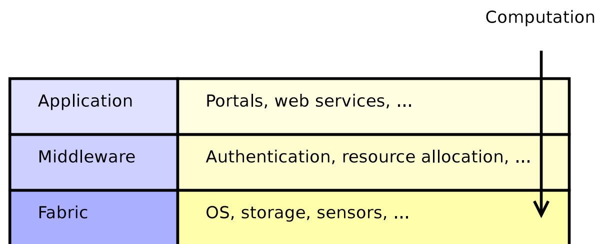


Figure 1.1: The grid layer stack.

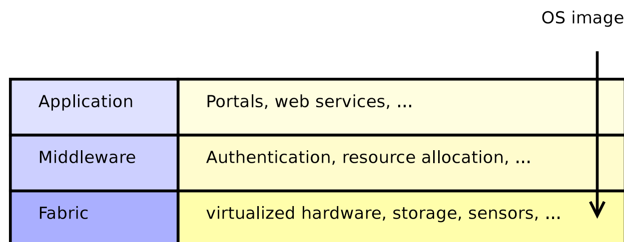


Figure 1.2: An alternative grid implementation.

computations may be moved away from a node which is going down for maintenance.

- An illustration of the grid layer stack is shown in Figure 1.1. The *fabric* layer of the grid layer stack consists of, among others, OSs, and this is the execution environment in which grid computations effectively run. A computational job is first submitted by the user through a high level user interface, such as, for instance, a web portal, in the *Application* layer. Then the user is authenticated, and the job is allocated resources in the *Middleware* layer. Finally, the job is run on a node in the *Fabric* layer on the execution environment it provides. In the case of LCG, this is *Scientific Linux CERN* [slc05].

  Grid computations depend on necessary libraries and dependencies being satisfied in these OSs. With virtualization, virtualized hardware is exposed at the fabric layer, as illustrated in Figure 1.2. Instead of submitting a computational job, the user may submit a complete OS with a filesystem, containing a computation. The submission and authentication proceeds as usual, but at the fabric layer, the user is allocated virtualized hardware. Thus, each user may be given the responsibility of satisfying their own software dependencies, supplied in their own OS submissions.

- Virtualization may enable the utilization of public computing resources, such as university computer labs or an organization's workstations. While one part of a workstation serves the original purpose of providing a tool for students' or employees' work, another part may serve as a computational resource for a grid, for example during the night when they usually are not used, or otherwise during low utilization.

- Encapsulation allows the monitoring and provisioning of hardware resources. This further allows the account keeping of VOs' resource usage, and resources can be brokered between VOs.

- Upgrades to OSs and software can be done on virtualized hardware without bringing down the computer. In a transitional phase, both old and upgraded OSs may run simultaneously on the same computer, allowing upgrades without downtime.

The potential drawbacks of virtualization are also worth mentioning. Multiple processes running simultaneously on the same CPU lead to the cache being overwritten subsequently by the processes. This is often referred to as *thrashing* of the cache and leads to poor utilization of the cache and a significant slowdown. Naturally, processes running simultaneously in different OSs on the same CPU will lead to the same problem. Thus, running two OSs simultaneously on a single CPU is not considered beneficial in HPC when running computations, in terms of resource utilization.

# Chapter 2

# Background

An OS is a complex piece of software. It manages the computer hardware and provides a foundation on which user application software can run. All the hardware devices of a computer, such as harddrives, network interface cards, keyboards and monitors are managed by the OS. Also, the OS abstracts an interface to memory management, processes, timers and file systems in order to make programming of user application software easier. Without an OS, a computer is essentially useless.

Xen is in many ways similar to an OS. It provides a foundation on which OSs can run, and, similarly to an OS, it has to manage the computer directly. Programming Xen therefore requires knowledge about the computer architecture it is programmed for. Xen is programmed for the *x86*[1] architecture. Knowing the x86 architecture is therefore necessary in order to understand the design of many parts of Xen software, which are specifically programmed for the x86 architecture. The x86 architecture is presented in Section 2.2.

Knowing the *IA-64*[2] architecture is of course necessary when programming system software for it. Therefore, in order to understand how the IA-64 architecture is different from the x86 architecture and how system programming is affected by these differences, an introduction to the IA-64 architecture is given in Section 2.3. The two architectures do, however, have many similarities, which are basic in many commodity computer architectures. Therefore, some essential concepts of commodity computer architecture are presented in Section 2.1.

Xen is originally designed with the x86 architecture as the target environment. The x86 architecture was, however, probably never intended to support virtualization. This makes efficient virtualization of the architecture difficult to achieve, which is part of the reason why this has not been achieved with a near-to-native level of efficiency until recently. Guest OSs that run in the Xen system are modified to achieve high efficiency. Still, they let user applications run unmodified. Thus, many aspects of the original OS need to remain intact in order to still support their interfaces to the user applications. Xen must

[1] The term *x86* usually refers to Intel, AMD or other chip manufacturers' PC architectures that are backwards compatible with the 80386 architecture, except for the ones that use 64-bit extensions, namely AMD's Opteron and Intel's EMT64 technologies. The term *IA-32* is also often used.

[2] The term *IA-64* refers to Intel's Itanium processor series, also referred to as *Itanium Processor Family (IPF)*.

therefore facilitate for many aspects of a traditional OS such as memory management, device drivers and I/O. How Xen provides an efficient platform for virtualization is discussed in Section 2.6.

Xen heavily leverages code from Linux both in the virtualizing software and in guest Linux kernels. Understanding Linux helps to understand some general OS concepts and how Xen virtulizes hardware. Also, this helps to understand how guest OS are modified to run on Xen. A description of some important aspects of the Linux OS kernel is given in Section 2.4.

Some important principles in virtualization govern the development of Xen and are important guidelines that are necessary to follow when implementing on complementary architectures. Also, looking at other VMM technologies may help the understanding of Xen. Some important principles of virtual machines are discussed in Section 2.5, along with some state of the art examples of virtualization technologies.

## 2.1 Commodity Computer Architecture

The von Neumann model is by far the most prevalent paradigm for modern computer architectures. It specifies a processing unit and a storage unit, which cooperate to run a program. Indeed, the most common computer architecture on desktop computers in use today—the PC, or in more technical terms, *x86*, architecture (see Section 2.2)—is an architecture which stems from the von Neumann model.

X86 processors are usually relatively cheap and common and are therefore often called *commodity hardware*. Commodity computers such as the PC are popular in offices and private homes and are in these contexts used for diverse applications, such as word processing, games and multimedia. The rich requirements of these applications have evolved commodity desktop computers into powerful all-round computers, capable enough to be used in mainframe computers, both in application servers and scientific computations. Commodity hardware can be attractive in many HPC applications as well as desktop applications, but mainly because of its low total cost of ownership compared to specialized computer hardware. Integrated circuit technology evolves to allow transistors to be ever smaller, and new applications for commodity processors are realized. Recently, commodity computer architectures such as the x86, have been used also in embedded devices, showing how diverse applications evolve as integrated circuit logic becomes cheaper, less power consuming and smaller.

For some critical applications, however, commodity hardware might not suffice, or other computer architectures may give significant performance benefits compared to what x86 computers have to offer. While simply relying on the predictions of Gordon E. Moore [Moo65], that time can let computers catch up with the performance demands, for a long time often has been a practical solution, alternative architectures may provide benefits beyond what next generation x86 processors give. Intel aims, with the IA-64 architecture (see Section 2.3), to provide a set of architectural improvements in a new computer architecture more or less independent from the x86 architecture.

One thing that separates commodity computers from many other computers is the need for a diverse set of ways to interface with the human user, in order to satisfy the user's diverse set of requirements. This has played an important role

in the development of processor architectures. Many features exist to satisfy users' interaction with the system, to support a diverse set of applications and to increase performance of the system without asking too much of the system programmer. Some of these features are described in the following sections. *Interrupts* are the processor's method of communicating the event of a user interaction with the system to the processor, which is further discussed in Section 2.1.1. *Privilege separation* allows the system to be protected from programming mistakes, as discussed in Section 2.1.2. Being von Neumann architectures, commodity architectures also employ schemes for managing storage, which is discussed in Section 2.1.3.

### 2.1.1 Interrupts

In order for a computer to be useful it needs to be able to handle input from users and react to it. For instance, when a user types a character key on the keyboard, she expects the computer to react to it by for instance showing the typed character on the computer display. Similarly, the OS needs to be notified when a DMA (Direct Memory Access) transfer has completed or a packet arrives on the network, in order to react to it. When the processor receives the electrical signal that indicates the press of a certain key or the completion of a DMA transfer, it then interrupts the current running process and executes an *interrupt* procedure.

Interrupts are either *maskable* or *nonmaskable*, that is, either they can be disabled or they can not. Critical events, such as hardware failures, often give rise to nonmaskable interrupts. The procedure which a particular interrupt is to execute is called an Interrupt Service Routine (ISR). When an interrupt occurs, a special table is referenced, indexed by the types of interrupts, and the ISR corresponding to the interrupt is executed or called.

All devices that deliver interrupts are connected to an *interrupt controller* by an *Interrupt ReQuest (IRQ)* line. Whenever a device needs to interrupt the processor it raises an IRQ signal, telling the interrupt controller that it is ready to perform some operation. In basic terms, an x86 interrupt controller further

- stores a vector corresponding to the received signal into its I/O port,
- issues an interrupt to the CPU by raising a signal on the INTR pin,
- waits until the CPU acknowledges the interrupt and then clears the INTR pin.

The number of IRQ lines is limited, thus some devices share IRQ lines. IRQ sharing is achieved by having one ISR per device that uses a particular IRQ. Since the ISR does not know initially which device issued the interrupt, each ISR related to the particular IRQ is executed; however, each ISR first checks if it was its particular device that generated the interrupt. Another way devices share IRQs is by multiplexing them in time.

The processor does not only interrupt itself on external impulse. It may also issue interrupts when an instruction is erroneous or otherwise problematic, for instance on divide-by-zero errors or page faults. These internally generated interrupts are usually called *exceptions* rather than interrupts.

### 2.1.2 Privilege Separation

In a multi-user time-sharing OS there is a need to protect the underlying execution environment from malicious or erroneous use. For instance, normal users should not be allowed to alter the execution of kernel or other users' processes. Also, faulty user programs should not be able to break the execution of the kernel by, for instance, altering memory reserved for kernel processes. Thus there is a need for protecting the memory reserved for the kernel through a privilege separation mechanism. Similarly, certain instructions should only be allowed to be executed and, certain registers should only be allowed to be accessed by the kernel.

Commodity workstation and mainframe computer processors maintain privilege separation through protection of memory segments or pages (see Section 2.1.3). The protection information for a segment or page is stored as meta-information which is compared to the running process' privilege level. Also, certain registers and instructions, which, for instance, control the processor's operation or interaction with other hardware, are protected.

The currently running process' privilege level, usually referred to as the *Current Privilege Level (CPL)*, is kept in a register, which also allows the processor to check if the process is allowed to execute certain instructions or access certain registers. If a process with insufficient CPL tries to access protected registers or execute protected instructions, a *Privilege Operation fault* is raised by the CPU, which may be handled as any other exception and recovered or ignored.

### 2.1.3 Memory Management

Memory is usually addressed differently from a processor and from a user program's point of view. Managing memory directly by its physical addresses makes programming complex; intstead, the OS may allow memory to be accessed through *virtual* addresses. In a virtual addressing scheme, when an application addresses a particular memory location, it provides a virtual address, which then is translated into the corresponding physical address of the memory location.

There are different ways to virtualize memory access. The two most prevalent in commodity computer architecture are *segmentation* and *paging*. Segmentation is a technique for logically arranging memory such that different processes accessing memory do not interfere with each other, thus making programming easier and more secure. Having privilege separation on a segment lets certain processes have access to segments that other do not.

Through paging, memory is arranged into pages, which are disjunct memory ranges that are contiguous in physical memory and in the way they are presented as virtual memory. Similar to a segment, a page may demand that accesses to it are privileged. The difference, however, between segmentation and paging, is that when using segmentation, the program explicitly addresses and manipulates the segments, while using paging, the program uses a virtually linear address space which may be non-contiguous in reality. Segmentation and paging in the x86 architecture is described in Section 2.2.1.

In order to translate a virtual address into a physical address the OS looks up the address in a *page table*. The page table is a data structure stored in main memory, which stores the addresses of the individual pages. The procedure of

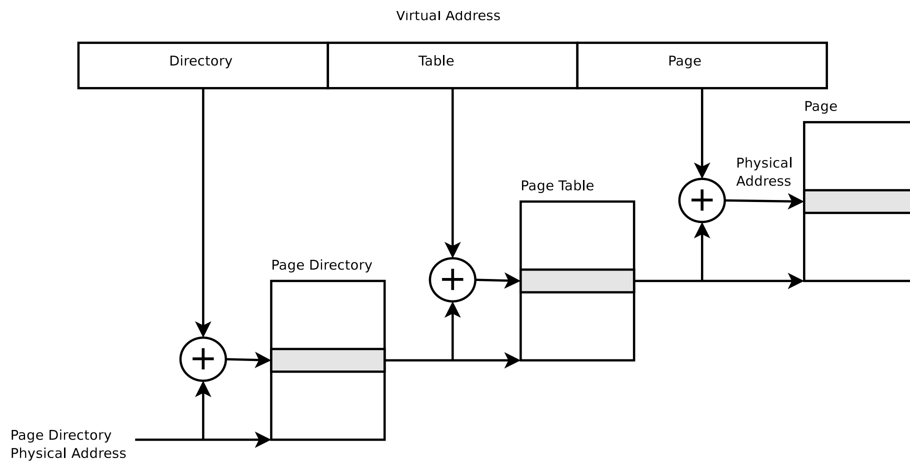


Figure 2.1: Virtual address to physical address translation.

translating a virtual address into a physical address is illustrated in Figure 2.1. As the number of pages is potentially very large, the data structure is expanded into a hierarchy. The top data structure is a *page directory*, which further holds several page tables. When a virtual address is translated through the page table, the *directory* part of the address points to a page table entry in the page directory. Further, the *table* part of the address points to a *Page Table Entry (PTE)* in the page table. Finally, the last part of the address is used as an offset within the page, which is the physical address translation of the original virtual address. This hierarchical scheme allows for not allocating memory for page tables until they are actually used, reducing the page table memory profile proportional to the number of pages used, rather than the number of pages in the whole virtual address space.

Having to look up addresses through main memory, however, is a costly affair. Given that a virtual to physical address translation may take two or more memory lookups in addition to the intended physical one, looking up the page directory, page table and possibly higher level data structures, the overhead of a pure main-memory approach is significant. Most processors therefore employ a *Translation Lookaside Buffer (TLB)* in order to speed up virtual address translation. The TLB acts as a cache, storing a subset of the possible translations, after a possibly speculative scheme with regards to which translation is most likely to occur next. As the TLB is implemented physically close to the core of the CPU, with high-speed logic, translations are performed with a significant speedup compared to translations through main memory. TLBs are, however, limited in the number of translations they can keep. The first time a translation is used, it must be found in the page table, with the same memory access overhead. Also, as new translations are used, they will replace older ones, which again may need to be translated through memory in the future.

## 2.2 The x86 Architecture

The x86 architecture fits quite nicely under the CISC (Complex Instruction Set Computer) paradigm. Traditionally, computers that fall under this paradigm, in contrast to RISC (Reduced Instruction Set Computer) computers, afford a less rich register set due to a higher richness in ISA (Instruction Set Architecture) complexity. Even though modern x86 CPUs may afford a richer register set, backwards compatibility often dictates that new registers are redundant.

Segmentation was the first memory management paradigm in the x86 processor. It remains a legacy from the processors that were developed before the 80386 processor. The 80386 processor introduced paging, which allowed the use of virtual memory. The x86 processors of today therefore support both segmentation and paging for managing main memory. How segmentation and paging of memory is managed in the x86 CPU is discussed in Section 2.2.1.

In Linux, control of the CPU is handed between programs through a mechanism called *process switching* (see Section 2.4). When control is handed from process $p$ to process $q$, the x86 CPU can automatically store some of the registers that are used by $p$, or $p$'s *hardware context*, to memory. After $q$ is finished, and $p$ is given control of the CPU again, the CPU automatically loads $p$'s hardware context from memory into the registers. In the case of segment registers and general purpose registers, this is further discussed in Section 2.2.2. Hardware management of FPU (Floating Point Unit) registers is discussed in Section 2.2.3.

The x86 processor as used in PC's has to interface with many different types of hardware. Hardware which is external to the CPU, can be communicated with through a set of different interfaces. *I/O ports* are the addresses which some devices are connected through. The I/O ports can be accessed either through special instructions, or they can be mapped in the physical address space. Also, as discussed in Section 2.1.1, devices deliver interrupts to the CPU to notify of certain events. The CPU also notifies itself about certain events, such as divide-by-zero errors and timer events. Interrupts are further discussed in Section 2.2.4.

### 2.2.1 Memory Management

Backwards compatibility has always been preserved in the x86 series architectures. Though segmentation strictly is not necessary for most modern applications, as it is made redundant by paging, the x86 maintains segmentation in order to allow support for existing software that uses segmentation. Segmentation was introduced to make memory management safer and more scalable, by encouraging or making it easier to manage separation of memory spaces between user space and kernel space and between code and data. This was achieved by having different, although possibly overlapping, memory segments for data and code and assigning them different privilege levels, giving heritage to *rings*, as illustrated in Figure 2.2. In Linux, only privilege levels zero and three are used, in which Kernel Mode and User Mode processes run, respectively. These privilege levels are commonly referred to as "rings".

Attributes of a segment, such as its size, its physical starting address and the privilege level required to access it, are stored in its *segment descriptor*. Segment descriptors are stored either in the *Global Descriptor Table (GDT)* or the *Local Descriptor Table (LDT)*, which are data structures in memory.

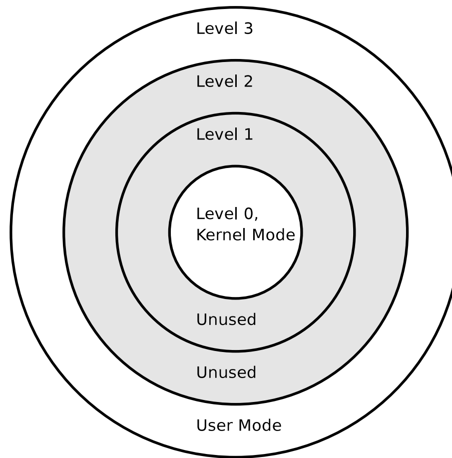


Figure 2.2: Privilege separation in Linux on the x86 architecture.

*Segment selectors* are used to reference the segment descriptors in the descriptor tables. A segment selector points to a specific segment descriptor in the GDT or LDT through an index and a flag that specifies whether the segment descriptor is in the GDT or in the LDT. The segment descriptor defines the level of privilege needed to read or write to its corresponding segment in the *Descriptor Privilege Level (DPL)* field.

To make segment addressing faster, the segment descriptor of the currently used segments are stored in segmentation registers. A particular segmentation register, which holds the segment identifier of the currently used code segment, namely the `cs` register, is significant because it also includes a 2-bit field called the Current Privilege Level (CPL). This field keeps the privilege level of the current executing process, one in the range zero through three. Level zero is the most privileged and allows access to all processor registers and to hardware device interfaces. Also interrupt handling instructions are instructions of privilege level zero . Intel's intention of rings one and two is for having the OS run device handling routines in these levels; however, they are rarely used. Ring three gives access only to a limited set of registers and instructions, excluding interrupt handling instructions and hardware device interfaces.

As segments have differentially privileged access, so do pages, though only two different levels, "User" and "Supervisor". Pages that are flagged "User" may be accessed by any process, while pages that are flagged "Supervisor" may only be accessed by a process when the CPL is less than three.

The pointer to the top level page directory of the current process is kept in the `cr3` control register. Whenever the address space needs to be changed, such as, for instance, after a process switch, the address of the next process' page directory is written to the `cr3` register. This, at the same time, triggers a flush of the TLB, invalidating all the TLB's translation entries.

### 2.2.2 Process Switching

The x86 CPU can automatically manage the contexts of different processes. It offers a mechanism for storing a complete CPU state except for FPU and *Streaming SIMD Extensions (SSE)* (see Section 2.2.3) states. The most essential registers that comprise this state are (see Appendix A.1):

- The segment registers—CS, DS and others.
- The general purpose registers—EAX, EBX, ECX and similar.
- The pointer to the last executed instruction—EIP.
- The pointer to the page directory—CR3.
- The pointer to the *Kernel Memory Stack* (see Section 2.4.4)—ESP.

In order to manage this state, the CPU uses a special segment called the *Task State Segment (TSS)*. This is the segment in which the context of a process is stored upon an automatic process switch. The TSS's segment descriptor is called the *Task Gate Descriptor*, which can be stored in a GDT, an LDT or an *Interrupt Descriptor Table (IDT)* (see Section 2.2.4). By performing a far JMP or CALL instruction, a task may target any segment visible to it through a GDT, LDT or IDT. As a task enters a segment its CPL is compared to the segment's DPL. If the CPL is more or equally privileged than the DPL the task is allowed to continue, and the CPL assumes the DPL. If the targeted segment is the TSS, the CPU automatically performs a hardware context switch.

### 2.2.3 The Floating Point Unit

The floating point capabilities of the x86 architecture has evolved with the demands for more faster and more efficient floating point calculations in 3D games, multimedia and HPC. The architectures that x86 is based on, started out without floating point capabilities, and the x87 floating point co-processor was later added to extend the processors, with floating point arithmetic capabilities. The external x87 co-processor was accessed as other external devices and results were returned with external interrupts. The FPUs of modern x86 processors are still logically x87 co-processors, although they are now integrated into the circuits of the CPU and thus much faster. Also, later, with the Pentium processor, the *MMX* extension allowed the FPU to be utilized more efficiently, directly through MMX registers. In addition, the Pentium III processor introduced the SSE extensions, which perform floating point arithmetic in the *XMM* registers, which are separate from the FPU registers. SSE allows one operation to be executed on multiple vectors in parallel in a single instruction unit. The SSE extensions have also evolved, and recently *SSE2* and *SSE3* extensions are included in Intel's x86 processors.

The x86 processor still generally performs floating point arithmetic in the FPU. The registers of the FPU are utilized either through `esc` instructions or, as of the Pentium processor, alternatively through MMX instructions. On a hardware context switch, the processor does not automatically save the FPU or XMM registers in the TSS as with general registers, so the OS needs to store and restore these registers manually. The processor does, however, help the OS

to save these registers only when needed, that is, only when a process really uses the FPU or XMM registers. That is, the FPU and XMM state is *lazily managed*.

In the CR0 register (see Appendix A.1), the TS flag indicates whether a context switch is performed, and on every hardware task switch the TS flag is automatically set. The TS flag may also be changed manually, although this requires a CPL of 0. When a new task executes it may or may not use the FPU or XMM registers. If it uses the FPU, the processor will issue an exception, telling the OS that the FPU context has not yet been loaded. The OS may then proceed to manually load the FPU context from the TSS and clear the TS flag. Further attempts to use the FPU or XMM registers will not raise exceptions as long as the TS flag remains cleared. If the process does not use the FPU, however, a FPU context switch is not necessary.

### 2.2.4 Interrupts

A wide range of external devices are connected through a *Programmable Interrupt Controller (PIC)*. The traditional PIC hardware has a few limitations, such as lack of support for SMP. It has later been replaced with the *Advanced Programmable Interrupt Controller (APIC)*, which is bacwards compatible with the traditional PICs but has support for SMP. All interrupts—including exceptions—in the x86 processor are handled through the *Interrupt Descriptor Table (IDT)*. Interrupts of external devices are handled by the ISRs as IRQs. Pointers to the ISRs of the interrupts and exceptions are stored in the IDT.

Examples of exceptions are *General Protection Fault* and *Page Fault*. The General Protection Fault is raised whenever the protection mechanisms of the processor are violated, for example, if a user process crosses into ring zero. The process should then be notified and possibly terminated. Page Faults are raised whenever a page that is referenced, is not present in the page table. The OS should then take the necessary actions to correct the page table and resume the execution of the faulting process.

One of the most important external interrupts is perhaps the timer interrupt. It is assigned the IRQ number zero on all x86 processors and is necessary, for instance, to share the time of the processor between processes, by interrupting a process when it has used a share of its time. The x86 processor offers a set of hardware mechanisms for letting software know the time:

- The *Real Time Clock (RTC)* is a clock that keeps wall clock time and runs even when the computer is powered off.
- The *Time Stamp Counter (TSC)* counts the number of CPU clock cycles since the computer was turned on, i.e. it runs with the same frequency as the CPU and accumulates from zero.
- The *Programmable Interval Timer (PIT)* can be programmed to deliver interrupts at a given time frequency.
- The *CPU Local Timer* is similar to the PIT, however it only delivers per CPU interrupts and is only present in recent x86 processors.

## 2.3 The IA-64 Architecture

The Intel Itanium architecture is one of the participants in the great 32-bit to 64-bit leap, which has had a significant impact on the processor industry the last couple of years. While AMD have chosen a 64 bit extension of the x86 architecture, Intel early decided to break with the old[3] x86 architecture and release the new 64-bit IA-64 architecture through the Itanium processor series.

The Itanium's shift from the x86 architecture introduces a new paradigm for programming, which Intel calls *Explicitly Parallel Instruction-set Computing (EPIC)*. A good introduction to the EPIC paradigm is given by Mark Smotherman [Smo02]. This new architectural paradigm shifts the responsibility of optimizing the execution of programs from the CPU over to the compiler or assembly programmer. The compiler or assembly programmer has the responsibility of programming explicitly parallel programs, specifying which instructions may execute in parallel, as opposed to Pentium architectures, where the CPU decides the issuing of instructions. At the same time, the IA-64 architecture has similarities to certain RISC architectures. Among other things, the IA-64 CPU has a large number of registers. Further features of the IA-64 architecture are summarized in Section 2.3.1. The IA-64 register set is discussed in Section 2.3.2.

The IA-64 processor's interface to hardware has had the opportunity to be designed from scratch, independently from legacy x86 interfaces, such as the BIOS (Basic Input Output System) and the PIC. This makes the new IA-64 hardware interfaces easier to program. The IA-64's equivalent to the BIOS, the *Extensible Firmware Interface (EFI)*, is described in Section 2.3.3. This is followed by a description of the IA-64 architecture's interface to interrupts, in Section 2.3.4.

Like the x86 architecture, the IA-64 architecture employs paging of memory. In order to make virtual address translation more efficient, it also employs a TLB. The IA-64 TLB is, however, more sophisticated and allows flexible management of address spaces. Also, maintaining compatibility towards x86 software, it has support for segmentation, in IA-32 compatibility mode. Memory management in the IA-64 architecture is further discussed in Section 2.3.5.

Some computer architectures provide dedicated facilities for monitoring system performance, such as the UltraSPARC series' Performance Control Register and Performance Instrumentation Counter [E+98]. The original 80386 specification did not include support for CPU performance monitoring. Support for this has been added later, and some of the x86 series' models, particularly newer models, do have performance monitoring facilities. These are, however, often obscured, sometimes intentionally, being unstandardized and undocumented. The IA-64 architecture makes performance monitoring easier, with its *Performance Monitoring Unit (PMU)*, which is discussed in Section 2.3.6.

### 2.3.1 Key Features

The Itanium incorporates a set of features that separate it from traditional architectures. These features play a part in defining the IA-64's ISA. One of the most distinctive features is the concept of *predication*. This is one of the features

[3] While some argue that the x86 architecture is too old and complicated after many patches and extensions, others argue "Don't fix it if it's not broken".

Figure 2.3: The format of a bundle.

which exposes the programmer with the explicitness of the EPIC paradigm. Predication avoids costly branch prediction misses by executing the resulting instructions of either condition—true or false—of a conditional branch while committing only the result of the condition which was eventually met.

Two other important concepts of the IA-64 ISA also expose the explicitness of the EPIC paradigm, namely *groups* and *bundles*. The software programmer is given the responsibility of exploiting the instruction level parallelism of the architecture. Instructions are issued one bundle at a time, and the instructions in a bundle may be executed in parallel, with a few exceptions. One bundle consists of up to three instructions, but rules define which instructions may be bundled together.

The structure of a bundle is illustrated in Figure 2.3. Three instruction *slots*, each 41 bits wide, can each hold an instruction. Several different types of instructions are defined [Intb]:

- *Memory (**M**)*—memory instructions.
- *Integer (**I**)*—integer instructions not using the ALU.
- *Integer (**A**)*—integer instructions using the ALU.
- *Branch (**B**)*—branch instructions.
- *Floating point (**F**)*—floating point instructions.
- *Extended (**L+X**)*—allows 64-bit immediate values to be used, as it spans two slots.

A *template*, residing in the least significant five bits of the bundle, defines what type of instruction is inserted into each slot. For instance, the template 0x10 defines that slot 0 is an **M** instruction, slot 1 is an **I** instruction and slot 2 is a **B** instruction.

The structures defined by these templates are governed by the availability of execution units in the processor. If a set of instructions require the use of a particular type of execution unit, and there are $n$ units of this type available, then $n$ instructions from this set may execute in parallel. In this case, a template will define that a bundle may consist of up to $n$ of these instructions.

Grouping of bundles further allows instructions to execute in parallel. When a set of bundles are grouped together, the processor assumes that the instructions in these bundles have no dependencies and may be executed in any sequence, be it sequential or parallel.

The IA-64 architecture specifies a large number of registers. The general integer and floating point register files both have 128 registers. A potential problem of having a large number of registers is that a large state may need to be stored in memory on each context switch, which is a slow process relative to CPU speed. How the large set of registers is managed in the IA-64 processor is discussed in Section 2.3.2.

Programming for the IA-64 architecture differs significantly from x86 programming. Still, the IA-64 architecture allows x86 programs to run in an *IA-32 compatibility mode*. This mode, however, does not take advantage of some of the IA-64's performance critical features. For example, explicitly parallel programming for the IA-64 architecture is not possible given the x86 ISA. Thus, the IA-32 compatibility mode is provided mostly for backwards compatibility towards legacy programs.

### 2.3.2 Registers

In RISC processors, *register conventions* are needed to dictate how OSs are to use and manage the registers. This is also adopted by the IA-64 architecture. The set of registers in an IA-64 CPU is divided into several *register files*. The *General Register File* consists of 128 64-bit registers, R0 through R127. These are general purpose integer registers. They may be used freely, only constrained by conventions, except for R0, which is always zero. Convention states that they are divided into two classes, *preserved* and *scratch*. Preserved registers must be explicitly managed by any process that uses them, that is, on any function call, the process must itself store the necessary registers in memory. Scratch registers, however, are at a process' disposition to use at convenience.

The IA-64 architecture provides the programmer with the illusion of an unlimited register stack through a mechanism called the *Register Stack Engine (RSE)*. It makes utlization of the general register file easier for programmers. It presents the programmer with a logical *register stack frame*, which consists of a range of allocated registers. Whenever a branch is executed, the allocated register stack frame is stored on the stack, and upon return from the branch the state of the allocated registers is restored from the stack. Registers R32 through R127 are used by the RSE on Itanium, and any number of registers within this range may be allocated to a register stack frame.

The register stack stores and retrieves register frames in main memory in a LIFO (Last In, First Out) manner. Register stack frames stored in memory can also be accessed directly in memory. The BSP application register points to where the beginning of the current register stack frame would be in memory if it were to be put on the stack.

As in the x86 architecture, floating point registers may be managed lazily in the IA-64 architecture. This may be achieved by disabling access to these registers, which will cause an exception when a task tries to access them.

### 2.3.3 Hardware Abstraction

The IA-64 architecture abstracts away small differences in different hardware implementations through three interfaces called *Extensible Firmware Interface (EFI)*, *Processor Abstraction Layer (PAL)* and *System Abstraction Layer (SAL)*.

The EFI maintains an interface for booting an OS kernel called *EFI boot services*. This includes services for allocating and freeing memory. These services are accessed in physical addressing mode only. This is used by OSs for booting.

The routines for the EFI, PAL and SAL interfaces are accessed through a table called the *EFI system table*, which points to the individual routines, in their respective memory spaces. An OS accesses these routines by looking up in the table and executing the routines directly. This table is defined by convention and must be followed by the OSs.

### 2.3.4 Interrupts

Interrupts are similar in concept in the IA-64 architecture, to the x86 architecture. In the IA-64 architecture, the legacy of the PIC is no longer necessary to maintain. It is replaced with the *Streamlined Advanced Programmable Interrupt Controller (SAPIC)*, which allows faster delivery of interrupts and more CPUs in an SMP system. The IA-64 keeps interrupt handlers in a table called the *Interruption Vector Table (IVT)*. The starting address of this table is stored in the IVA register.

Similarly to the x86, the IA-64 architecture provides mechanisms for keeping time.

- The EFI provides an interface for reading wall clock time, similar to the x86's RTC, although abstracted.
- The *Interval Time Counter (ITC)* counts CPU cycles, similar to the x86's TSC.
- The *Periodic Timer Interrupt* allows the delivery of interrupts at a given frequency, similar to the x86's PIT.

### 2.3.5 Memory Management

The IA-64 CPU uses a set of dedicated registers called *region registers* to make the TLB more efficient. They divide the address space into logically separate *regions*. Each entry in the TLB is tagged with a *region id*. On a TLB lookup, the entry of a region register is compared to the region id of the TLB entries. If the region id differs from the value in the region register, the TLB entry is invalid for this particular region. Thus, unlike in the x86 architecture, the TLB need not be flushed on address space changes, as invalid TLB entries are logically marked as invalid. The IA-64 TLB stores, in addition to address translations, protection information of the PTEs it stores. This information can be stored both as a part of the mapping entry in the TLB and in a separate *Protection Key Register* file. Protection information in the Protection Key Register file is accessed through the reference of a *protection key*. Every TLB entry has a protection key field.

The IA-64 architecture specifies two TLBs—one for data and one for instructions. These TLBs are both partitioned into a *cache* part and a *register* part. The cache TLB entries are hardware managed, while the register TLB entries are managed directly by the OS just like ordinary registers.

Whenever a translation entry from virtual address to physical address does not exist in the TLB, the entry must be found in the page table. This can be a costly affair, as discussed in Section 2.1.3. The IA-64 architecture allows entries to be filled into the TLB more efficiently by using the *Virtual Hashed Page Table (VHPT)*. This feature is also referred to as the *hardware walker*.

For the hardware walker to function, it requires a logical second page table, formatted in a special way. There are two different supported page table formats, namely *short-format* and *long-format*. Using the long format, the OS uses a hash table as its page table, while using the short format, the OS must have a *virtually-mapped linear page table*. *Linear* means that the keys in the table are linear, so unlike page tables with several levels, which require one lookup per level, this table only requires one lookup. The table is mapped in virtual memory, residing in the virtual address space in each region in which it is enabled, thus it can also be mapped by the TLB.

When a virtual address, $va$, is to be translated to a physical address, $pa$, first the TLB is referenced. Given a miss in the TLB for $va$, in the traditional manner, a page fault would be raised, and the OS would walk the tables to find the PTE that translates $va$ to $pa$—a costly operation. With the VHPT walker enabled, however, a miss in the TLB does not raise a page fault. Instead, the address of the PTE for $va$, $va'$, is calculated by the walker. Since the VHPT resides in virtual address space, and given high spatial locality, the address of the PTE is with a high probability already in the TLB. Thus, only one extra memory lookup is needed. If the translation of $va'$ does not exist in the TLB, a *VHPT Translation Fault* is raised, and the OS may continue to look up in the page table in the traditional manner.

An OS may also choose to skip the page table completely and avoid the protection mechanisms provided by paging and access memory directly, using physical addresses. The IA-64 CPU controls whether addresses are physical or virtual in the PSR register, using the DT flag (see Appendix A.2).

### 2.3.6 The Performance Monitoring Unit

The IA-64 architecture offers, with the PMU, a standardized and documented facility for monitoring processor performance. The PMU is controlled by a set of registers that can be programmed to monitor certain performance aspects of the CPU. Two register files, consisting of the *Performance Monitor Data (PMD)* and *Performance Monitor Configuration (PMC)* registers, respectively, compose the PMU. The former register file captures performance data, such as number of processor cycles or instructions executed, while the latter controls what performance data the former captures.

The IA-64 architecture specifies eight PMC registers and four PMD registers and their semantics, though 248 PMC and 252 PMD logical registers are reserved for implementation at different IA-64 architectures' discretion. For example, the Itanium processor extends the PMU to include monitoring capabilities of TLB misses and branching characteristics, among others. The Itanium PMU allows a user to get the number of cycles or instructions used or the number of a particular instruction executed by a program, whether in user or kernel space or both; and to help finding hotspots, the location of where cache or TLB misses occured in code, can be captured.

## 2.4 Linux

Linux was initially intended by Linus Thorvalds as a surrogate for the Unix OS on the x86 processor. However, its liberal licensing and the emergence of the Internet attracted many developers and allowed Linux to soon be ported to additional architectures such as MIPS, SPARC, Alpha, PA-RISC and, in 2000, IA-64. An important factor to the growth and popularity of Linux has been the availability of the free and open source GNU libraries and compiler. Comprehensive descriptions of the Linux kernels for the x86 and IA-64 architectures are given by the books "Understanding the Linux Kernel" by Daniel P. Bovet & Marco Cesati [BC02] and "IA-64 Linux Kernel - Design and Implementation" by David Mosberger & Stéphane Eranian [ME02], respectively.

Most modern OSs employ multitasking, that is, they allow multiple programs, or processes, to run simultaneously. For the OS to provide the illusion of running different processes simultaneously, it lets each process only to run for a very short period at a time, before switching to the next process. When switching to a new process, the kernel stores the register contents—the process' hardware context—in memory, then brings in another process by loading its hardware context from memory, and lets it run for a while, and so on. How the hardware context is managed when switching between tasks, is discussed in Section 2.4.3.

The timer is an important mechanism in an OS. It is, among others, used to interrupt a running process when it has used its time share, and pass control to the next process. The timing mechanisms in Linux are described in Section 2.4.2. The timer signals the kernel when it is time to switch processes through an interrupt. How the kernel handles interrupts such as these is described in Section 2.4.1.

Processes can have different privileges in terms of which memory segments or pages they are allowed to read or write and which instructions they are allowed to execute. The privilege levels are usually interpreted as protection rings, as shown in Figure 2.2, . Most OSs run in ring 0 while applications run in ring 3 . In Unix terms these are called *kernel mode* and *user mode*, respectively.

As Unix, Linux has kernel and user processes. While kernel processes run exclusively in kernel mode, user processes run mostly in user mode but may also enter kernel mode through *system calls*. In order for user processes to do operations that require higher privileges, it has to enter the kernel through a system call. System calls are further discussed in Section 2.4.4.

Each process—that is, each address space—in Linux has its own page table, which is kept in physical memory. Linux keeps the page table tree in three levels, with a *Page Global Directory*, a *Page Middle Directory* and a *PTE Directory*. A page in Linux can be kept in main memory or moved to the swap space on a harddisk. The kernel tags a page which has been moved to the swap space as *not present*, as opposed to the case in which it resides in main memory, in which it is tagged *present*. Also, a page has information about read and write permission and the *dirty-bit* which signifies whether a page has been written to. These attributes are all kept in the page's PTE.

The traditional approach for finding performance hotspots in Linux is to instrument the code. Instrumenting, however, can have an influence on the execution of Linux and return results that are unrealistic, also certain low level functionality is difficult to monitor using software. The IA-64 port of Linux

includes software for utilizing the PMU. This is further discussed in Section 2.4.5.

### 2.4.1 Interrupts

The kernel manages many devices, such as harddrives, keyboards and network interface cards. Whenever such a device sends an interrupt to the processor, the kernel needs to readily respond. If any user space process is running, the kernel must suspend this process' operation and start executing the ISR corresponding to the interrupt. The operations of transferring control from a user process to the kernel and vice versa are called *kernel entry* and *kernel exit*, respectively. Kernel entry and exits are described in Section **??**.

Some actions are time critical and have to be handled immediately, such as acknowledging an interrupt coming from the interrupt controller. To ensure that nothing interferes with the execution of a critical action, maskable interrupts are disabled during execution. Noncritical actions, such as reading keyboard input, however, are executed without disabling maskable interrupts. This allows more critical interrupts' ISRs to execute while less critical ISRs are interrupted and deferred until later. Some actions may even be deliberately deferred until later. On such interrupts, the action is only initiated in the ISR, and further execution continues outside the ISR at a later, possibly more appropriate, time. A high level description of how Linux handles interrupts is shown in Figure 2.4.

Actions that are deferred until later are based on a kernel mechanism called *softirqs*. These are used among others to deliver packets to and from network cards. Softirqs are executed dynamically by a kernel thread called *ksoftirqd*, which is short for *Kernel Softirq Daemon*. The Kernel Softirq Daemon runs in parallel with other kernel threads all the while during the kernel's lifetime.

*Tasklets* are another type of procedures registered to an IRQ. An important difference between tasklets and softirqs is that a tasklet can be defined dynamically, i.e. during execution of the kernel. Also, tasklets are serialized in that two instances of a tasklet can not be executed simultaneously. Two instances of a softirq, however, may execute simultaneously on several CPUs.

### 2.4.2 Time

OSs keep track of time for a number of reasons:

- In order to provide the user with the convenience of a time source for use in different applications.
- In order to allow applications to perform actions at a specific moment.
- In order to allow itself to schedule the execution of processes in a time sharing environment.

A *timer* delivers a timer event at a given moment similarly to an alarm clock. This event is captured as an interrupt by the OS, which the OS then handles with the corresponding ISR. The OS can also generate timer events itself, which can be useful to generate alarms for user applications. This can be implemented by the OS as a list of alarms to be generated which is referenced at a given frequency. Linux implements this in the *dynamic timer* mechanism.

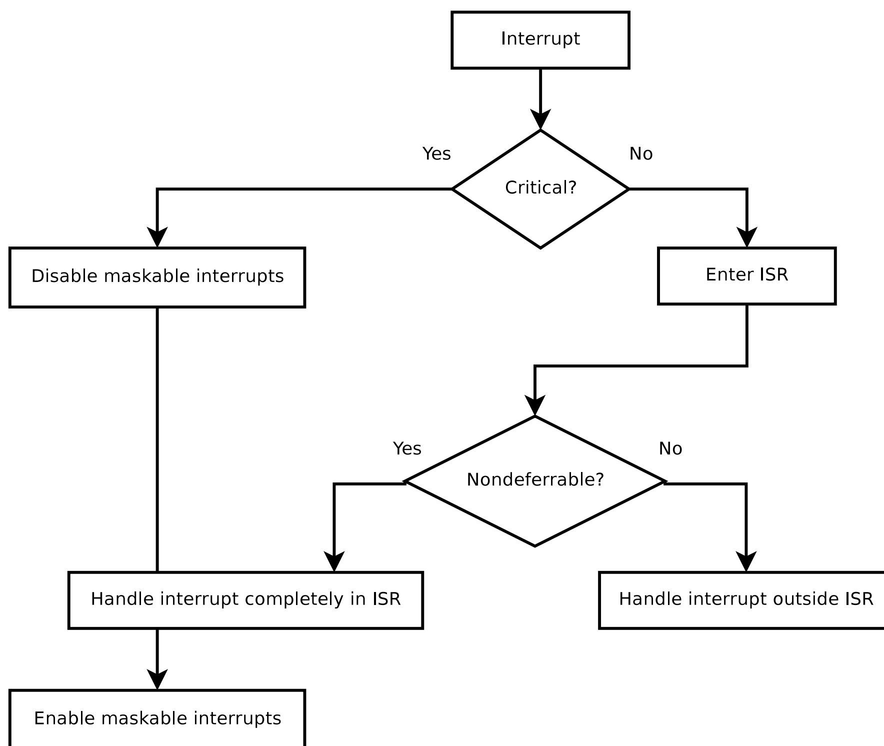


Figure 2.4: Flowchart showing how interrupts are handled in Linux.

### x86 Implementation

In order to keep track of wall clock time, Linux uses the PIT to trigger an interrupt each 10 ms. At each interrupt the kernel updates the system clock, which gives the clock a resolution of 10 ms. To give the clock higher resolution the kernel estimates time intervals smaller than 10 ms using the TSC. The PIT interrupt is also used to deliver timer events to applications.

### IA-64 Implementation

Similar to Linux/x86, Linux/ia64 uses a periodic interrupt to update the system clock, in this case generated by the Periodic Timer Interrupt, and higher resolution time intervals are in this case estimated by the ITC.

## 2.4.3 Processes

Linux keeps track of what a process is doing in a data structure called a *process descriptor*. Each time a process is descheduled process state is stored in its process descriptor, and each time it is scheduled the process state is loaded from the process descriptor again. The scheduler uses information, such as process priority, stored in the process descriptor, to schedule the execution of processes. Also, some hardware context, depending on the computer architecture, are kept in this data structure. When accounting for other data structures, such as the current address space and file descriptors, which the process descriptor also keeps pointers to, the data structure becomes quite complex.

When switching from one process to another, the Kernel Memory Stack is used to store some of the hardware context. For each process, the process descriptor and a Kernel Memory Stack are stored in two consecutive pages. The previous process pushes some of its hardware context onto its Kernel Memory Stack before being descheduled. Then, the next scheduled process pops its hardware context from its Kernel Memory Stack to return to the state the process was before it was previously descheduled. In Linux/x86, the `esp` register points to the top of the Kernel Memory Stack. Linux/ia64, in addition, needs to manage the Kernel Register Frame Stack in the RSE.

A list of processes is maintained as a doubly linked list, in which each process descriptor has pointers to the previous and next process descriptors in the list. Also, process descriptors keep pointers to the process descriptors of parent and child processes.

Linux uses the timer to schedule the execution of tasks. When a timer interrupts the currently running process, as it has depleted its dedicated time slot, control is taken from the running process and given to the kernel. The kernel then schedules another process for dispatch. As control is handed over from one process to another, one process is likely to have its hardware context changed by another process. Thus, when one process is descheduled, the kernel stores some of this context into memory, ensuring that the context is restored when the process is scheduled again. The hardware context of a thread is managed in a set of three data structures, `pt_regs`, `switch_stack` and `thread_struct`.

`pt_regs` is used to store the scratch registers and is therefore the most used data structure. It manages state when control is taken from a process and given to the kernel, through, for instance, a system call or a device interrupt. As

register convention dictates, not all hardware context is necessary to manage. Of the General Register File, preserved registers are by convention managed by processes themselves, and anyone who implements a user program should know that these registers may be changed by other processes when control is given to them. Therefore, only scratch registers are necessary to manage, since processes are not expected to manage these themselves. Also, since, with a few exceptions, the kernel does not need floating point registers, these registers are also not managed by the kernel.

The `switch_stack` data structure is used to manage state when control is transferred from one user process to another, which is primarily the preserved registers. The `thread_struct` is used to manage registers that are changed more rarely and can be set to be used only when the state of these registers changes.

Apart from the three data structures for managing hardware context, higher level information on each thread is managed in data structures called *task structures*. The kernel maintains a list of tasks in a doubly linked list, which is constructed by tasks' task structures pointing to two other tasks' task structures—the next and previous tasks in the scheduling queue. Other data maintained in task structures include process IDs, address space definitions and scheduling and file system information.

The first process which is started in Linux is the *Init* process. It initiates the system and starts most of the services and the shell. All new processes are *forked* from existing processes, making a tree of all processes. All processes stem from the Init process.

#### x86 Implementation

Even though the x86 architecture allows for automatic hardware context switching between processes (see Section 2.2), Linux does most of the context switching manually, thus it can not use the far `jmp` or `call` instructions to execute system calls. By manually storing the register contents into memory through `mov` instructions, it achieves the same effect as if using the facilities of the x86 processor. However, when switching from user mode to kernel mode, the x86 architecture forces the OS to fetch the address of the Kernel Memory Stack from the TSS. Although Linux/x86 employs manual task switching, it is able to manage the floating point and XMM registers lazily by manually setting the `TS` flag in the `cr0` register.

### 2.4.4 Kernel Entry & Exit

System calls let user processes do operations that require ring zero privileges. The user process executes a special instruction, which further calls a particular system call procedure, which is executed in kernel mode. After this subroutine is finished, control is given back to the user process. Examples of system calls are filesystem and process management operations.

System calls are not regular continuations of the executing process like ordinary function calls are. Rather, a system call involves a change of, among other things, privilege level, address space and a change from a User Mode Stack to the Kernel Memory Stack. Also, user processes do not have access to kernel address space because of protection. The passing of parameters to the system call, thus, is done differently from that in a function call.

Linux often uses registers to pass parameters, though this limits the number of parameters, as CPUs often have a limited number of registers to use for this purpose. If registers can not hold enough data for passing parameters, pointers to memory are used as parameters. The kernel accesses these parameters through the user process' address space, through which also results of the system call may be returned. Before doing this, however, the kernel needs to check the validity of the address.

Linux offers an interface with a set of routines for accessing user process address space. By using this interface, the kernel can verify that the address referenced is

1. within user address space—by checking if the address is within the active segment,
2. valid—otherwise, a page fault exception is raised, and recovery actions are taken with the page fault handler,
3. accessible, that is, readable or writable.

#### x86 Implementation

Linux/x86 executes a system call by issuing an interrupt with the `int` instruction (see Appendix B.1), with an immediate vector of 0x80. The IDT entry at position 0x80 points to the procedure which is to be executed, which further calls the specific system call. Each system call has a specific number, and the system call to be executed is selected by the user mode process by putting that number in the EAX register.

#### IA-64 Implementation

Unlike Linux/x86, Linux/ia64 uses the `break` instruction to perform a system call. This is, however, somewhat similar to the x86. The `break` instruction triggers an exception, which is caught in the IVT, and a system call corresponding to the immediate value given by the `break` instruction, is called. Since the IA-64 architecture has a richer set of registers, it allows for several more parameters to be passed in registers than the x86 architecture does. Still, sometimes parameters are too big or too many to be passed in registers, and user process address space must be utilized.

Managing the state of the RSE makes kernel entry and exit more complex. The kernel first takes control of the RSE's mode of operation. Then, to protect the current register stack frame, which is the user level register stack frame of the process which is interrupted, the value of BSP is increased such that it points to the free area after the current register stack frame. The user process' register stack frame must be saved in the user backing store and a new register stack frame is then allocated for the interrupt or exception handler to use.

### 2.4.5 Performance Monitoring

There is little support for utilizing the performance monitoring facilities of some x86 CPU models in Linux. x86 performance monitoring facilities are unstandardized and poorly documented, which makes it hard to make a standardized

interface. Programmers and users are usually left with the option of instrumenting the program, which can take some work and may influence the program's execution, which further may influence the results of the measurements. Some kernel patches do exist, however, which allow utilization of the x86 performance monitoring facilities, such as Mikael Pettersson's *Perfctr* [Pet05].

Linux/ia64 includes support for performance monitoring using the IA-64 PMU, straight from the main source tree. Stéphane Eranian's *Perfmon* [Era05] is integrated into the Linux source, and user space tools such as the *Pfmon* tool and the *Libpfm* library exist to make interfacing with the PMU easier.

## 2.5 Virtualization

The term *Virtual Machine (VM)* spans a range of software and hardware mechanisms, from bytecode interpreters to virtual computer architectures. The term signifies what it is—a machine that is virtual in the sense that it imitates real machines.

The benefits of VMs are many. VMs can be run as a software mechanism on top of a conventional OS and allow more flexible use of the computer and OS's resources. The resources can be divided into separate execution environments, called *domains* by Paul Barham et al. [B$^+$03] or in many cases, simply VMs. Michael Rosenblum [Ros04] gives three essential concepts that are valuable in virtualization.

- If the VM maintains *isolation* between domains, execution in one domain should not adversely affect execution in another domain. Also, one domain should not be able to inspect the state of another domain.
- Since the VM *encapsulates* the domains, the VM has access to the domains' states and can control and monitor their execution. If necessary, a program running in a domain should not be aware that the underlying execution environment is virtual.
- Also, VMs that maintain *software compatibility* allow the programs written for the same VM architecture be executed on VMs running on different hardware architectures.

Different methods for virtualizing a computer architecture exist. One method involves *interpreting* each instruction, and then emulating the computer architecture by simulating what will happen if this instruction is executing on a real computer architecture. Another method involves running the guest software directly on the hardware, while executing sections of the software in a monitoring mechanism. The advantage of the former method is that every instruction may be verified, logged and monitored. The drawback, however, is that a severe performance penalty incurs as each instruction is interpreted, translated and then executed. In simulating architectures, this method is often referred to as *execution-driven* simulation. The advantage of the latter method is that the guest software runs much faster, as there is less overhead from interpreting instructions. This method can also be used in simulating architectures, by executing instructions on a processor and logging the results, the execution can be monitored. This latter method is often referred to as *trace-driven* simulation.

Examples of simulators are *Bochs* [boc05], *Simics* [sim05a] and *SimpleScalar* [sim05b]. These are not as concerned with performance as they are with providing a tool for simulation. Simply interpreting instructions has a high performance impact, but usefulness is regained in the possibilities of monitoring each instruction. This is useful for design of low level software, such as OSs, and for simulating new computer architectures. Also, software compatibility is provided in the sense that the same software may be run across different hardware architectures.

Other VMs are more concerned with performance, as they aim to provide the user with a platform for running software for more general practical applications. Various techniques are used in order to gain performance. The methods Microsoft's *Virtual PC* [msv05] and *QEMU* [qem05] use, involve translating blocks of instructions and running them natively, allowing blocks to be rerun without retranslating. This method gives a performance increase compared to pure interpreting, and may also still allow software to be run across real architectures. The method used by *VMWare* [vmw05] to virtualize an x86 processor involves running some code directly on hardware and some interpreted, giving better performance than the former methods. The rationale behind this method is further discussed in Section 2.5.1. One of the drawbacks with VMWare's solution is that it only allows x86 software to run on x86 machines.

### 2.5.1 Virtual Machine Monitors

In virtualizing computer architectures, the mechanism that provides the user with VMs is often referred to as a *Virtual Machine Monitor (VMM)* or a *hypervisor* [4]. Figure 2.5 illustrates a traditional VMM running in a host OS. The host OS runs directly on physical hardware, while the VMM runs as a process inside the host OS's execution environment, along with other user processes. Two guest OS are hosted by the VMM and run on the VMM's virtualized hardware.

OSs are usually designed to run directly on a particular computer architecture, and thus they expect the computer to behave in a certain way. As the design of an OS has many degrees of freedom, if the VMM fails to virtualize the processor completely, in every detail, the OS might fail in execution. Most traditional VMMs aim to let any OS run unmodified on its hypervisor, that is, to completely virtualize the underlying architecture. While aiming to give its hosted OSs a virtual architecture logically equal to the OSs native architecture, VMMs also try to give good performance characteristics.

As an x86 OS kernel protects itself from user processes through the use of ring protection, as do an x86 hypervisor require the notion of protection in order to be able protect sensitive system state from alteration and inspection from domains. Thus, whenever a domain's execution is performed directly on the processor—that is, uninterpreted—the guest OS must run in a less privileged ring than zero. When an OS is run in a ring other than zero, the processor behaves differently from what the OS expects, and thus the execution may fail. This is the reason why VMWare interprets and emulates code which belongs to the guest OS kernel. Xen's approach to this problem is discussed in Section 2.6.

[4]The term *supervisor* is sometimes used when referring to the OS mechanism, which runs in privileged mode and *supervises* other software processes. The term *hypervisor* reflects the fact that it is a mechanism that lies under (or closer to hardware than) the supervisor and effectively supervises the supervisors.

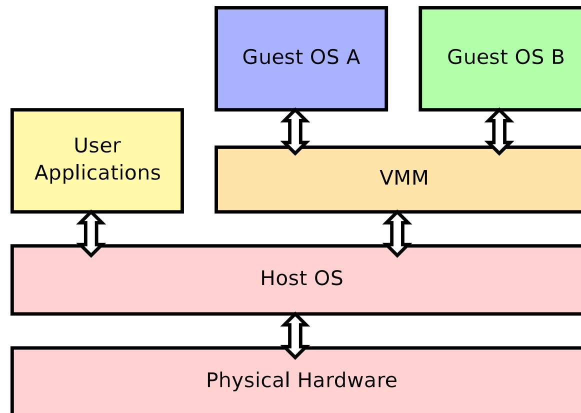


Figure 2.5: Illustration of a VMM hosted on a host OS, running two guest OSs.

## 2.6 Xen Virtual Machine Monitor

The Xen VMM addresses the main problem with traditional x86 VMMs, namely efficient utilization of processor resources. Most OSs require to run in ring zero and the processor to behave in a certain way. They thus incur the performance penalties of translation of OS code. Xen attacks this problem at the roots by modifying the OS so that it behaves differently. Instead of doing this during execution, the changes are made in the source code before compilation and execution. This has been inspired by some of the work in *Denali* by Andrew Whitaker et al. [W+02].

[B+03] uses the term *guest OS* to refer to an OS which is hosted in a VMM and the term *domain* to refer to the virtual machine in which the guest OS is running, i.e. on a VMM there are one or more domains, and in each domain a guest OS is running. Xen does not provide a fully virtualized platform on which guest OSs may execute. Instead, the guest OS needs to be modified to run on this platform—the hardware is effectively *para-virtualized* [W+02]. The modifications do not, however, affect the guest OS's interface to the user applications, thus they need not be modified. A few OSs have been modified in order to run on the Xen/x86 architecture, such as NetBSD and Linux, with Linux being the most mature. Since Microsoft Windows' source code is closed for the public, a port of Windows relies on the willingness and efforts of Microsoft. Para-virtualization is further discussed in Section 2.6.1.

The components of a complete Xen virtualization infrastructure include a Xen hypervisor, a *privileged domain*, one or more *unprivileged domains* and some user-space *control tools*, as shown in Figure 2.6. This contrasts to traditional VMMs (see Figure 2.5) in that the VMM runs as hosting mechanism, in full control of the hardware. As the names suggests, the privileged domain has higher privileges than the unprivileged domain. Control software running in the privileged domain has the privileges to control the operation of the hypervisor, including starting, suspending and shutting down unprivileged domains. The privileged domain is commonly referred to as *domain 0*.

The usual mechanisms for interrupts from devices are replaced with an asyn-

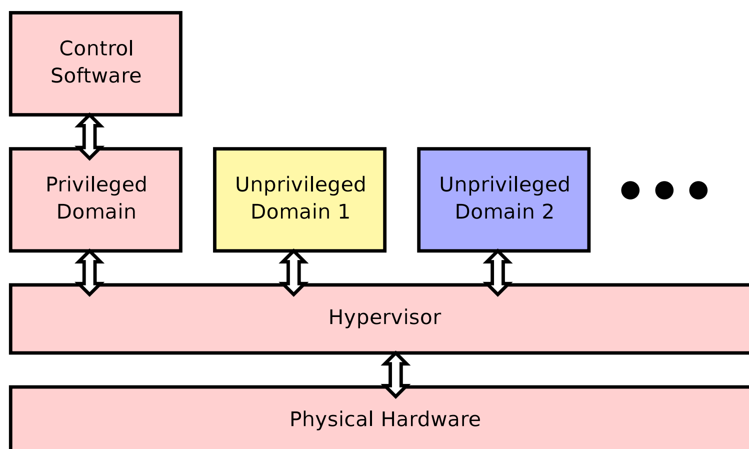


Figure 2.6: The Xen hypervisor, hosting several domains.

chronous event mechanism provided by the Xen hypervisor. CPU resources are distributed dynamically and fully utilized, while main memory resources are partitioned among domains. Block and network interface drivers must also be virtualized in order to divide these resources among domains. The virtualization of these and other devices is described in Section 2.6.2.

In addition to managing its own memory, the hypervisor needs to manage the domains' memory usage to make sure that they do not access each other's memory spaces. This is necessary in order to maintain isolation. Memory management in Xen is further discussed in Section 2.6.3.

The execution of domains is scheduled, similarly to the scheduling of processes in Linux. Xen provides three different scheduling algorithms, *Borrowed Virtual Time (BVT)* [DC99], *Atropos* and *Round Robin*, behind a unified API, making it easy to add different schedulers. The BVT scheduler is discussed in Section 2.6.4.

### 2.6.1 Paravirtualization

In Xen guest OSs, instructions that in some way may reveal that the OS is running on a VMM are replaced into functional procedures which emulate the original operation, statically in the source code. The guest OS is modified such that when it originally would execute privileged instructions, instead it makes a *hypercall* into the hypervisor, in which the operations are executed. This is similar to how a user process makes a system call into the kernel to execute privileged operations. System state which is protected is virtualized through the hypervisor. For instance, the hypervisor keeps a *virtual CPU* for each domain. When a guest OS would normally write to a protected register, instead, the hypervisor writes the value to the corresponding virtual register in the virtual CPU.

### 2.6.2 Event Handling

When sharing a single device resource such as, for instance, a harddrive, two guest OSs using the device simultaneously can lead to problems. One malicious or faulty domain can adversely affect the execution of other domains. Also, if several domains rely on a single device, if the device fails due to hardware failure, all the domains will be affected. If several domains are to share a device, it is preferable, in order to enforce isolation and maintain stability, to virtualize devices' resources. In order to maintain isolation, one domain does not know what the other is doing, and if a resource is to be shared, the sharing must be fair. Similarly, if a device is to be managed only by one domain, the hypervisor must ensure that only that domain accesses that device. Thus, the hypervisor manages access to devices. Keir Fraser et al. [F$^+$04b, F$^+$04a] discuss the Xen I/O model in detail.

The hypervisor uses an event mechanism to manage access. Instead of letting the domains handle interrupts directly, they are handled within the hypervisor. This way, the hypervisor can schedule the execution of the ISR of the domain which manages the device. After recieving an interrupt, the hypervisor schedules the execution of the corresponding domain and issues to it an asynchronous event notification. Event notifications are delivered to domains through *event channels*. The hypervisor communicates these events with a domain through a shared memory page. This approach lets guest OSs use their own drivers unmodified to control devices.

As well as letting domains manage devices directly, a Xen managed system as a whole can benefit from devices being virtualized. This allows devices to be shared while ensuring isolation and has other potential benefits. Virtualized devices are managed in *isolated driver domains (IDD)*. Transfers between an IDD and a guest OS are serviced by *I/O descriptor rings*, in which transfers are asynchronously scheduled. IDDs and guest OSs are notified of queued transfer descriptors through event channels.

Block devices, such as harddrives, are managed using IDDs, as illustrated in Figure 2.7. Domain 0 has direct access to a harddrive through the hypervisor's event mechanism, while providing an abstraction for other domains through an IDD, through which they have access to virtual block devices. A driver that domain 0 uses to provide other domains with a virtual block or network device is called a *backend driver*, while a driver that a domain uses to control a virtual block or network device is called a *frontend driver*. Domain 0 uses the same driver as the native driver of the guest OS that instantiates domain 0 uses, to control the device physically.

### 2.6.3 Memory Management

Since several domains share the same memory, care has to be taken in order to maintain isolation. The hypervisor must ensure that two unprivileged domains do not access the same memory area. Each page or directory table update has to be validated by the hypervisor in order to ensure that domains only manipulate their own tables. The domain may batch these operations to make sequential updates more efficient. Segmentation is virtualized in a similar manner. The hypervisor makes sure that also domains' segments do not map memory areas that overlap or that are invalid in any other way.

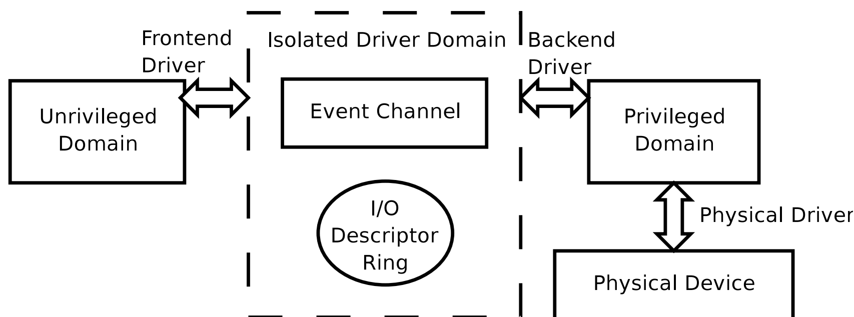


Figure 2.7: Device virtualization in Xen.

The x86 TLB poses some overhead to Xen/x86. Since the x86 TLB is hardware managed, every switch between domains requires a flush of the entire TLB. Flushing the TLB makes all previous entries of a given address space invalid, and thus they have to be reentered when control is given back to the address space. Different domains have different virtual memory spaces and thus different page tables. Switching between page tables invalidates previous entries in the TLB. Had the TLB been software managed and tagged, only entries used would need to be invalidated, eliminating the need to flush the TLB.

Domains are allocated physical memory by the hypervisor, which is not necessarily contigous. This memory does not map directly to the machine's physical memory and is in that sense *pseudo-physical*. The translation from physical to pseudo-physical is not transparent, and guest OSs need to translate addresses from pseudo-physical to physical themselves. The hypervisor provides a *machine-to-physical* and a *physical-to-machine*, which map pseudo-physical to physical addresses and vice versa.

### 2.6.4 Time

Guest OSs are provided with different mechanisms to manage time that replace their native counterparts. Wall time is kept by the hypervisor and can be read by a domain. Also, virtual time is kept and only progresses during a domain's execution, which allows correct scheduling of domains' internal tasks.

Xen needs to shuffle the execution of domains just as native OSs need to shuffle the execution of tasks in order to maintain the illusion of simultaneous execution. The BVT scheduling algorithm is presented by Kenneth J. Duda [DC99]. It allows a domain to be dispatched with low latency. This allows the domain to be quickly dispatched to respond to events, such as packets arriving on the network. BVT operates with the concept of *Effective Virtual Time (EVT)*. Whenever the scheduler selects a new domain to run, it picks the domain with the lowest EVT.

# Chapter 3

# Analysis

Some of the problems in virtualizing the x86 processor have parallels in the IA-64 architecture. Therefore, lessons learned from previous work in x86 virtualization are of value to also to IA-64 virtualization. An analysis of x86 virtualization can thus be helpful. The main problems of virtualization in Intel architectures are discussed in Section 3.1.

Implementing an IA-64 port of Xen is most efficiently done by trying to reuse as much code as possible. Many years of work have been put in Gnu General Public License (GPL) licensed program code, and this code is freely available to use under the terms of the GPL. Particularly, Xen/ia64 has the possibility of leveraging code from both Linux/ia64 and the Xen hypervisor, both being GPL licensed. A discussion of the Xen hypervisor and how this relates to IA-64 follows in Section 3.2.

## 3.1 Virtualization on Intel Architecture

The x86 architecture is hard to virtualize efficiently. It was not designed with virtualization in mind, as is, for instance, the Power5 architecture [K+04] of IBM. Since x86 is a very prevalent computer architecture in many areas of computing, virtualization has not been applied widely in these areas before. Rich Uhlig et al. [U+05] give summary of some of the challenges in virtualizing the x86 architecture.

As previously discussed, a guest OS may not run in ring zero, because the hypervisor must be protected in order to enforce isolation. VMMs therefore let the guest OS run in less privileged rings. The method of running an OS in another ring than that which it was intended to run in, is called *ring compression* or *depriviligation* [U+05]. *Ring aliasing* [U+05] refers to some of the problems that may arise when using ring compression. Some problems that may arise when running an OS deprivileged on Intel architectures, are:

1. *Faulting privileged instructions*—when running in less privileged rings, some of the OS's native instructions may fail and raise exceptions. These operations are referred to as *privileged* operations.

2. *Faulting access to protected state*—some registers are protected from being written to and possibly read from. A regular instruction trying to access

such a register will also cause an exception to be raised, if running with a CPL other than zero.

3. *Nonfaulting privileged instructions*—in some cases, an OS's instruction may execute without failing and rising an exception, although the result of the operation is morphed due to it being executed in a less privileged ring. Such operations are referred to as *privilege-sensitive* [MC04] operations.

4. *Nonfaulting access to protected state*—some processor state can be read unprivileged, while afterthought has shown that this state should have been protected. This state can contain data which is only correct if accessed when CPL is zero. Nonfaulting write access to protected state can be considered a bug in the architecture.

5. *Privilege level leakage*—refers to the problem of the processor revealing to the guest OS that it is running deprivileged.

6. *Protected state leakage*—refers to the leakage of other state that may reveal to the OS that it is not running natively.

In the following context of virtualization, such instructions and operations which morph the execution environment of an OS, are collectively referred to as *illegal* instructions or operations.

Problems 1 and 2 can be solved dynamically by the hypervisor during runtime. Since whenever a privileged operation is executed in a ring other than zero, a *Privileged Operation Fault* is raised, the hypervisor can capture this exception and execute an ISR which emulates the intended operation, and then gives control back to the OS. An efficiency penalty, however, is incurred, as emulating these operations usually takes longer than executing them directly on the processor. Problems 3 through 6 are handled less dynamically. Privilege-sensitive operations can not be observed by the hypervisor unless they are interpreted.

There are two approaches to virtualizing the guest OSs' underlying platform. One method involves replacing privileged and illegal operations with hypercalls, while another method involves replacing illegal operations with operations that the hypervisor will trap and handle, as discussed in Section 4.1.2. The latter method, and also to some extent, the first method, involves virtualizing the CPU state. This is discussed in Section 4.3.

The various stages in the life of an OS, from programming to execution, are illustrated in Figure 3.1. For each stage the phase of the OS, the technology used to address the problems of virtualization and the technique the technology uses to address the problems. The OS starts out in the programming phase. In this phase the OS can be modified to run on a virtual execution environment provided by the hypervisor. This is a very efficient approach since no instructions need to be interpreted and since modifications can be done more easily with efficiency being the goal, since programming is done in a high-level language. Modifications at this stage, however, are more static in that the programming phase is absolutely necessary for para-virtualization.

In the compilation phase, the compiler can make decisions to make machine code more virtualization-friendly. *L4Ka* (see Chapter 6) modifies the assembly instructions generated by the compiler so that illegal instructions are replaced

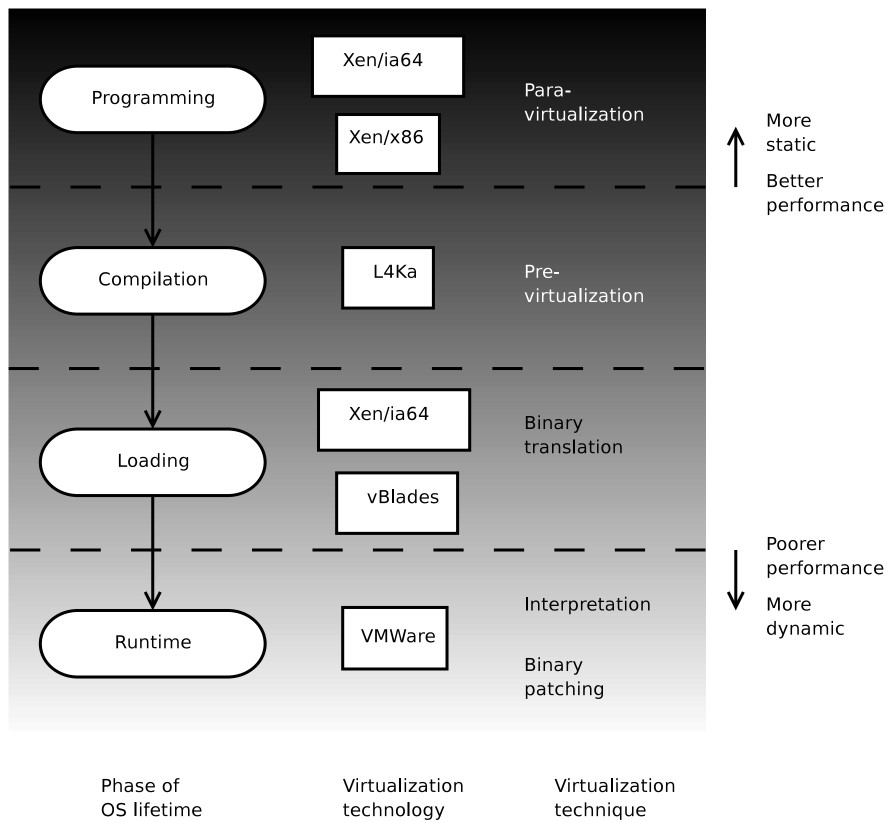


Figure 3.1: Different methods of virtualization.

with routines that emulate the intended instructions. Also, operations are simulated on a mechanism to reveal illegal memory operations, which are also replaced. This is called *pre-virtualization*.

*VBlades* (see Chapter 6) takes a similar approach by replacing illegal instructions and operations with faulting instructions that are trapped by the hypervisor and emulated (see Section 4.1.2). This usually does not achieve as good performance as para-virtualization since the emulation routines are slower than the instructions that are being replaced.

The most dynamic approach is taken by VMWare. By monitoring the execution of the OS at runtime it can allow any OS which is able to run on the underlying physical hardware to also run on its VMM. However, as some instructions need to be interpreted or replaced dynamically, dynamic monitoring also takes a performance hit.

## 3.2 The Xen Hypervisor

The Xen hypervisor is a minimal Linux-like platform. Like the Linux kernel multiplexes between user processes, the hypervisor multiplexes between domains; and like the Linux kernel keeps all its processes' meta-information, such as hardware context and scheduling information, the hypervisor keeps domains' hardware context and scheduling information. In particular the hypervisor's virtual CPUs allow the hypervisor to emulate privileged operations. This is achieved through para-virtualization, which is discussed from a technical point of view in Section 3.2.1.

Also, like the Linux kernel's processes interact with the kernel through syscalls, the Xen hypervisor's domains interact with the hypervisor through hypercalls. How hypercalls are implemented in Xen is discussed in Section 3.2.4.

The process of bringing up a complete Xen environment involves booting the hypervisor, launching domain 0 and launching unprivileged domains. Each of these subprocesses can be divided into several processes, many involving different runtime systems and complex protocols, making the process complex and difficult to grasp. Section 3.2.2 describes the process from the Xen hypervisor is in control of the system when the user powers on the computer to launching unprivileged domains more thoroughly.

The hypervisor manages, in addition to its own address space, modifications to domains' address spaces. Domains have read-only access to their page tables, and the hypervisor manages page table updates in order to ensure that domains' address spaces to not overlap. Memory management in Xen is further discussed in Section 3.2.3.

The hypervisor does not, unlike Linux, need drivers for devices such as network interface cards and harddrives—this is left to the privileged domain. It is up to the privileged domain to interact with these devices and to virtualize them such that other domains can use the devices in an isolated manner. The hypervisor sets up the IRQ subsystem for the machine, and the privileged domain registers for these interrupts. The hypervisor then delivers interrupts to the domain through event channels, which are discussed in Section 3.2.5.

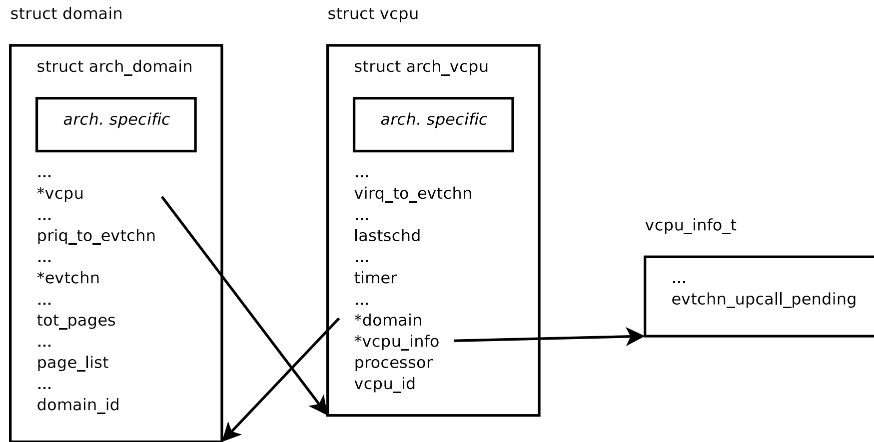


Figure 3.2: The `domain` and `vcpu` data structures.

### 3.2.1 Para-virtualization

The Virtual CPU is an important data structure in the Xen hypervisor. Since a guest OS is not allowed to modify protected registers in the CPU, they instead keep this state in a virtual set of registers provided by the hypervisor. System state is kept in memory in data structures as shown in Figure 3.2.

The `domain` data structure contains information for managing guest OS's memory and devices. The `domain_id` variable signifies the id of the domain which it belongs to. `tot_pages` counts the number of pages the domain currently has allocated, while `page_list` is a linked list which contains information about the pages allocated to the domain. `*evtchn` points to an array containing information about the event channels connected to the domain, while `pirq_to_evtchn` is an array containing the mapping between physical IRQs and event channels (see Section 3.2.5).

The `arch_domain` substructure contains architecture specific memory management information. `vcpu_id` holds the id of the virtual CPU, similarly to `domain_id` for domains, and `processor` signifies to which physical CPU in an SMP system the virtual CPU belongs. `*vcpu_info` points to another data structure, which holds more information about the virtual CPU. `timer` and `lastschd` are used by the scheduler for scheduling decisions. `virq_to_evtchn` contains mapping between virtual IRQs and event channels (see Section 3.2.5).

### 3.2.2 System Startup

Even though Xen is heavily based on Linux, it's startup procedure differs in that control is left to domain 0 early in the process.

1. The command line arguments provided by the boot loader are parsed.
2. Paging is initiated.

3. The scheduler (see Section 3.2.6) is initialized.
4. Wrappers for interrupts are initalized.
5. Time management is initalized.
6. The accurate timers (see Section 3.2.6) are initalized.
7. The scheduler is started.
8. Management of the PCI bus is initalized.

After the Xen hypervisor is initialized, the execution of domain 0 may start. Domain 0 is launched by the following procedure.

1. The hypervisor initializes domain 0's memory, including a shared memory page, machine-to-physical mapping and page tables (see Section 3.2.3).
2. The hypervisor adds the domain to the scheduler queue.
3. The guest OS to run in domain 0 is loaded into memory.
4. Execution of the guest OS is started in the scheduler.
5. Domain 0's guest OS initalizes its virtual and physical IRQs.
6. Initial event channels are set up (see Section 3.2.5).
7. Backend devices are set up (see Section 3.2.5).

The loading of unprivileged domains happens after Xen and domain 0 have booted. This is performed by user space tools and is dependent on the guest OS which is to be run. The user space tools run in domain 0 and use the various operations of the `dom0_op` hypercall to launch additional domains. For instance, when launching a paravirtualized Linux guest OS, the user space tool needs to

1. Load the kernel image,
2. Initialize the domain's memory,
3. Configure the domain's console port,
4. If the domain is to be used for backed drivers, configure these drivers (see Section 3.2.5),
5. Create the virtual devices for the domain (see Section 3.2.5).

### 3.2.3 Memory Management

Domains have read-only access to their page tables. Any changes to a domain's page table go through the hypervisor. This way the hypervisor can make sure that domains do not pages to memory areas outside their designated address spaces.

Each page has a set of meta-information, which the hypervisor uses to validate a domain's page table updates. `TOT_COUNT` counts the sum of how many times a domain uses a page frame for a page directory, page table or for mapping through a PTE. As long as this value is more than zero, the hypervisor can not

make it free for another domain's inclusion in its memory space. TYPE_COUNT is used for a mechanism which ensures that page tables are not writable from domains. It specifies that a page frame is in one of three mutually exclusive modes. Either it is used as a page directory, a page table or writable by the domain.

Page table updates are issued by domains through the *mmu_update* hypercall with the MMU_NORMAL_PT_UPDATE operation (see Section 3.2.4). Before performing an update to the page table of a domain, the hypervisor first checks TOT_COUNT of the page table entry to be modified. If it is zero, the hypervisor proceeds with checking that the PTE is not in the hypervisor's private address space. After the page table entry is updated the TOT_COUNT and TYPE_COUNT attributes are updated, and the TLB is flushed.

In order to assist domains in translating pseudo-physical addresses into physical addresses, the hypervisor maintains the mapping between pseudo-physical and physical pages. Whenever a domain needs access to physical address space, for instance to access the physical address of a PTE, it needs to translate the pseudo-physical address into the corresponding physical address. The hypervisor provides two tables, *machine-to-physical* and *physical-to-machine*, which all domains may use. Domains may alter these tables through the *mmu_update* hypercall through the MMU_MACHPHYS_UPDATE operation. Changes to the tables performed by domains are validated by the hypervisor in order to ensure that the mapped physical addresses are within the domain's address space.

### 3.2.4 Hypercalls

Domains issue hypercalls similarly to how Linux processes issue system calls, issuing an interrupt with a vector signifying the intended operation. Since Xen targets binary compatibility for libraries and other userspace applications, it must maintain the system calls of a native Linux environment. Thus, interrupt vector 0x80 remains reserved for system calls. A domain instead issues a hypercall with an interrupt with vector 0x82, which is unused in native Linux.

Like user processes in Linux, guest OSs do not have access to Xen's address space. Parameters for the hypercall that do not fit into registers are left in the domain's address space. Pointers to these parameters are passed to the hypervisor in registers. The variables are then fetched by the hypervisor from the guest OSs address space, and results are returned by writing directly to the domain's address space.

Much of Xen's operation can be described through its hypercalls, as they are essential to the method of para-virtualization. An extensive list of hypercalls that Xen implements is shown in Table 3.1. The set of hypercalls implemented in Xen may change from version to version; the most essential are discussed in the following.

#### update_va_mapping

As discussed in Section 3.2.3, domains are not allowed to modify their page tables directly. The hypervisor thus needs to provide an interface for modifying page tables. The *update_va_mapping* hypercall allows a domain to modify a single page table entry. This is useful, for instance, when handling a page fault.

| Name | Description |
|---|---|
| set_callbacks | Registers the event handlers of a domain. |
| set_trap_table | Installs a pseudo-IDT on a per-domain basis. |
| update_va_mapping | Allows domains to update page tables one PTE at a time. |
| update_va_mapping-_otherdomain | Allows a privileged domain to update the page tables of other domains one PTE at a time. |
| mmu_update | Batch page table update and pseudo-physical memory operations for a domain. |
| set_gdt | Allows the guest OS to install a GDT. |
| update_descriptor | Updates a segment descriptor. |
| stack_switch | Stores the hardware context of a domain in the TSS. |
| fpu_taskswitch | Sets the `TS` flag in the `cr0` register before a context switch. |
| sched_op | Operations that manipulate the scheduling of domains. |
| set_timer_op | Issues a timer event after a given amount of time. |
| dom0_op | Operations for managing domains, such as creating and destroying them. |
| set_debugreg | Sets the value of a debug register. |
| get_debugreg | Returns the value of a debug register. |
| dom_mem_op | Increases or decreases a domain's memory allocation. |
| multicall | Executes a series of hypercalls in a batch. |
| event_channel_op | Sets up and manages event channels. |
| xen_version | Returns the version of the currently running Xen hypervisor. |
| console_io | Allows a domain to access the console through the Xen hypervisor. |
| physdev_op | Substitution for OS access to BIOS for recieving PCI configurations. |
| grant_table_op | Gives or revokes access to a particular page for a domain. |
| vm_assist | This hypercall is deprecated. It allows a domain to switch between modes of memory management—for instance, support for writable page tables and superpages. |
| boot_vcpu | Used for setting up the virtual CPUs when booting domains. |

Table 3.1: The hypercalls of Xen/x86.

**update_va_mapping_otherdomain**

A privileged domain may want to modify pages that belong to other domains. The *update_va_mapping_otherdomain* hypercall updates a page table entry for a different domain. It may only be executed by a privileged domain.

**mmu_update**

The *mmu_update* hypercall can execute a set of memory management operations for domains. The operation is specified by a parameter:

- Updating a single PTE at a time is inefficient when a large number of PTEs need to be updated, since the hypervisor then is entered every time. `MMU_NORMAL_PT_UPDATE` updates a set of page table entries in a batch.

- `MMU_MACHPHYS_UPDATE` updates a machine-to-physical table entry.

**set_gdt**

The pointer to the current GDT is contained in the GDTR register. Writing to the register directly, for example, by using `mov`, or using the `lgdt` instruction to load a new pointer into the GDTR, are privileged operations. To install a GDT, a domain thus needs to use this hypercall. This allows legacy x86 programs that use segmentation, to use a GDT.

**update_descriptor**

When it comes to a domain modifying a PTE in its page table, the hypervisor does not allow the domain to change it to an arbitrary value. The same applies for segmentation. The hypervisor needs to ensure that segment descriptors map valid segments in memory that belong to the domain. The *update_descriptor* hypercall is used to manage segment descriptors in the GDT or LDT.

**stack_switch**

Switching the stack pointer is necessary when switching inbetween processes. This involves updating the ESP register to point to the stack of the new process. Writing a new pointer into the ESP register is a privileged operation, and therefore it may not be executed by guest OSs, as they run in ring 1. This operation thus is executed in the hypervisor in Xen, through this hypercall.

**fpu_taskswitch**

The *fpu_taskswitch* hypercall is used to set the TS flag in the CR0 register, as is required in order to employ lazy management of the FPU and XMM registers, as described in Section 2.2.3. A guest OS does not necessarily employ lazy management, however Linux does.

**set_callbacks**

The *set_callbacks* hypercall is used to register callbacks which the hypervisor may use to send events to a guest OS. They handle the events generated by the event channel, similar to ISRs to interrupts. It specifies an *event selector*

which maps to a corresponding event handling routine at an *event address*. The event selector contains the current segment, and the event address points to an address within that segment. In the Xenlinux guest OS, the event selector used is the Xenlinux kernel's code segment.

**set_trap_table**

Some interrupts captured in the hypervisor's IDT are redirected as events to guest kernels. For instance, guest kernels must be notified of page faults. The *set_trap_table* hypercall installs a virtual IDT on a per-domain basis. The hypervisor may then use the guest kernel's IDT to look up the interrupt handling routine to handle interrupts.

**sched_op**

The *sched_op* hypercall allows a domain to make some hypervisor scheduling decisions on its own behalf.

- The *yield* operation tells the hypervisor that it can deschedule the currently running domain and pick a new domain to schedule.
- *block* causes the domain to sleep until an event is delivered to it.
- *shutdown* shuts down, reboots or suspends a domain's execution.

**set_timer_op**

As discussed in Section 2.4.2, an OS uses timers for a various reasons. For instance, an OS uses timers to schedule the execution of processes. The *set_timer_op* hypercall requests a timer event to occur after a given amount of time, delivered through an event channel.

**dom0_op**

The *dom0_op* hypercall is used by domain 0 to manage other domains. It can do a set of different operations, specified by a parameter. The various operations include bringing up new domains, pinning domains to CPUs in an SMP system, pausing a domain and destroying a domain.

**set_debugreg & get_debugreg**

The *set_debugreg* and *get_debugreg* hypercalls are used to access the x86 debug registers. This can be useful for debugging purpososes. By using the set_debugreg hypercall, a domain can write a value into a particular debug register. The get_debugreg hypercall returns the value of a particular debug register.

**dom_mem_op**

Domains have a maximal memory limit set when they are created. Within this limit, domains can increase and decrease their memory allocation during runtime. The *dom_mem_op* hypercall is used for allocating or deallocating memory for a domain.

#### multicall

The multicall hypercall facilitates the execution of several hypercalls in a batch. This is provided as an optimization as certain high level operations may need several hypercalls to execute.

#### event_channel_op

The *event_channel_op* hypercall offers a set of operations that use or manipulate the event channel.

- *alloc_unbound*—finds a *free* port whose state can be changed into *unbound*.
- *bind_virq*—binds a virtual IRQ to a port.
- *bind_pirq*—binds a physical IRQ to a port.
- *bind_interdomain*—binds two ports together.
- *close*—closes an event channel.
- *send*—sends a message over an event channel.
- *status*—returns the current status of an event channel.

#### console_io

The *console_io* hypercall is mainly for debugging purposes. Like when booting any OS, when bringing up a guest OS, problems with configuration, drivers or hardware can make the guest OS fail to boot properly. In Linux, the reason for such a problem is assessed through the console. Domain 0 will print error messages directly to the console, for instance a VGA display. Boot messages of unprivileged domains are printed to their individual consoles. These consoles can be accessed through the domain's network interfaces using, for instance, SSH; however, should the boot of a domain fail, its network interface might not be available.

This interface allows the user to read from or write to the console of a guest OS.

### 3.2.5 Event Channels

There are two methods of communicating events between entities, namely *event channels* and *shared memory rings*. Event channels are used to send event notifications asynchronously. The notifications are queued and picked up by domains after they are scheduled for execution. Shared memory rings are used for sending larger chunks of data between domains, such as, for instance, data being transferred from a backend block device to a frontend block device.

Event channels are managed by the hypervisor through the *event_channel_op* hypercall (see Section 3.2.4). A domain can have a set of event channel connection points, or *ports*, which each may connect to one end of a channel. The other end of a channel connects to either a *physical IRQ*, a *virtual IRQ* or a port of another domain. A physical IRQ refers to the native IRQ of a device, while a virtual IRQ refers to an IRQ controlled by the hypervisor. This scheme

allows, for example, an event generated by a backend driver to be delivered as a virtual IRQ to a frontend domain.

After an event channel is established, the notification of an event is left pending as a bitmask in the `evtchn_upcall_pending` variable, which is found in the `vcpu_info` data structure, which again pointed to in the `vcpu` data structure (see Figure 3.2). When the value of `evtchn_upcall_pending` is non-zero, this means that there is an event pending. The guest OS reads the value of the variable through the shared page. If it is non-zero, it executes the pending interrupts and resets the variable to zero.

The frontend block device driver is a state machine and is controlled by a *domain controller*, which is a set of control tools for managing domains and devices. This is run in a privileged domain, which is required for managing other domains and devices. When the frontend driver is initialized, it notifies the controller through a special event channel, which is called a *control interface*. This is shown in Figure 3.3. Note that the constant names have been abbreviated in the figure. The first message is a `BLKIF_INTERFACE_STATUS_UP` message constant, which tells the domain controller that the frontend block device is ready. The domain controller responds with a `BLKIF_INTERFACE_STATUS_DISCONNECTED` message. This transitions the driver from the state *closed* to *disconnected*, as illustrated in Figure 3.4. At the same time, the driver initializes a shared ring buffer and responds to the domain controller with a `FE_INTERFACE_CONNECT` message. The domain controller then responds with a `BLKIF_INTERFACE_STATUS_CONNECTED` message, which also contains information about the event channel of the backend driver. This stimulates the driver to connect to the event channel and transitions the driver into the final *connected* state. The new event channel connects to the backend driver and is mapped to an IRQ in the frontend domain.

Before the frontend domain is run and the frontend driver is created, the backend device is set up, also by the domain controller. The domain controller starts by sending a *CMSG_BLKIF_BE_CREATE* message to the backend driver. A handle for the new frontend driver is then put in the backend driver's hash table. Then, the domain controller proceeds with a *CMSG_BLKIF_BE_VBD_-CREATE* message, which initializes a virtual block device in the backend driver. Finally, a *CMSG_BLKIF_BE_VBD_CONNECT* message makes the backend driver connect to the new event channel. If one of many error conditions occur, an error message is returned.

### 3.2.6 Time and Scheduling

The hypervisor has procedures and data structures for implementing *accurate timers*. Information about each timer is maintained in data structures, `ac_timer`, which keep track of, among others

- When the timeout event is to occur,
- Which CPU the timer belongs to,
- A routine which the timer is to execute when it reaches the specific moment.

The scheduler maintains three different timers:

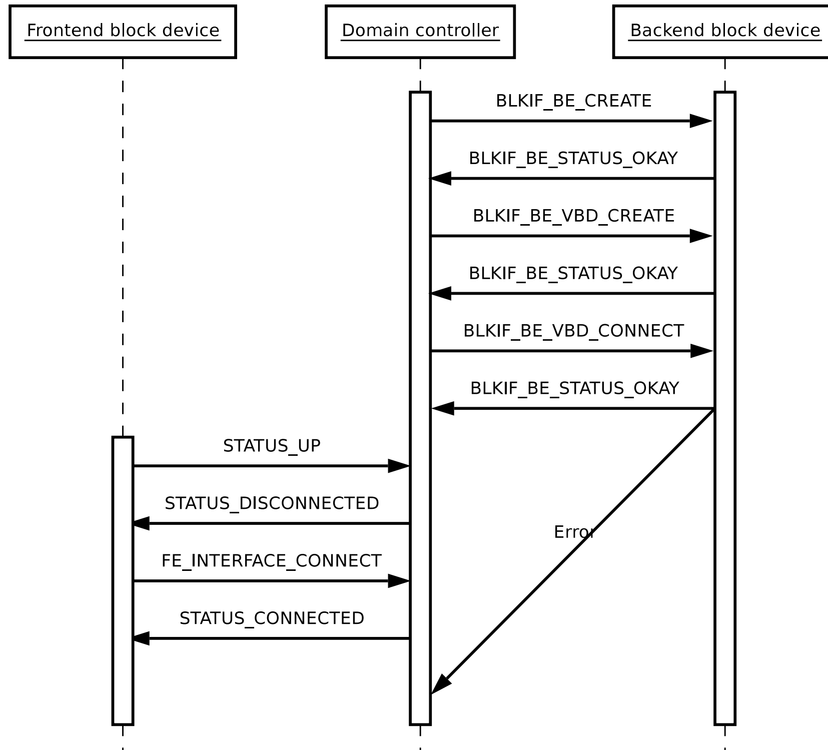


Figure 3.3: Virtual block device initialization phase.

Figure 3.4: The states and transitions of a frontend block device.

- `s_timer` is used on each CPU to make decisions about preemtion and scheduling.
- `t_timer` is used on each CPU to implement a timer for sending timer interrupts to the running domain.
- `dom_timer` specifies the timeout for domain timers.

Virtual timer interrupts are delivered to the domains through event channels using virtual IRQs. The hypervisor's scheduler decides when to deliver the interrupt by setting a new `ac_timer`. Scheduling of domains is performed throught the `__enter_scheduler()` procedure in the hypervisor, which is registered as a softirq routine. The scheduler depends on the accurate timer for execution of its softirq, and for each CPU an accurate timer is initialized.

# Chapter 4

# IA-64 Implementation

This chapter presents the implementation of the IA-64 port of Xen, or *Xen/ia64*. Many of the solutions used in the vBlades project (see Chapter 6) can be transferred into Xen/ia64, as they are similar projects in many ways. This, and other design issues are discussed in the following Section 4.1. The implementation of the essential aspects of Xen/ia64, such as para-virtualization and memory management, is presented Sections 4.2 through 4.6. Aspects which are left for future implementation, are left in Section 4.7.

## 4.1 Analysis

As discussed in Section 3.1, if a VMM is to successfully virtualize its underlying architecture, all instructions that make the system behave differently depending on the CPL, must either be replaced directly in the guest kernel program—that is, either using para-virtualization or binary translation—or they must generate faults which the VMM can catch, which makes it possible to emulate the faulting instruction. Unfortunately, not all IA-64 instructions whose altering of system state depends on CPL, fault when they should. The problematic *privilege-sensitive* instructions of IA-64 are discussed in Section 4.1.1.

Daniel J. Magenheimer and Thomas W. Christian [MC04] have examined different methods of both virtualizing and para-virtualizing with Linux on the IA-64 architecture. Their research shows that the IA-64 architecture is also "uncooperative" in respect to virtualization and that para-virtualization therefore is beneficial also on the IA-64 architecture. The IA-64 implementation of Xen can take advantage of some of the design choices made in vBlades. Particularily the concepts of *optimized para-virtualization*, *transparent para-virtualization* and *metaphysical memory* are worth taking into consideration and are discussed in Sections 4.1.2, 4.1.3 and 4.1.4, respectively.

Many of the hypercalls of Xen/x86 are redundant in Xen/ia64. Particularly, hypercalls that deal with x86 segmentation are not needed in Xen/ia64, as IA-64 OSs are not expected to use segmentation. Also, the use of metaphysical memory may alleviate the need for some hypercalls. The relevance of the most important Xen/x86 hypercalls for Xen/ia64 is analyzed in Section 4.1.5.

The IA-64 architecture sports support for different page sizes. This, however, poses some problems, since the hypervisor cannot predict the page size of a guest

OS which has a user selectable page size, such as Linux has. Particularly, VHPT support is a feature depending on page sizes; this is discussed in Section 4.1.6.

Finally, in Section 4.1.7, a few implementation issues of the Linux/ia64 Perfmon capability are discussed.

### 4.1.1 Privilege-Sensitive Instructions

There are three privilege-sensitive instructions in the IA-64 architecture. One example is the `cover` instruction. It allocates a new register stack frame with zero registers. This instruction is used in the process of manipulating the RSE on an interrupt. In a following kernel entry in Linux, the register stack frame is switched from the user stack frame to the kernel stack frame. This protects the user level register stack frames from being overwritten by interrupt handlers (see Section 2.4.4).

The `cover` instruction itself is not a privileged instruction, so the need to emulate this instruction is not obvious. However, looking at what the instruction does in detail reveals that this is indeed a privilege-sensitive instruction. If the PSR.IC flag is clear, a `cover` instruction writes to the IFS register. Writing to the IFS register, however, requires a CPL of zero. Therefore, even though `cover` may be executed regardless of CPL, it may entail breaching protection, and is thus privilege-sensitive.

Two other privilege-sensitive instructions are the `thash` and `ttag` instructions. The `thash` instruction generates the hash address of an address translation, that is, given an address, it will find the location of that address in the VHPT hash table. The problem with this instruction is that the location that the instruction returns is the one of the main page table and not that of a guest OS. Similarly, the `ttag` returns tag information about an address, and, also similarly, the information returned comes from the main page table. These instructions being unprivileged, a guest OS may execute them without faulting and the hypervisor being able to catch them. Thus, the guest OS may be given faulty information.

### 4.1.2 Optimized Para-virtualization

Modifying an OS to run above the Xen/x86 hypervisor requires some essential changes in the source code of the OS, as discussed in Section 2.6. There are, however, certain disadvantages to para-virtualization. Most importantly, porting the OS kernel to the Xen/x86 virtual architecture requires a certain amount of programming effort. Secondly, adapting existing systems to running Xen requires the substitution of the current, possibly specially adapted, OS kernels. Moreover, having different binaries for non-virtualized and virtualized instances of Linux can be an administrative problem for both users and OS providers in that one extra instance of each OS image needs to be provided.

To alleviate these problems, vBlades introduces the concept of *optimized para-virtualization* [MC04]. This means trying to satisfy two goals to an extent which is feasible—or "optimal":

- Make the changes necessary to the OS kernel to run on the hypervisor as little as possible.

- Make sure the efficiency of the para-virtualization is not impacted too much by the former.

These two goals are, however, conflicting, since reducing code impact to, say, zero, leads to the efficiency impact of interpreting or binary patching virtualization, as discussed in Section 2.5.1. Clearly, the qualifications of these goals are subjective.

vBlades achieves a compromissorial solution by not modifying the source code of the guest OS at all; rather, the binary code of the OS kernel is modified, similarly to binary patching, though between compile-time and run-time. The instructions that are privilege sensitive are known and can be translated into other non-privilege-sensitive and privileged or other instructions that raise exceptions and can then be trapped by the hypervisor. These instructions are then emulated by the hypervisor. The procedure of translating these instruction is called a *binary translation pass*.

### 4.1.3 Transparent Para-virtualization

The term *transparent para-virtualization* [MC04] means that the same para-virtualized Linux kernel compiled binary can be used with both physical hardware and the hypervisor as execution environments, transparently—i.e. the OS itself knows whether it is running on the hypervisor or on physical hardware. As noted in Section 4.1.2, having different binaries for running on physical hardware and on the hypervisor can be an administrative problem. If the performance difference between a transparently para-virtualized and native OS kernel running on physical hardware is small enough, an OS provider may simply issue a single binary, and the user can choose whether to run on a hypervisor or on hardware.

VBlades uses a special flag to let the OS know whether it is running on the hypervisor or on hardware. This flag is set in a reserved bit in a privileged configuration register, such that it will not interfere with normal operation. The IA-64 architecture specifies that reserved registers are always set to zero. When an OS is running on the hypervisor, the hypervisor returns the value of the register to the OS, while setting the flag, letting the OS know that it is running virtualized. Otherwise, the flag is read as zero by the OS, letting it know that it is running on hardware.

VBlades have shown that transparent para-virtualization is feasible. Performance differences between native Linux and transparently para-virtualized Linux are small and expected to be less than 0.1 per cent in disfavor of transparent para-virtualization [MC04].

### 4.1.4 Memory Management

Region registers allow the address spaces of different processes to separated. This is exploited to separate the address spaces of domains in vBlades. Updating the region registers is a privileged operation, so the hypervisor can intercept updates to these registers by the guest OS and ensure that their address spaces are isolated.

The hypervisor can not allow a guest OS to access physical memory directly. This would mean that domains can avoid protection mechanisms and adversely

affect other domains' execution. Sometimes, however, OSs do access physical memory and thus, in order for OSs to be ported to Xen/x86, physical memory is virtualized, as discussed in Section 2.6.3.

vBlades virtualizes physical memory in a different manner, referred to as *metaphysical addressing* [MC04]. In the metaphysical addressing scheme virtual addresses are translated into addresses within domains' private address spaces using region registers, making the guest OS believe it is actually accessing physical memory.

This approach to virtualizing physical memory access is advantageous in that it minimizes the para-virtualization effort for guest OSs, since they may access virtualized memory transparently, as if natively.

### 4.1.5 Hypercalls

Many hypercalls are not necessary on the IA-64 architecture, as they deal with specifics of the x86 architecture. The complete set of hypercalls of Xen/x86 is listed in Table 3.1. The most important hypercalls are described in more detail in Section 3.2.4. An evaluation of the relevance of these hypercalls in terms of Xen/ia64 implementation, follows.

#### update_va_mapping

Verifying page table updates can be avoided using the approach of vBlades, using region registers. Since each domain has its own logically disjunct memory area, page table mappings can not reference illegal memory areas.

#### update_va_mapping_otherdomain

Changing the page tables of other domains is not part of the native functionality of an OS. Though the *update_va_mapping_otherdomain* hypercall may be useful, it is not important in an initial implementation and should be deferred until later.

#### mmu_update

The isolation which is achieved through having page table updates be validated by the hypervisor, is also favorable in the IA-64 architecture. Also, seperating the physical address spaces of different domains is necessary to ensure isolation. However, taking vBlades' approach to memory virtualization may eliminate the need for some of this hypercall's operations. Instead of maintaining the "machine-to-physical" and "physical-to-machine" tables metaphysical addressing has the same effect. This eliminates the need for the `MMU_MACHPHYS_UPDATE` operation.

#### set_gdt

This hypercall allows legacy x86 programs that use segmentation to use a GDT on the Xen/x86 platform. The list of programs that use segmentation is limited, particularly in the Linux OS. Still, some legacy programs, programmed for Microsoft OSs, make use of the LDT. WINE [win05] runs in user space on top of different OSs and allows programs programmed for Microsoft OSs to run in

the host OS. WINE has been known to allow user space programs to modify the LDT. However, this is not a problem in the IA-64 architecture, as the GDTR and LDTR registers are directly modifiable by user space code. Moreover, the IA-64 specification [Intb] reads that the IA-64 architecture is

> defined to be unsegmented architecture and all Itanium memory references bypass IA-32 segmentation and protection checks,

and user level code can

> directly modify IA-32 segment selector and descriptor values for all segments,

including those specified in the GDT and LDT tables. Following that the segmentation registers are not protected, one should expect that guest OSs do not use segmentation.

### update_descriptor

Applying the argument above, that the *set_gdt* hypercall is redundant because the use of segmentation logic is not protected, the *update_descriptor* hypercall is also redundant.

### stack_switch

This hypercall is necessary for Linux/x86 because it uses the ESP register, which is protected, to point to the Kernel Mode Stack of the current process. In Linux/ia64, however, the SP register points to the Kernel Mode Stack. SP is an alias for R12, which is a general register. Therefore, modifying the Kernel Mode Stack pointer is not a privileged operation, and followingly the *stack_switch* hypercall is **redundant** in Xen/ia64.

### fpu_taskswitch

Linux/ia64 manages the upper floating point registers, registers F32 through F127, lazily. Whenever a process has used any of the upper floating point registers it becomes the *fpu-owner*. The DFH flag is set in the PSR register for other processes, signifying that access to the upper floating point registers is disallowed. Thus, when another process tries to use these registers, a *Disabled Floating-Point Register* fault is raised. The Linux kernel then proceeds with storing the upper floating point register state for the *fpu-owner*. Then, it lets the faulting process continue with using the upper floating point registers, after clearing the DFH flag.

Writing to the PSR register is a privileged operation, and thus this operation needs to be virtualized. This can be done either with the *fpu_taskswitch* hypercall, or the hypervisor can intercept the protection fault from accessing the register and emulate the operation. The latter solution leads to less paravirtualization impact on the guest OS kernel.

**set_callbacks**

Callbacks registered in the hypervisor are necessary for the functioning of event channels. The hypervisor uses these callbacks to deliver events from the hypervisor to the domain. This hypercall is necessary in order to virtualize block and network devices and therefore necessary for running unprivileged domains.

**set_trap_table**

[MC04] shows another approach to registering the ISRs of guests; the hypervisor registers the guests' IVA and on an interrupt transfers execution to the domain's ISR in its IVT. Taking this approach, trap tables need not be manually initilized by a guest domain, making this hypercall redundant. This approach is also more attractive with regards to an "optimized" implementation.

**sched_op**

The *sched_op* hypercall's operations, *yield*, *block* and *shutdown* allow domains to control their own execution. However, the *yield* and *block* operations are not necessary for the domains' operation. The *shutdown* operation can be regarded as necessary if para-virtualized Linux is to behave similar to native, as native Linux is expected to be shut down when being instructed to do so. In the IA-64 architecture, the shutdown mechanism is accessed through the EFI. Using the EFI's shutdown service is a privileged operation, thus, virtualizing this operation is necessary. The *sched_op* hypercall is one solution. Another solution is similar to one taken by vBlades, in which the faulting EFI call is intercepted by the hypervisor and emulated.

**set_timer_op**

Linux depends on a timer to schedule tasks. However, the *accurate timer* (see Section 3.2.6) may be used to genarate timer interrupts instead the native hardware timers. This hypercall can be considered **redundant** in Xen/ia64.

**dom0_op**

This hypercall is used by domain 0 to manage other domains. Since it is needed to bring up other domains it has **high priority**.

**set_debugreg & get_debugreg**

In Xen/ia64 these should be replaced by hypercalls that manage the Perfmon interface. However, this is not essential for Xen's operation and can be deferred until later; therefore it should have **low priority**.

**dom_mem_op**

Again, looking at the vBlades implementation, the virtual memory address space is not partitioned between domains. This is because the region registers allow domains to have logically separate virtual memory regions. Domains thus need not request an increase in memory allocation from the hypervisor. The

*dom_mem_op* hypercall therefore is not necessary for operation and is **redundant**.

**multicall**

The multicall hypercall facilitates the execution of several hypercalls in a batch. This is provided as an optimization as certain high level operations may need several hypercalls to execute. This hypercall may be implemented on a later stage and has **low priority**.

**event_channel_op**

The *event_channel_op* hypercall is necessary to set up and use the event channel interface. Event channels are necessary for communication between domains. If several domains are to share virtualized block and network devices, there needs to be a communication link between the domain running the backend drivers and the domains running frontend drivers.

**console_io**

The *console_io* is needed to see what is happening in unprivileged domains. Seeing the output from guest OSs is pertinent for the development of their support. However, letting the guest OS print console messages directly to the same console as domain 0 is sufficient as a provisional solution.

### 4.1.6 Virtual Hash Page Table

VHPTs are important for Linux/ia64's performance, and how VHPT should be implemented in Xen/ia64 is worth discussing. There are three approaches in this matter—either

- Let each domain have a VHPT maintained by the hypervisor,
- Have a single global VHPT maintained by the hypervisor,
- Or implement both solutions, and let the user choose.

Having a single global VHPT maintained by the hypervisor poses a problem if different guest OSs have different page sizes. Given that different guest OSs use different page sizes, every time the hypervisor switches from one domain to the other, the VHPT must be purged and reinstalled with the page size of the new domain; this can be a costly affair. One solution to this problem is to have a minimal page size in the global VHPT, say 4 KB, and let guest OS's larger pages be composed of multiples of the smaller global pages.

To maintain a separate VHPT for each domain also poses some problems. The hypervisor then needs to allocate memory for each of these VHPTs, memory which is not allowed access to from domains, which makes this a waste of memory from a domain's point of view. Thus, every time bringing up new domains requires the allocation of a new such a chunk of free memory, which may be hard to find, once several domains are running.

### 4.1.7 Performance Monitoring

The PMU of the IA-64 CPU offers the capability of monitoring usage of CPU resources. By porting Perfmon to the Xen/ia64 platform, existing Perfmon tools can be used to monitor different domains's performance characteristics. This would allow easier identification of possible areas for improvement in Xen code.

An important question is whether to place Perfmon in the hypervisor or in the para-virtualized Linux kernel. The minimal code impact argument applies here aswell. The hypervisor can let the guest OS keep its existing Perfmon functionality and emulate or para-virtualize the interface to hardware. By incorporating Perfmon functionality into the hypervisor, the hypervisor can monitor fine-grained performance aspects of domains without performance impact, similar to how the Linux kernel with Perfmon can monitor performance aspects of processes.

In adapting Xen to Perfmon, a few restrictions are placed:

- In order for Perfmon to monitor different processes, PMU state must be included in process switches. This also needs to be done when switching between domains.
- Perfmon uses hooks for process fork and exit, which are used to start and end, respectively, the monitoring of processes. These hooks are also needed in the creation and termination of domains if domains are to be monitored.
- Perfmon uses data structures and a sampling buffer for recording data. For this, memory allocation and remapping procedures are needed.
- *File descriptors* are used to interface with Perfmon. Thus, the ability to create file descriptors is necessary.
- A system call is needed to access Perfmon in the guest kernel, namely *perfmonctl*. If Perfmon resides in the hypervisor, this means that a new hypercall is needed.
- An interrupt handler for PMU interrupts needs to be registered. PMU interrupts are external interrupts and IRQs may be handled in the hypervisor or forwarded to domains.
- If guest kernels are to contain the Perfmon logic, they need access to PMU registers, which are generally protected. This has to be virtualized in some manner.

## 4.2 Para-virtualization

Xen/ia64 lets guest OSs run in ring two. This demotion from their original CPL of zero has some implications on how CPU state is handled. In order to give guest OSs the illusion of running on a real platform, the CPU is virtualized, that is, some registers are emulated. Also, as discussed in Section 4.1.2, privileged and privilege sensitive instructions need to be virtualized. A few examples of virtualized and para-virtualized operations are discussed in the following.

mov instructions that target the PSR register, are privileged instructions, since the PSR register is protected. When such instructions are executed, a Privileged Operation Fault is raised, and the hypervisor handles the exception in its IVT. The hypervisor can then find out which instruction generated the exception in the IFA register. Thus, there is no explicit need to para-virtualize these instructions. The hypervisor can instead emulate the intended instruction. Appendix C.4 shows the procedure that virtualizes the mov instruction for reading the PSR register. The vcpu_get_psr procedure reads the value of the real PSR register, but filters the value such that a CPL of zero is set in the virtual CPU. Also, the IC and I flags are filtered according to the state of the virtual CPU.

Linux uses the thash instruction in handling a set of exceptions. One example is the *Data Dirty Bit* fault. It is raised when a page is stored to while its dirty-bit flag is off in its PTE. The Data Dirty Bit ISR handles the fault by turning on the dirty-bit in the PTE. In order to get the address of the PTE, the ISR uses the thash instruction. As described in section 4.1.1, this instruction is sensitive and must be para-virtualized.

In order to para-virtualize this instruction, it is replaced with the tak instruction. This instruction is privileged, and execution of it by a guest OS running in ring two, will cause a Privilege Operation Fault. This fault is caught by the hypervisor, and thus the intended thash operation can be emulated.

The cover instruction also needs to be para-virtualized. It is used in a few places in the Linux source code, particularly in some of the IVT's ISRs. The SAVE_MIN_WITH_COVER macro defines the procedure for switching to the kernel stacks—both the Kernel Memory Stack and the Kernel Register Frame Stack. This is used to save the context before entering the ISR. In native Linux, the IFS register is saved as a part of this context. In Xenlinux, the macro is modified such that if a cover instruction should fail—that is, not being able to update the IFS register—a fake IFS register, with the right state, is saved instead.

Faults generated by domains are caught by the hypervisor's IVT since it is the one running in ring 0. Thus, in order to notify the guest OS of the fault, the hypervisor *reflects* the fault to the guest OS. For example, when a TLB miss occurs in a guest OS, the TLB miss is reflected to the guest OS's virtual CPU. The hypervisor calculates the location of the VHPT of the guest OS and stores it in the virtual CPU's virtual IHA register.

Like Xen/x86, the IA-64 port makes use of the domain and vcpu data structures (see Figure 3.2). These data structures are made to be architecture independent, so they are used without modification. Architecture specific information is put in sub-structures, arch_domain and arch_vcpu. For the purpose of virtualizing the CPU, the arch_vcpu data structure is used.

The arch_vcpu and arch_domain data structures are illustrated in Figure 4.1. Entries in the data structures are listed ascending, with lower memory addresses at the bottom. In the arch_vcpu the first six bottom entries give a virtual interface to handling the TLB. By virtualizing the IVA register, the hypervisor can specify the location of the IVT of a domain. Also, the ITC and ITM registers are virtualized to allow timing in the domain. metaphysical_rr0 specifies the region to use when in metaphysical addressing mode. This value is set to the real region register by the hypervisor before entering metaphysical addressing mode.

In the arch_domain data structure, active_mm is a pointer to another data

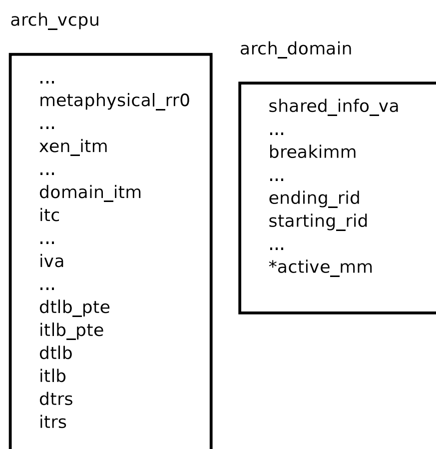


Figure 4.1: Architecture specific parts of the `domain` and `vcpu` data structures.

structure, which contains memory management information about, among others, memory areas and page tables. `starting_rid` and `ending_rid` specify the region id range the domain can use. `breakimm` specifies the immediate which is to be used as the parameter to the `break` instruction when the domain issues a hypercall. `shared_info_va` specifies the address of a shared memory area between the hypervisor and the domain.

## 4.3 Optimized Para-virtualization

Whenever a privileged instruction is executed with insufficient CPL, a Privilege Operation fault exception is raised. By extending the Linux/ia64 IVT to include handlers for this exception, the hypervisor can then take the necessary actions to emulate the instruction that generated the exception. When a Privilege Operation fault occurs, the hypervisor looks at the IFA register to find out which instruction caused the exception. The hypervisor then emulates the instruction by manipulating, instead of the state of the real CPU, the state of the virtual CPU.

Emulating privilege sensitive instructions requires a little more work than privileged instructions. In a binary translation pass, privilege sensitive instructions are first translated into instructions that fault. These exceptions are then handled by the hypervisor similar to privileged instructions.

IA-64 binary code has to be examined more carefully than x86 code. Since instructions are bundled, and some instructions may span more than one slot, the template needs to be examined to find out the format of the bundles. The binary translation program reads a bundle at a time, each being 128 bits long. By looking at the bundle's template, it can be established whether it can contain

a privilege sensitive instruction. For instance, if the bundle is in the format **BBB**, one of the instructions can be a `cover` instruction, since the ISA defines `cover` as a **B** instruction. Since the IA-64 ISA specifies that a `cover` instruction must be the last instruction in a group, it must also be the last in a bundle, and therefore slot 2 may need translation.

Privilege sensitive instructions are replaced with instructions that fault. For instance, `cover` is replaced with `break.b 0x1fffff`. This fault is caught by the hypervisor and the intended instruction is emulated.

## 4.4 Memory Management

Each domain is assigned a range of RIDs. The hypervisor manages the state of the region registers on each domain switch. These registers are stored in a `rrs` array, when a domain is descheduled. When a domain is rescheduled, the `rrs` values are loaded back into the physical region registers.

Like vBlades, Xen/ia64 uses metaphysical memory in order to give guest OSs the illusion of having access to physical memory. When a guest OS tries to address physical memory directly, first, it tries to clear the PSR.DT flag (see Appendix A.2) by executing the instruction,

```
rsm psr.dt
```

Since the guest OS has insufficient privileges, the CPU raises a Privilege Operation fault. This exception is intercepted in the IVT under the *General Exception Vector*. By reading the ISR register, the hypervisor assesses the cause of the exception. It can then emulate the instruction on the virtual CPU by using a replacement routine, `vcpu_reset_psr_sm()`, storing the state in a virtual PSR register.

Whenever a domain accesses memory, the hypervisor finds this domain's mode of memory operation in the domain's virtual PSR register. If it happens to be metaphysical, the hypervisor first makes sure that the address is within the domain's address space. It then calculates the page table entry corresponding to the physical address. It then translates the address into a corresponding virtual address, transparently for the domain. The mappings between the virtual and the physical addresses are calculated by adding a constant, 0xf000000000000000, to the physical address.

## 4.5 Firmware Interface Emulation

When the hypervisor loads a guest OS, it is, instead of following the convention of the EFI system table, given a custom table from the hypervisor, which replaces it. The new table, instead of pointing to EFI routines, now points to hypercalls that emulate these routines. This can be done quite elegantly because of metaphysical memory. This also allows guest OSs that are previously unmodified to use the emulated EFI routines, which is a more attractive solution with regards to optimized virtualization.

The guest OS expects the EFI System Table to reside at the beginning of its physical memory area. Since the guest OS really is using metaphysical memory, the hypervisor controls where the guest OS will find the EFI System

Table. At the start of the domain's metaphysical memory area, the hypervisor insterts a fake EFI System Table, along with a corresponding runtime services table. In the runtime services table, fake replacement routines for the native EFI routines are insterted. These replacemet routines emulate the real routines by doing some appropriate operation, or, if they are unimportant, they may simply fail without causing any problems to the guest OS. Some operations, such as *efi_get_time* and *efi_reset_system*, require the hypervisor to access the real EFI System Table. In this case the emulation routine calls a hypercall, which makes the hypervisor execute the real EFI routine.

For instance, instead of the native `ResetSystem()` EFI runtime service, the domain will call a replacement hypercall, when jumping to the `ResetSystem()` address in the fake runtime service table. The hypervisor receives the hypercall and may decide whether or not to allow the domain to reset the system. If it decides to allow a system reset, it uses the real runtime service table to execute the real `ResetSystem()` EFI routine. In the case of *efi_get_time* the system time is transparently returned to the guest OS.

## 4.6 Events

During execution of a domain, the hypervisor is still in control over interrupts and exceptions. The hypervisor's IVT is still installed, such that interrupts and exceptions will interrupt the execution of a domain, and control is transferred to the hypervisor's ISRs. This way, the hypervisor can catch privileged operations from a domain and emulate the operation. Thus, General Protection exceptions are handled directly in the hypervisor, as described in Section 4.3.

External interrupts, however, are left as pending for domain 0, and the domain is woken from the scheduler. The virtual CPU is notified of the interrupt by setting a `pending_interruption` variable in the `vcpu` data structure and the interrupt vector is stored in the virtual CPU's virtual IRR register.

Hypercalls are invoked with a `break` instruction (see Appendix B.2). A vector, *imm*, is used as an argument, which determines whether it is a hypercall or, for instance, a system call. The vector used is defined by the macro `BREAKIMM`, and this vector is compared to the domains' `breakimm` variable in their `arch_domain` data structures (see Figure 4.1). This lets each domain define its own hypercall vector. A second vector determines which hypercall to execute. This vector is not used as a direct argument to the `break` instruction, but loaded into a register before the call.

In the case of the *event_channel_op* hypercall, the vector used is defined by the `EVENT_CHANNEL_OP` macro. The two example code snippets in Appendices C.1 and C.2 show the high-level interface between the hypervisor and the domain. An event channel is established between two domains by sending an `evthcn_op_t` data structure to the hypervisor. This data structure contains information about which two domains and ports it connects. A pointer to this data structure is then loaded into register R3, and an *event_channel_op* hypercall is invoked. The hypervisor context switches and stores some of the context, including R3, in a `pt_regs` data structure. The intended hypercall type is found in the stored context in memory, and the pointer to the `evthcn_op_t` data structure is found in the `r3` variable, also in the stored context. The hypervisor then continues to establish the event channel.

## 4.7 Further Work

One topic left for future implementation is support for VHPT. Currently, VHPT needs to be disabled in the guest Linux kernel at compile-time. This is a reasonable demand in terms of para-virtualization but not in optimized para-virtualization terms. Guest kernels that have VHPT enabled will need modification, which violates the principles of optimized para-virtualization. Also, the option of VHPT is preferable in terms of achieving an as close to native as possible performance.

As the IA-64 port of Xen nears sufficient stability, starting to pinpoint performance hotspots can be valuable. To do this, it is valuable to make use of the performance monitoring capabilities of the IA-64 CPU. The natural way of enabling performance monitoring in Xen/ia64 is to modify the Perfmon interface such that it can be used on Xen/ia64.

Currently, there is no mechanism for passing large data structures inbetween the hypervisor and a domain. Data structures that are passed, have to be sized within the limits of a page, as there is no guarantee that consecutive pages in the hypervisor's memory space are also consecutive in a domain's memory space. This is sufficient for some data structures, such as, for instance, messages that are sent through the *event_channel_op* hypercall in order to establish event channels. However, in the long run, ensuring that larger data structures are passed complete should be implemented through a copy mechanism.

Support for multiple processors in an SMP system has been deferred until a later stage, when Xen/ia64 is more stable and functional. SMP is non-trivial, and implementing support for it at an early stage may complicate development. Also in Xen/x86_64, the x86_64 port of Xen/x86 (see Section 6), support for multiple processors has been delayed. Still, supporting multiple processors is critical for the progress of Xen/ia64 in HPC.

Event channels and the control interface are functional, but still some work needs to be done in order to make virtual block drivers and network drivers work. Virtual block and network drivers are necessary for the functioning of unprivileged domains, and therefore the further development of these drivers should have high priority.

Entering the kernel through a `break` exception is costly, and Linux/ia64 developers have begun solving the problem of costly system calls with the implementation of *fast system calls*. Charles Gray et al. [G+05] show that by using the `epc` instruction instead of the `break` instruction for inducing a system call, a more than tenfold speedup can be achieved in terms of the number of CPU cycles required to enter the kernel. A similar speedup may be achieved if `epc` is used instead of `break` in hypercalls.

# Chapter 5

# Performance Analysis

This performance analysis aims to assess the distance to the goal of achieving reasonable efficiency in paravirtualizing the IA-64 architecture. Section 5.2 explains the methodology used in measuring the performance of Xen/ia64. The results of the benchmarks are presented in Section 5.3.

## 5.1 Experimental Environment

The OpenLab GRID testbed cluster consists of two-way HP rx2600 nodes with Itanium 2 processors. The nodes run the *Scientific Linux CERN 3 (SLC3)* Linux distribution, which is based on Red Hat Enterprise Linux 3. The hardware and software configuration of the machine used for performance analysis is summarized in Table 5.1.

## 5.2 Methodology

Two methods are used to analyze the execution of Xenlinux. The first analysis is a macro-benchmark, and involves building the Xen hypervisor and measuring the time it takes to complete. Xen is chosen over Linux as the build subject because building Linux in **para-xenlinux** halts after around 10 minutes, due to an

| *Hardware* | |
|---|---|
| Model | HP rx2600 |
| CPU | Itanium 2 |
| CPUs per node | 2 |
| Memory | 2 GByte |
| Storage | Ultra320 SCSI |
| *Software* | |
| OS | Linux 2.6.10 |
| OS distribution | Scientific Linux CERN 3.0.4 |
| Compiler | GCC version 3.2.3 20030502 (Red Hat Linux 3.2.3-42) |

Table 5.1: The configuration of the OpenLab GRID testbed cluster.

unresolved, though probably short-lived, bug, at the time of writing. Building Xen takes shorter time and thus avoids this bug. The second analysis instruments the execution and counts the number of virtualized operations executed. These analyses are discussed in the following sections.

### 5.2.1 Build Benchmark

A kernel build benchmark is a realistic benchmark. Compiling is very CPU and memory intensive and should give an overall view of performance. Still, it should be noted that the build benchmark is not perfect and that results have some variability due to conditions of the underlying hardware, which is non-deterministic.

Performance is analyzed by measuring the time it takes to compile the Xen hypervisor kernel. Time is measured and compilation started by entering

```
time make
```

`time` returns how much time the compilation has used with respect to wall clock, user mode and kernel mode. Each time, before compiling Xen,

```
make clean
```

is run to ensure that Xen is compiled from scratch.

Four data points are measured:

- *Native Linux*—the native Linux kernel, which runs without virtualization. In the following, this data point is referred to as **native-linux**.
- *Para-virtualized Xenlinux on bare metal*—the transparently para-virtualized 2.6.12 kernel running directly on physical hardware, i.e. without Xen. In the following, this data point is referred to as **native-xenlinux**.
- *Para-virtualized Xenlinux on Xen*—the transparently para-virtualized 2.6.12 kernel on the Xen hypervisor. In the following, this data point is referred to as **para-xenlinux**.
- *Binary translated Linux on Xen*—a Linux 2.6.11 kernel, binary translated, running on Xen. In the following, this data point is referred to as **privified-linux**.

For each data point, five measurements are made, and the best and worst results are discarded. Then, the average of the three middle results is calculated. All measurements are made in *single user mode*, except for **privified-linux**, which is run in normal multi-user mode, and thus other processes may influence its measured performance. Also, **privified-linux** uses an older version of Xen, in which the reported CPU frequency is 9/15 of the real CPU frequency, which makes the `time` command measure times that are 15/9 of the real time. The real time from **privified-linux** measurements are therefore inferred by multiplying the reported time with 9/15.

| Time / s | Wall | User | Kernel |
|---|---|---|---|
| **native-linux** | 24.10 | 22.19 | 0.88 |
| **native-xenlinux** | 24.45 | 22.54 | 0.85 |
| **para-xenlinux** | 25.27 | 23.02 | 1.22 |
| **privified-linux** | 33.22 | 25.65 | 5.87 |

Table 5.2: Measured time when compiling Linux.

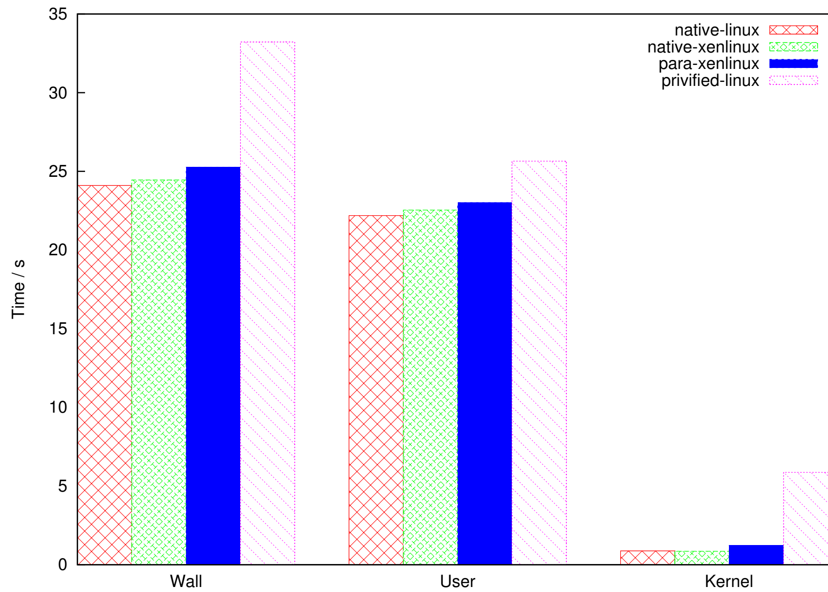


Figure 5.1: Bar chart showing the times measured in the build benchmark.

### 5.2.2 Instrumentation

The Xen hypervisor is instrumented with counters for measuring the amount of virtualized privileged operations are executed. Just before starting the compilation benchmark, these counters are zeroed, and just after the benchmark has finished, the counters are read. A part of the program, showing procedures for zeroing and reading the counters, is shown in Appendix C.3.

## 5.3 Results

The results of the two analyses are presented in the two following subsections.

### 5.3.1 Build Benchmark

The times measured when building Xen in native Linux and Xenlinux are shown in Table 5.2. The corresponding bar chart is shown in Figure 5.1.

| *Operation* | *Count* |
|---|---|
| `rfi` | 228811 |
| `rsm.dt` | 600 |
| `ssm.dt` | 303 |
| `cover` | 6 |
| `itc.d` | 142387 |
| `itc.i` | 72976 |
| `=tpr` | 1 |
| `tpr=` | 3 |
| `eoi` | 2 |
| `itm=` | 2 |
| `thash` | 3 |
| `ptc.ga` | 19709 |
| `=rr` | 3 |
| `rr=` | 26 |
| Total | 10610822 |

Table 5.3: Measured number of para-virtualized operations executed.

The performance between difference **native-linux** and **native-xenlinux** is small. **native-xenlinux** takes 1.5 % more time than **native-linux**. This shows that transparent para-virtualization is quite feasible on Xen/ia64. The results also show that **para-xenlinux** has a reasonably low overhead compared to **native-linux**, amounting to 4.9 %. However, similar benchmarks performed by Dan Magenheimer do not show as high overhead. Finally, **privified-linux** takes 38 % more time than **native-linux**, showing that this method is at the moment not a very feasible solution. The measurements also verify that most of the overhead from virtualization in **privified-linux** come from running in kernel mode, in which overhead is as high as 567 %.

### 5.3.2 Instrumentation

Counting privileged operations during kernel compilation, the following numbers show some hotspots with possibilities for improvement. Table 5.3 shows a number of para-virtualized operations of particular interest for locating para-virtualization overhead. The corresponding bar chart is shown in Figure 5.2. These para-virtualized operations consume more CPU cycles than their corresponding native operations. As an example, the most numerous para-virtualized operation is the `rfi` instruction. Natively, this instruction changes a lot of state in the CPU (see Appendix B.2). In Xen, this involves changing a lot of state in the virtual CPU, which lies in memory. Changing the virtual CPU in memory amounts to more CPU cycles than a native `rfi`, which only takes around 13 cycles on the Itanium 2 architecture [cpi05].

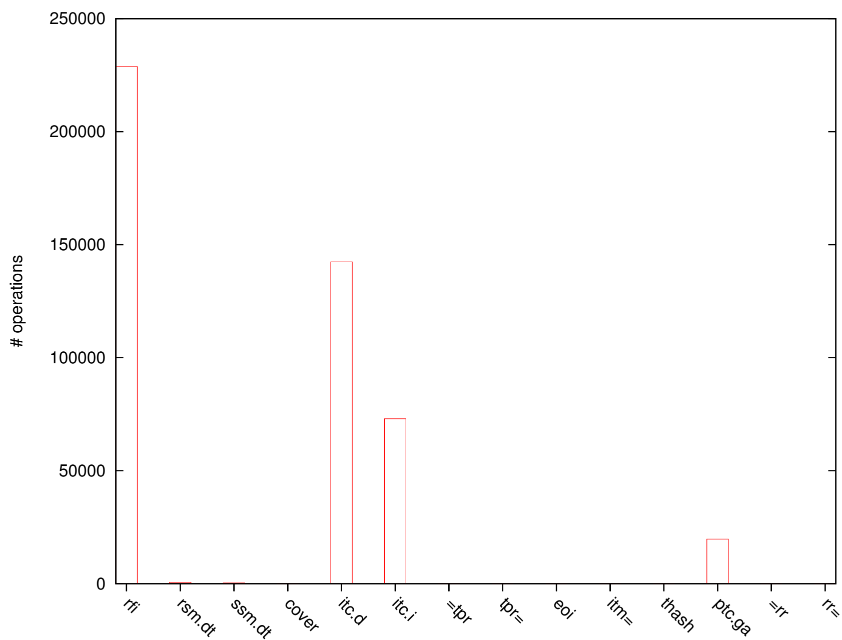


Figure 5.2: A chart over the number of para-virtualized operations executed.

# Chapter 6

# Related Work

A lot of interesting work has been happening both around Xen and within the general topic of virtualization. One project of particular relevance to Xen/ia64 is *vBlades* [MC04]. The vBlades VMM runs on IA-64 hardware and successfully virtualizes the underlying hardware. At the same time, it supports para-virtualized guest OSs. Knowledge from the vBlades project has been transferred to the Xen/ia64 project. The vBlades project has been shelved in favor of the Xen/ia64 project.

The L4Ka project [l4k05b] at the University of Karlsruhe is doing a lot of interesting research around virtualization. *Pistachio* is a microkernel which runs on many computer architectures, among them IA-64. *Marzipan* is a VMM based on the Pistachio microkernel, which also runs on IA-64. Most interesting, perhaps, is the *Pre-virtualization with Compiler Afterburning* project [l4k05a]. As illustrated in Figure 3.1, Pre-virtualization refers to the technique of making the necessary changes to a guest kernel for virtualization at compile-time. It substitutes *virtualization-sensitive operations* statically in the kernel, and also detects sensitive memory operations and substitutes them with calls to the VMM. This project is under active development, and the developers do not consider the software to be mature [l4k05b].

*vNUMA* [CH05] is another virtualization project for the IA-64 architecture. It seeks to make programming and managing for a distributed memory platform, such as a cluster, easier. This is achieved by virtualizing a shared memory platform on the distributed memory platform.

The most exciting development lately within virtualization, however, is Intel and AMD's introduction of virtualization extensions on their processors. These extensions aim to solve one of the main problems with Xen's para-virtualization approach: the guest OS needs to be modified. At the same time, efficiency is maintained. Intel introduces the VT extensions, which is discussed in Section 6.1.

An interesting approach to Linux virtualization on a mainframe is taken by *vServer* [P04̈]. It divides a single Linux execution environment into several separate execution environments, called Virtual Private Servers (VPS). VPSs are similar to virtual machines, except that only certain aspects of the native Linux execution environment are virtualized. The project takes a more pragmatic approach than Xen by only virtualizing the mechanisms required to provide execution environments for a set of traditional server applications, such as, for

instance, web servers.

Xen has also been ported to the 64-bit extended version of the x86 architecture, the *x86_64* architecture. This port has progressed far, and is more mature than the IA-64 port. Still, essential features, such as support for SMP, are lacking. Xen/x86_64 can be regarded as a near future 64-bit virtualization technology, which is attractive also for this architecture, due to the increased address space. One interesting challenge that this project has faced, is that the x86_64 architecture does not support segmentation in 64-bit mode. This means that the privilege separation scheme of x86 paging (see Section 2.2.1), consisting of the two privilege levels "User" and "Supervisor" is used. Effectively, ring compression in the x86_64 hardware thus forces guest OSs to run in the "User" ring.

## 6.1 Intel Virtualization Technology

With the VT extensions, the problems of ring aliasing are alleviated. This means that guest OSs do not need to be modified, and thus closed source OSs, such as Microsoft Windows, can run in a virtual execution environment.

The problem of ring aliasing is solved by introducing a new execution environment more privileged than ring 0, referred to as *VMX Root*. This new environment has the rings 0 through 3 just as traditional processors, and, indeed, programs that run in ring 0 in VMX Root behave as if running in ring 0 in a non-VT Intel processor. The VMX Root environment is dedicated to the hypervisor. The *VMX Nonroot* environment, however, is dedicated to the guest OSs. It also gives a similar execution environment with rings zero through three, except, when the guest OS executes in ring zero, control is handed over to the hypervisor. The VMX mechanism is illustrated in Figure 6.1. When a guest VM, running in VMX Nonroot, tries to enter ring zero, `VMEXIT` is executed, and control is passed over to the hypervisor running in VMX Root. In effect, the guest OS is deprivileged without ring compression, thus alleviating the problem of ring aliasing, while, at the same time, maintaining isolation.

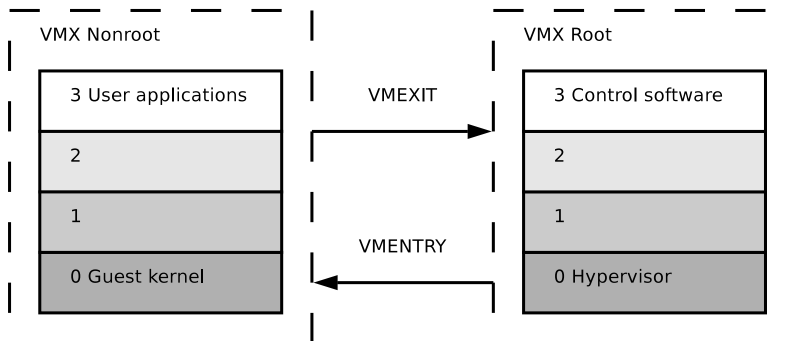


Figure 6.1: VT's VMX Root and Nonroot execution environments.

# Chapter 7

# Conclusion

In terms of virtualization, the IA-64 architecture is an "uncooperative" architecture. This forces the use of para-virtualization in order to achieve good performance. However, para-virtualization requires a certain amount of labor. Thus, trying to make as little as possible changes in the guest OSs' source code necessary, allows OSs to be para-virtualized with less effort. Making a small code impact on the native Linux kernel through optimized para-virtualization is an interesting approach, but some direct changes to the source code seem necessary in order to achieve close-to-native performance. Transparent para-virtualization may alleviate some of the administrative problems of using different OS kernel binaries for native OSs and para-virtualized OSs. At the same time, performance analyses in this thesis show that the performance impact from running a transparently para-virtualized binary on physical hardware is low.

Certain features of the IA-64 architecture make it more "cooperative" than the x86 architecture. One uncooperative attribute of the x86 architecture is the TLB, which needs to be flushed on every change of address space. In the IA-64 architecture, however, each TLB may individually be invalidated for an address space, thus eliminating the need for flushing. Region registers further allow special regions in memory, which can be used, for instance, to implement the physical memory abstraction, metaphysical memory. The EFI provides a standardized abstract interface to certain hardware funtions, which makes it easier to virtualize access to these functions. Also, the IA-64 PMU is, as opposed to x86 monitoring functions, documented and standardized, and allows the monitoring of Xen, in order to find performance hotspots.

The IA-64 port of Xen presents itself as a viable future virtualization technology for Itanium machines. Future Intel architectures will have better support for virtualization, and Xen is a good candidate for utilizing the virtualization capabilities of the future architectures. The performance analyses in this thesis show that Xen/ia64 is at present time not mature enough for production use. Benchmarks demonstrate that the overhead of using Xen/ia64 is not very high. However, some parts of Xen still remain to be ported to IA-64, and sporadical failures during execution reveal that some work remains before Xen/ia64 nears production quality.

# Appendix A

# Registers

## A.1 Some Essential x86 Registers

| *Mnemonic* | *Description* |
|---|---|
| ESP | Stack pointer; points to the top of the Kernel Mode Stack in Linux/x86. |
| CS | *Code Segment*—points to the code segment of the currently executing task. |
| DS, ES, FS, GS | *Data Segments*—point to the data segments of the currently executing task. |
| EAX, EBX, ECX | General-purpose register used for general storage of operands and results in arithmetic and logic operations, and memory pointers. |
| EIP | Points to the executing instruction relative to `cs`. |
| CR0 | *Control Register 0*—used to control operating mode and states of the CPU. It contains the TS flag, which is set at every context switch in order to indicate for the OS that FPU state may have changed. |
| CR3 | Points to the currently executing task's page directory. |
| GDTR | *Global Descriptor Table Register*—points to the current GDT. This register is protected. |
| LDTR | *Local Descriptor Table Register*—points to the current LDT. This register is protected. |

## A.2 Some Essential IA-64 Registers

| *Mnemonic* | *Field* | *Description* |
|---|---|---|
| PSR | | The *Processor Status Register* manages certain aspects of the execution environment of the current task, such as whether interrupts are enabled or not. Accessing this register is a privileged operation. |
| | DT | The dt flag specifies whether memory is accessed virtually or physically. |
| | IC | *Interrupt Collection*—this flag specifies whether interrupts trigger the storing of processor state. |
| | I | *Interrupt Bit*—if set, unmasked external interrupts will interrupt the processor and be handled by the external interrupt handler. Otherwise, the processor will not be interrupted by external interrupts. |
| | CPL | *Current Privilege Level*—this 2-bit field specifies the privilege level of the currently executing process. |
| | DFH | *Disabled Floating-point High register set*—specifies whether access to the high floating point register set, registers F32 through F127, produces a *Disabled Floating-Point Register* fault. |
| IFS | | The *Interruption Function State* register is used by a `rfi` instruction to reload the current register stack frame. |
| IPST | | *Interruption Processor Status Register*—has the same format as PSR, only, it is used to restore CPU state after an `rfi` instruction. |
| IP | | *Instruction Pointer*—points to the address of the next instruction bundle to execute. |
| IIP | | *Interruption Instruction Bundle Pointer*—used to restore IP after an `rfi` instruction. |
| IFA | | *Interruption Faulting Address*—contains the address of the instruction that has caused an exception. |
| ISR | | *Interruption Status Register*—contains information about what has caused an exception. |
| IVA | | *Interruption Vector Address*—keeps the address of the IVT. |
| IHA | | *Interruption Hash Address*—keeps the address of the VHPT which is used to resolve translation faults. |
| IRR0–IRR3 | | *External Interruption Request Registers*—shows the vectors of pending interrupts. Each non-reserved bit in the collective amount of bits in these registers correspond to one interrupt vector. |
| BSP | | *Backing Store Pointer*—contains the pointer to the location in memory which the current register stack frame would be stored if that were necessary. |
| SP (R12) | | *Stack Pointer*—by convention, this register is used to point to the Kernel Mode Stack. It is part of the General Register File. |
| IIM | | *Interruption Immediate*—used for diagnosing the reason for some faults. |
| F0–F127 | | *Floating point registers*—used for floating point calculation. |

# Appendix B

# Instruction Set Architectures

## B.1 Some Essential x86 Instructions

| *Mnemonic* | *Description* |
|---|---|
| `lgdt` | *Load Global Descriptor Table*—loads an address into the GDTR register. This instruction is protected. |
| `int` | *Call to Interrupt Procedure*—generates an interrupt in the processor. An immediate parameter points to the interrupt procedure which is to be called. |
| `mov` | *Move*—copies the value of a source operand to a target operand. The source operand may be an immediate value, a register or a memory location. |
| `esc` | *Escape*—allows interaction with the x87 FPU. |

## B.2 Some Essential IA-64 Instructions

| *Mnemonic* | *Parameter* | *Description* |
|---|---|---|
| `rfi` | | Returns the hardware context to that prior to the interruption. |
| `tak` | | Returns the protection key for a data TLB entry. |
| `cover` | | Allocates a new register stack frame with zero registers. Updates BSP to the location of a potential next register stack frame in memory. |
| `break` | *imm* | Triggers a *Break Instruction Fault*. *imm* is loaded into the IIM register. |
| `epc` | | *Enter Privileged Code*—increases the CPL to the privilege level given by the TLB entry for the page containing the `epc` instruction. |

# Appendix C

# Source Code

The program code of Xen is quite extensive, and, since it is open source licensed, it may be downloaded from the web. The latest Xen/ia64 and Xenlinux/ia64 development versions are downloaded using Mercurial [mer05] from `http://xenbits.xensource.com/ext/xen-ia64-unstable.hg` and `http://xenbits.xensource.com/ext/xenlinux-ia64-2.6.12.hg`, respectively. Also, nightly snapshots are taken and distributed in the following package: `http://www.cl.cam.ac.uk/Research/SRG/netos/xen/downloads/xen-unstable-src.tgz`.

Some example code snippets are shown in the following appendices.

- Appendix C.1 shows how an interdomain event channel is established from a domain.
- Appendix C.2 shows the hypervisor's handling of hypercalls after a `break` instruction.
- Appendix C.3 shows two procedures using the interface to the privileged operation counters in the hypervisor.
- Appendix C.4 shows the virtualization of a privileged `mov` instruction which targets the PSR register.

## C.1 Event Channel Hypercall

```
void bind_interdomain(int dom1, int dom2, int port1, int port2){
    evtchn_op_t op;
    evtchn_op_t* opp;
        op.cmd = EVTCHNOP_bind_interdomain;
        op.u.bind_interdomain.dom1  = dom1;
        op.u.bind_interdomain.dom2  = dom2;
        op.u.bind_interdomain.port1 = port1;
        op.u.bind_interdomain.port2 = port2;
    opp = &op;

    asm volatile("ld8 r3=%0; "
            :: "m" (opp)
            : "r3");
```

```
    asm volatile("mov r2=%0; break %1;"
            :: "i" (EVENT_CHANNEL_OP), "i" (BREAKIMM)
            : "r2", "r8", "memory");
}
```

## C.2 Hypercall Handlers in the Hypervisor

```
int
ia64_hypercall (struct pt_regs *regs)
{
        struct vcpu *v = (struct domain *) current;
        struct ia64_sal_retval x;
        unsigned long *tv, *tc;

        switch (regs->r2) {
            case FW_HYPERCALL_PAL_CALL:
                //printf("*** PAL hypercall: index=%d\n",regs->r28);
                //FIXME: This should call a C routine
                x = pal_emulator_static(regs->r28);
                if (regs->r28 == PAL_HALT_LIGHT) {
                        do_sched_op(SCHEDOP_yield);
                        //break;
                }
                regs->r8 = x.status; regs->r9 = x.v0;
                regs->r10 = x.v1; regs->r11 = x.v2;
                break;

    (...)

            case __HYPERVISOR_dom_mem_op:
#ifdef CONFIG_VTI
                regs->r8 = do_dom_mem_op(regs->r14, regs->r15,
                            regs->r16, regs->r17, regs->r18);
#else
                /* we don't handle reservations; just return success */
                regs->r8 = regs->r16;
#endif
                break;

            case EVENT_CHANNEL_OP: // Evtchn hypercall
                evtchn_op_t* op = regs->r3;
                do_event_channel_op(op);
            default:
                printf("unknown hypercall %x\n", regs->r2);
                regs->r8 = (unsigned long)-1;
        }
        return 1;
```

```
}
```

## C.3 Privileged Operation Counters

```
long get_privop_counts(char *s, long n)
{
        register long r32 asm("in0") = (unsigned long)s;
        register long r33 asm("in1") = n;
        register long r8 asm("r8");
        asm volatile("mov r2=%0; break %1;"
                :: "i" (GET_PRIVOP_CNT_HYPERCALL), "i" (BREAKIMM)
                : "r2", "r8", "memory");
        return r8;
}

long zero_privop_counts(char *s, long n)
{
        register long r32 asm("in0") = (unsigned long)s;
        register long r33 asm("in1") = n;
        register long r8 asm("r8");
        asm volatile("mov r2=%0; break %1;"
                :: "i" (ZERO_PRIVOP_CNT_HYPERCALL), "i" (BREAKIMM)
                : "r2", "r8", "memory");
        return r8;
}
```

## C.4 Paravirtualization

```
IA64FAULT vcpu_get_psr(VCPU *vcpu, UINT64 *pval)
{
        UINT64 psr;
        struct ia64_psr newpsr;

        // TODO: This needs to return a "filtered" view of
        // the psr, not the actual psr.  Probably the psr needs
        // to be a field in regs (in addition to ipsr).
        __asm__ __volatile ("mov %0=psr;;" : "=r"(psr) :: "memory");
        newpsr = *(struct ia64_psr *)&psr;
        if (newpsr.cpl == 2) newpsr.cpl = 0;
        if (PSCB(vcpu,interrupt_delivery_enabled)) newpsr.i = 1;
        else newpsr.i = 0;
        if (PSCB(vcpu,interrupt_collection_enabled)) newpsr.ic = 1;
        else newpsr.ic = 0;
        *pval = *(unsigned long *)&newpsr;
```

```
        return IA64_NO_FAULT;
}
```

# Bibliography


[And04] Rune J. Andresen, *Virtual machine monitors*, http://openlab-mu-internal.web.cern.ch/openlab-mu-internal/Documents/Reports/Technical/Summer%20Students/vmm.pdf, Aug 2004.

[B+03] Paul Barham et al., *Xen and the art of virtualization*, SOSP'03, October 2003, http://www.cl.cam.ac.uk/Research/SRG/netos/papers/2003-xensosp.pdf.

[BA04] Håvard K. F. Bjerke and Rune J. Andresen, *Virtualization in clusters*, http://www.idi.ntnu.no/~havarbj/clust_virt.pdf, Nov 2004.

[BB03] Håvard K. F. Bjerke and Christoffer Bakkely, *Multiprocessor architecture overview*, http://www.idi.ntnu.no/~havarbj/dmark/index.html, April 2003.

[BC02] Daniel P. Bovet and Marco Cesati, *Understanding the Linux kernel*, second ed., O'reilly & Associates, Inc., Dec 2002.

[Bje04] Håvard K. F. Bjerke, *Grid survey*, http://www.idi.ntnu.no/~havarbj/grid_survey/grid_survey.pdf, Aug 2004.

[boc05] *Bochs*, http://bochs.sourceforge.net/, July 2005.

[CH05] Matthew Chapman and Gernot Heiser, *Implementing transparent shared memory on clusters using virtual machines*, USENIX 2005, General Track, April 2005, pp. 383–386.

[cpi05] *Gelato at UNSW WiKi*, http://www.gelato.unsw.edu.au/IA64wiki/, July 2005.

[DC99] Kenneth J. Duda and David R. Cheriton, *Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler.*, SOSP, 1999, pp. 261–276.

[E+98] Richard J. Enbody et al., *Performance monitoring in advanced computer architecture*, Workshops on Computer Architecture Education 1998, Jun 1998.

[EM00] Stéphane Eranian and David Mosberger, *The linux/ia64 project: kernel design and status update*, http://www.hpl.hp.com/techreports/2000/HPL-2000-85.pdf.

[Era05] Stéphane Eranian, *Perfmon*, http://www.hpl.hp.com/research/linux/perfmon/, June 2005.

[F+04a] Keir Fraser et al., *Reconstructing I/O*, http://www.cl.cam.ac.uk/TechReports/UCAM-CL-TR-596.pdf.

[F+04b] Keir Fraser et al., *Safe hardware access with the Xen virtual machine monitor*.

[FK] Ian Foster and Carl Kesselman (eds.), *The GRID: Blueprint for a new computing infrastructure*.

[G+05] Charles Gray et al., *Itanium—a system implementor's tale*, USENIX 2005, General Track, Apr 2005, pp. 265–278.

[HC05] Hannelore Hämmerle and Nicole Crémel, *LHC grid tackles multiple service challenges*, CERN Courier **45** (2005), 15.

[Hud05] Paul Hudson, *It's all about Xen*, Linux Format **67** (2005), 52–59.

[Inta] Intel, *IA-32 intel architecture software developer's manual*, http://developer.intel.com/design/pentium4/manuals/index_new.htm.

[Intb] Intel, *Intel itanium architecture software developer's manual*, http://www.intel.com/design/itanium/manuals/iiasdmanual.htm.

[Jar99] Sverre Jarp, *Ia-64 architecture: A detailed tutorial*, http://cern.ch/sverre/IA64_1.pdf, November 1999.

[K+04] Ron Kalla et al., *IBM Power5 chip: A dual-core multithreaded processor*, IEEE Micro **24** (2004), 40–47.

[KF91] Nancy L. Kelem and Richard J. Feiertag, *A separation model for virtual machine monitors*, Research in Security and Privacy, IEEE, May 1991, pp. 78–86.

[l4k05a] *Afterburning and the accomplishment of virtualization*, http://l4ka.org/projects/virtualization/afterburn/whitepaper.pdf, April 2005.

[l4k05b] *The l4ka project*, http://l4ka.org/, June 2005.

[lhc05] *What's next at cern?*, http://public.web.cern.ch/Public/Content/Chapters/AboutCERN/CERNFuture/WhatLHC/WhatLHC-en.html, June 2005.

[MC04] Daniel J. Magenheimer and Thomas W. Christian, *vBlades: Optimized paravirtualization for the Itanium processor family*.

[ME02] David Mosberger and Stéphane Eranian, *IA-64 Linux kernel design and implementation*, Prentice Hall, 2002.

[mer05] *Mercurial distributed SCM*, http://www.selenic.com/mercurial/, July 2005.

[Moo65] Gordon E. Moore, *Cramming more components onto integrated circuits*, Electronics **38** (1965).

[msv05] *Microsoft Virtual PC*, http://www.microsoft.com/windows/virtualpc/default.mspx, July 2005.

[Mun01] Jay Munro, *Virtual Machines & VMware, part I*, http://www.extremetech.com/article2/0,1558,10403,00.asp.

[PÖ4] Herbert Pötzl, *Linux-vserver technology*, http://linux-vserver.org/Linux-VServer-Paper.

[Pet05] Mikael Pettersson, *Perfctr*, http://www.csd.uu.se/~mikpe/linux/perfctr/, June 2005.

[qem05] *Qemu*, http://fabrice.bellard.free.fr/qemu/, July 2005.

[Ros04] Mendel Rosenblum, *The reincarnation of virtual machines*, Queue **5** (2004), 34–40.

[Sha05] Stephen Shankland, *Xen lures big-name endorsements*, CNET News.com (2005), http://news.com.com/Xen+lures+big-name+endorsements/2100-7344_3-5581484.html.

[sim05a] *Simics*, http://www.virtutech.com/, July 2005.

[sim05b] *Simplescalar*, http://www.simplescalar.com/, July 2005.

[Sin04] Amit Singh, *An introduction to virtualization*, http://www.kernelthread.com/publications/virtualization/.

[slc05] *Cern linux pages*, http://linux.web.cern.ch/linux/, June 2005.

[Smo02] Mark Smotherman, *Understanding EPIC architectures and implementations*, ACMSE2002, Apr 2002.

[Tur02] Jim Turley, *64-bit CPUs: What you need to know*, http://www.extremetech.com/article2/0,1558,1155594,00.asp, February 2002.

[U+05] Rich Uhlig et al., *Intel virtualization technology*, Computer (2005), 48–56.

[vmw05] *Vmware*, http://www.vmware.com/, July 2005.

[W+02] Andrew Whitaker et al., *Denali: Lightweight virtual machines for distributed and networked applications*, Tech. report, University of Washington, 2002, http://denali.cs.washington.edu/pubs/distpubs/papers/denali_usenix2002.pdf.

[win05] *Wine HQ*, http://www.winehq.com/, July 2005.

[Xen05] The Xen Team, *Xen interface manual*, 2005, Xen programming interface documentation—LaTeX source files included in the Xen development source distribution.