Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Masters Thesis

VMkit
A lightweight hypervisor library for Barrelfish

by
Raffaele Sandrini

Due date
2 September 2009

Advisors:
Simon Peter, Andrew Baumann, and Timothy Roscoe

ETH Zurich, Systems Group
Department of Computer Science
8092 Zurich, Switzerland

Abstract

Many virtualization solutions have been designed and implemented. They make use of a diverse set of different methods to achieve the secure multiplexing of physical hardware among multiple operating systems. Some are tailored to specific use cases, others aim to provide the widest possible spectrum of applications. Some solutions require the guest to be modified before or during the execution.

VMkit is a virtual machine monitor for the x86-64 architecture, integrated into the Barrelfish operating system. It allows unmodified guest operating systems to be run. It makes use of the virtualization extensions available in recent x86-64 implementations. VMkit is designed to be easy to implement and extend. The different components of the virtual machine are separated to achieve a high level of security.

An evaluation shows that VMkit's performance is comparable to existing solutions of similar architecture, while achieving increased system security and run Linux as guest operating system. Its implementation does not yet provide the same features as other publicly-available solutions, but is functional enough to perform real-world tests.

Contents

Chapter 1

Introduction

1.1 Problem and Motivation

Personal computers available today are powerful enough to run multiple operating systems simultaneously inside *virtual machines*. Due to this increased performance there are many virtual machine solutions available today, especially for the PC platform with x86 CPUs. Industry seeks such solutions to cope with the cost and energy requirements an increased number of computers may impose. The use of virtual machines can improve the utilization of such computers and therefore reduce these costs.

There is a broad field of application for virtual machines. They may be used to separate a heterogeneous set of different services from each other. Different services may have different operating system or configuration requirements. Such services may be hard or impossible to combine within a single operating system on the same computer. Using virtual machines, one can use the best fit for software, configuration and operating system for each different group of services or tasks.

The PC platform is able to support a very heterogeneous set of different devices. There is basically no special requirement for a device to run inside of a PC other than the ability to be attached to one of the platform's main buses, such as PCI. Therefore, another use case for virtual machines is driving the hardware of the underlying system. This can be especially useful in operating system research and development. One uses a virtual machine capable of running an operating system which is able to drive the hardware available in many personal computers. The development team can reduce the effort needed to write numerous drivers for many different devices available. They merely have to include such a virtual machine within their operating system in development.

Today individuals also use virtual machines on their desktop computer just to try new operating systems or use specific applications which are only available on specific operating systems. It is more convenient to set up a virtual machine rather than run the operating system exclusively on the computer. A potential user can stick to the environment he or she prefers and use the virtual machine *in parallel*.

The main challenge when building virtual machines is resource separation. All operating systems want to use the same resources, such as CPUs, memory, hard disks and networking cards, and it is the job of the underlying system to multiplex these resources among all virtual machines in a secure and fair fashion. This should be accomplished with minimal performance overhead for resource multiplexing and virtual machine control.

1.2 Aim

The aim of this work is to create a virtual machine monitor. To this end, current implementations of virtual machine monitors shall be reviewed, and the methods used analyzed and described. VMkit shall be oriented towards simplicity, ease of implementation, and security always having performance in mind. It shall be able to run on top of Barrelfish [6] and be capable of running at least Linux in its virtual machine.

1.3 Overview

Chapter two introduces the different terms involved when talking about virtual machines and different techniques to build them. It also summarizes some important hardware aspects. Chapter three analyzes other implementations of virtual machines done previously. It categorizes them and relates them to the techniques introduced in chapter two. Chapter four offers some background on Barrelfish, especially those parts which are important to understand the interfaces used within VMkit. In chapter five the global approach and the design is shown. It describes how the virtual machine is laid out and what the properties with respect to the background on virtual machines are. Chapter six describes the implementation of the whole system within Barrelfish. An evaluation of the implementation is presented in chapter seven and the work concludes in chapter eight where the implementation is rated according to the evaluation and the aim.

Chapter 2

Virtual Machines

The term *virtual machine* is used to describe different concepts. Not all of them are relevant for this work. It is used by certain platform environments such as Java or .NET, to describe the piece of software which translates instructions from special, mostly hardware independent byte code to instructions the machine, on which the virtual machine is running, can understand.

Another use of the virtual machine term is to describe a software system to provide an efficient, isolated duplicate of a real machine. A simulation of all relevant parts necessary to enable software to run within this machine. It can be either a perfect duplicate of the underlying hardware, or can simulate another machine. This type of virtual machine will be the focus for the rest of this work.

One of the first use cases for virtual machines was mainframe multiplexing. These computers were expensive and an organization was not able to purchase many of them. Virtual machines were used on top of these mainframes to improve their utilization.

The field of application for virtual machines is broader today. The aim moved away from the pure duplication of a certain machine to an abstraction of physical hardware in general. Virtual machines tend to provide virtual hardware which is able to run in many different environments and is easy to simulate and to use for the operating system running within the virtual machine.

2.1 Structure and Terminology

This section introduces key virtual machine structure concepts.

First of all, a *virtual machine (VM)* describes an entire system enabling the execution of arbitrary machine code within a confined, protected environment of a real machine. It consists of at least one monitor and one guest.

The *host* is the machine on which the virtual machine is running. The host may be a real machine or another virtual machine.

The *guest* is the software running within the virtual machine. In the most general case it may be code of an arbitrary machine, of which only the targeted real architecture is known to the virtual machine. Generally, this will be a setup which would without modification also run on a corresponding real machine.

A *virtual machine monitor (VMM)*, often shortened to *monitor*, is the controlling part of the virtual machine. It describes the piece of software managing the virtual machine guest and all the virtual devices the virtual machine offers. It is responsible for the security of the host and the guest, i.e. it has to ensure no unintended state exchange takes place between them. The virtual machine monitor is often also called *hypervisor*. Operating system kernels are often called *supervisor* and since the virtual machine monitor resides logically below the guest operating system the term hypervisor is used to describe it.

Two important operations will come up throughout this work. They are called *VM enter* and *VM exit*. When the monitor decides to run the guest within a virtual machine it will perform a VM enter. This operation will switch from host execution to guest execution and is called a *world switch*. When the guest is currently running and encounters a situation it cannot or must not handle, it will stop running and the monitor will be invoked. This is the opposite operation to VM enter and is called VM exit.

There are more terms which need defining when talking about virtual memory. When the monitor runs on a machine offering the use of virtual memory and the virtual machine also provides virtual memory, then there are four levels of address spaces which must be considered. Starting at the top layer there is *guest-virtual memory*, then *guest-physical memory*. The same is true for the host, there is *host-virtual memory* and *host-physical memory*. The terms for themselves are self explanatory. One just has to remember, there are in fact four levels of address spaces when talking about possible solutions to address memory.

The terms *virtualization* and *to virtualize* often describe a fuzzy meaning of running some virtual machine on top of another. In this work a clean distinction is drawn between *to virtualize* and *to emulate*. To virtualize some piece of hardware means granting multiple, unrelated clients controlled, direct hardware level access to the device, i.e. multiplexing it over different clients without them knowing it being multiplexed. To emulate some piece of hardware means running a piece of software that simulates the hardware to the clients, without them knowing they are not dealing with the real hardware. Virtual machine monitors often use both concepts for different virtual devices within the virtual machine they are providing.

2.2 Techniques

This section introduces key techniques used to build virtual machines.

One does not want to lose a lot of the real machine's performance through the virtualization of a new one. But this is not the only concern. A real machine may not be an ideal basis for constructing a virtual machine. It may have instructions which prevent the monitor from executing the virtual machine in a secure way. This and other correctness problems are addressed in several techniques for building a virtual machine and especially its monitor.

2.2.1 Trap and emulate

Virtual machines should act on their host as normal processes do. They should be protected from other processes and should protect and hide the entire host from the guest. Many hardware architectures achieve this protection through the introduction of privilege levels. Certain software, usually the operating system kernel, runs on a higher privilege level than the rest of the software. The architecture will trap from unprivileged into privileged software under certain well-defined conditions and to well-defined entry points within that privileged software. It is important that guests are executed in unprivileged mode, such that they will trap into system software once such conditions occur.

Trap and emulate describes a general technique used in virtual machines. It is, in one way or another, part of any virtual machine implementation. If the architecture running a guest experiences a condition which will leak or alter global CPU state or risk system security in any other way, then the execution of the guest will be stopped (VM exit) and its monitor invoked, hence a *trap*. The monitor will examine the exit reason and *emulate* the state changes to the guest without altering the host counterparts. The same technique is often used to simulate virtual hardware to the guest. The virtual machine performs a VM exit every time some virtual device register is accessed and the monitor will perform the simulation of the device and enter the virtual machine again.

On hardware architectures where all necessary instructions, i.e. instructions relevant to viewing and altering global system state and execution context, trap into system software, trap and emulate can be used as the technique to build virtual machines. If an architecture fails to trap in all necessary situations it may enable the guest to compromise the host security and the overall correctness of the system, therefore additional means, described in the next sections, must be employed to circumvent that problem.

2.2.2 Dynamic translation

Dynamic translation [2] is a technique where the guest code is altered dynamically, i.e. during run time, to bring it into a form suitable for running on the virtual machine. The guest should not notice these changes, therefore the semantics of the guest code must not be altered in a way it cannot function the same way it would without the changes. To perform the dynamic translation the guest code is analyzed on a basic block level, i.e. chunks of code which do not alter control flow. These blocks are then translated to code conforming with the rules the virtual machine enforces on guests.

There are different reasons for applying this method. When a hardware architecture does not allow pure trap-and-emulate virtualization, the guest needs to be modified to simulate a trap in all necessary situations not covered by hardware. This explicit VM exit is then handled by the monitor like a normal trap. It may also be used to improve the execution performance of the guest by replacing some low-level hardware access instructions by calls to the monitor to reduce the latency a full trap would incur.

A further use of dynamic translation is machine emulation. During machine emulation, all instructions are translated from the virtual machine architecture to the host architecture, i.e. compiling from one instruction set into another instruction set. This can be done on a basic block level to minimize the translation effort. Once a basic block is translated, it may be run many times without further modification. Using this method, the performance is increased over systems using guest code interpretation. During interpretation the guest code, when executed, is not read by the hardware but by the interpreter which is a

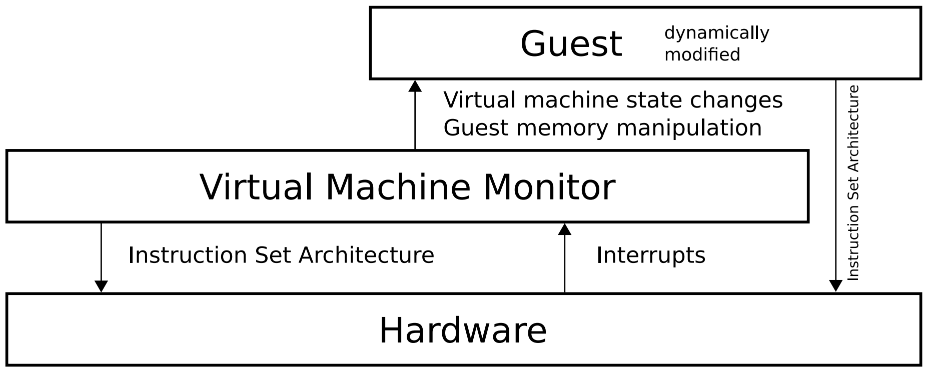
Guest
dynamically modified
Virtual machine state changes
Guest memory manipulation
Virtual Machine Monitor
Instruction Set Architecture
Interrupts
Instruction Set Architecture
Hardware

Figure 2.1: The structure of a virtual machine using dynamic translation. The guest is not aware of the the monitor's presence. There is no explicit communication between those entities. The monitor modifies the guest's memory such that it can safely execute instructions directly on the underlying hardware.

piece of software taking each guest instruction and performing the requested guest action on its behalf on the physical hardware.

2.2.3 Static translation

The same way a guest can be modified during run time to conform to the rules of the virtual machine, it can be modified during compile time. A guest can be written in such a way that it will run correctly on a certain virtual machine. This technique is often also called *paravirtualization* [5]; it refers to the fact that the guest, knowing it is executed on a certain virtual machine, may take advantage of the special circumstances a virtual machine offers. The same way user space applications can be written to conform to operating system interfaces, operating systems can be written to conform to certain virtual machine types. These virtual machines provide their guests a interface, usually called *hypercalls*, to access the virtual hardware resources offered by the virtual machine. This interface usually is considerably simpler and higher level compared to the complex interface physical computer hardware offers. If the underlying hardware does not allow complete trap and emulation, the virtual machine has to provide a set of hypercalls the guest can execute instead of using the unsafe instruction.

These types of virtual machines often achieve improved I/O performance because they know that virtual hardware is used and may optimize for that fact. However, if some guest is not available in source code form, it might pose a serious problem to port that particular guest to a virtual machine using paravirtualization.

2.2.4 Guest memory management

The virtual machine must simulate the same memory environment to the guest as it could expect running on real hardware. This includes at least a region of real memory which may be used freely and perhaps some memory-mapped

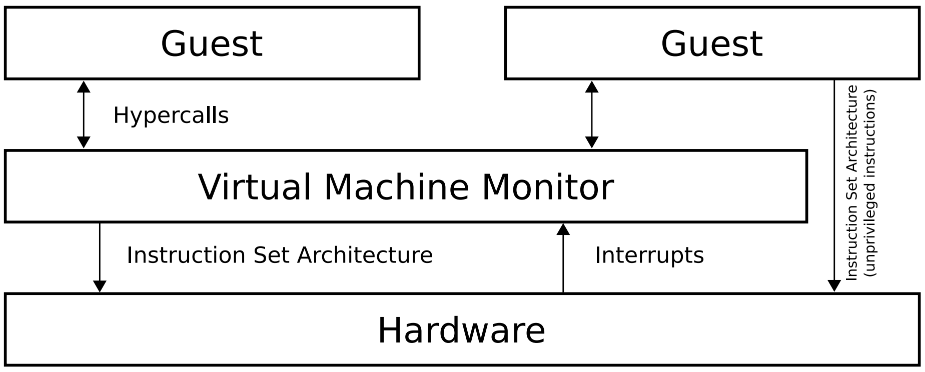
Guest
Guest
Hypercalls
Virtual Machine Monitor
Instruction Set Architecture
Interrupts
Instruction Set Architecture (unprivileged instructions)
Hardware

Figure 2.2: The structure of a virtual machine using static translation. Guests use explicit communication with the monitor through a special parvirtualization interface.

device registers. The guest operating system will manage this memory by generating specific page tables translating guest-virtual addresses to guest-physical addresses within these regions. It is up to the virtual machine monitor to ensure that the guest issues only proper memory accesses and to trigger hardware emulation code if some virtual device register is accessed. The overall performance of the virtual machine depends on the CPU and the memory system being virtualized to the greatest extend, i.e. memory accesses should trigger as few VM exits as possible. To make this happen, the monitor must know about the implications of all possible guest-memory accesses in advance, or needs a security mechanism which triggers only on relevant memory accesses. There are two major techniques dealing with this issue: *shadow paging* and *nested paging.*

Shadow paging

Shadow paging is a technique used by many virtual machine monitor implementations. It does not need any additional hardware support beyond normal paging. For every page table the guest operating system creates, the monitor maintains a corresponding shadow page table. The guest must be modified or the hardware configured, to perform a VM exit, every time the guest performs a page table switch. The monitor will then walk the entire guest page table and check whether it violates host system security. Mappings pointing to guest-physical memory will be copied to the shadow page table, and their physical destination address will be modified to the corresponding address within host-physical address space. Mappings pointing to virtual hardware registers will be flagged to trap the CPU on access. The region where the original page table is located will also be flagged to trigger a page fault on write access to prevent the guest operating system from changing the seemingly active page table without the monitor noticing it. Finally, the monitor will instruct the hardware to use its newly created shadow page table instead of the guest version. The guest will not notice there there is actually a different page table active. If the guest tries to read the address of the currently active page table, the virtual machine will exit and the monitor will emulate the original location to the guest. The guest will normally create a separate page table for every user-space process it manages. Therefore, the monitor will end up maintaining at least as many shadow page tables as the guest runs user-space processes. It will walk the active guest page table on most guest context switches. This imposes a significant performance penalty. There are many optimization techniques such as using special data structures to store the different page tables, or making the page walker more intelligent [5], preventing it from walking the whole guest page table every time.

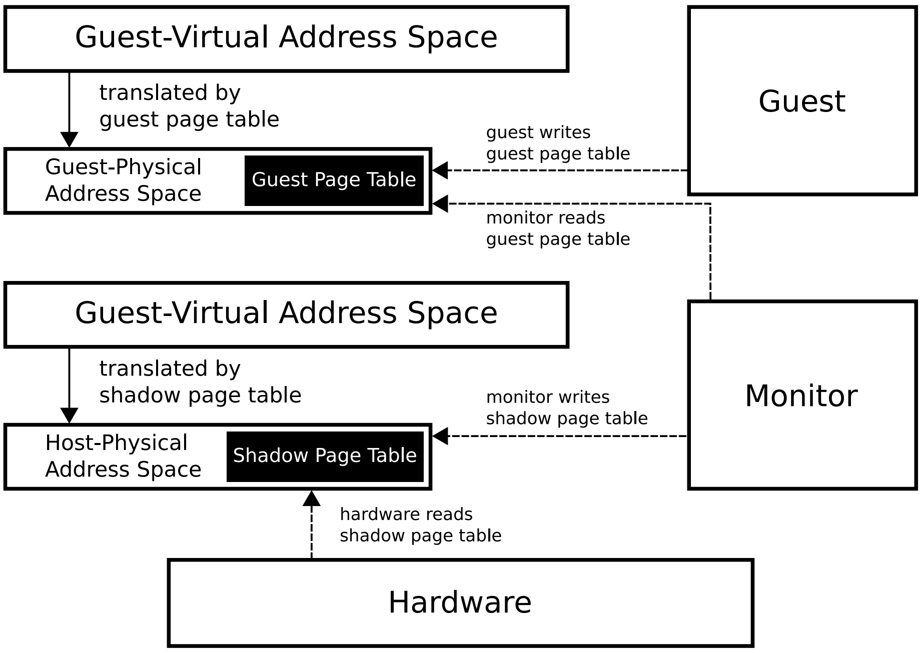
Guest-Virtual Address Space
translated by guest page table
Guest-Physical Address Space
Guest Page Table
Guest
guest writes guest page table
monitor reads guest page table
Guest-Virtual Address Space
translated by shadow page table
Host-Physical Address Space
Shadow Page Table
Monitor
monitor writes shadow page table
hardware reads shadow page table
Hardware

Figure 2.3: The shadow paging mechanism. The monitor reads the page table the guest has written and translates all the entries into a shadow page table. The hardware will then use this shadow page table to perform the actual address translation.

Nested paging

The other major method is nested paging [3, 29]. It requires specific hardware support. The hardware page table walker will translate the guest-virtual address via the guest page table to a guest-physical address. This address will then be further translated by the hardware walker via the nested page table to get the host-physical address. Using this method, the guest memory management and the protection of the monitor are two different separated tasks. Page faults within the guest page table can be forwarded to the guest directly. They are no different from page faults on real hardware. Nested page faults are handled by the monitor. It must check whether there was an access to illegal guest-physical memory or to a virtual device register. Guests never know about nested page faults and assume there was a normal memory access. Through this separation, the monitor needs never to walk the entire guest page table and check its correctness.

However, nested paging may have a negative influence on virtual-address translation. In the worst case, the number of memory accesses needed to resolve a particular guest-virtual address is squared compared to normal paging, because every access to the guests page table will result in a host-virtual address which may need to be resolved through the nested page table prior to use. However, the use of nested paging allows significant simplification inside the monitor, because the hardware takes responsibility for many of the additional memory operations.

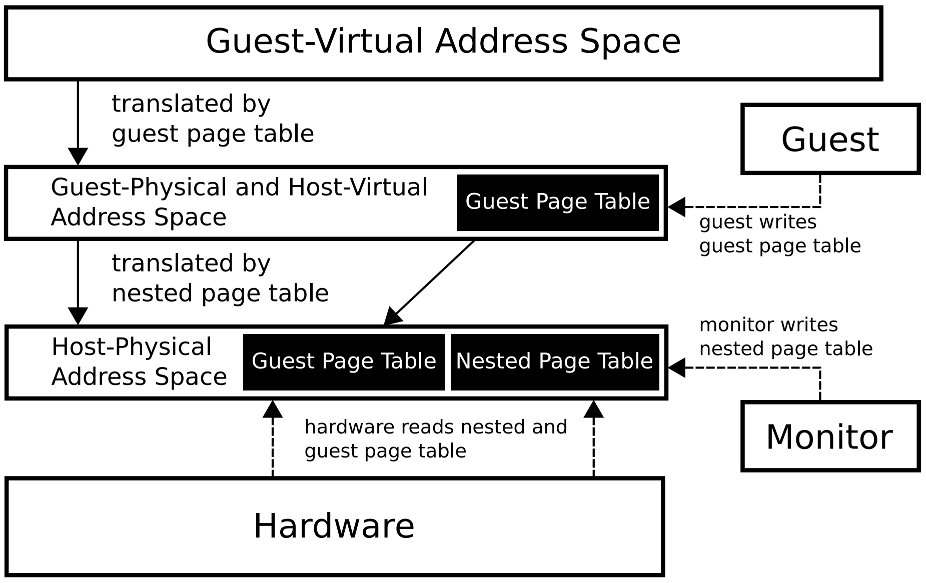
Guest-Virtual Address Space
translated by guest page table
Guest
Guest-Physical and Host-Virtual Address Space
Guest Page Table
guest writes guest page table
translated by nested page table
monitor writes nested page table
Host-Physical Address Space
Guest Page Table
Nested Page Table
hardware reads nested and guest page table
Monitor
Hardware

Figure 2.4: The nested paging mechanism. The hardware translates guest-virtual addresses first using the the guest page table and the resulting guest-physical address using the nested page table. The guest page table resides in guest-physical memory which is virtual to the host and therefore all accesses to the guest page table need first to be resolved by the nested page table.

2.2.5 Guest and monitor placement

There are several logically different ways an operating system kernel, a virtual machine monitor and a guest can coexist within the same host. The monitor can be entirely a part of an operating system kernel, i.e. run in privileged mode, or run within a user-space process. In the latter case the monitor can only allow the guest to execute unprivileged instructions. All remaining instructions have to be emulated to the guest, i.e. guest-kernel code. Also in the latter case, there is a choice whether the guest is joint or separated from the monitor with respect to the hardware protection domain. All these choices have influence on the performance and the security of the virtual machine.

It may improve performance to place the virtual machine monitor within the operating system kernel. Many kernels map their code to a special memory region inside the virtual address space which will never be accessible by any user space process. With this region free in all user-space processes, it can be mapped through every page table used within the system, eliminating the need to switch to and from a kernel address space on every system call. A monitor within the kernel could use this mechanism to avoid unnecessary context switching, since it would not be a single process anymore, but rather part of *every* process. However, placing the monitor inside the kernel gives it access to anything on the host machine. Therefore, the whole monitor would be a part of the trusted computing base (TCB). The TCB is the set of all software, which when compromised, may jeopardize the entire system. At a minimum, everything running inside the kernel is part of the TCB. The monitor will control the execution of a guest, hence a bug within the monitor could make it possible for the guest to exit its virtual machine and cause damage to the host. If the monitor is part of a user-space process, the host will be more secure from monitor malfunction. This extra security comes with the cost of additional context switches to and from the monitor.

With the monitor running in user space, the question remains whether the guest shall be joint with the monitor in the same process. When joint, the management of the guest's run-time environment, such as memory management, can be done within the monitor process and does not need further communication with a guest process. Furthermore, there is no need to switch contexts when transitioning between guest and monitor execution. Some architectures, such as the x86, need the TLB to be flushed on every context switch. The monitor needs protection from the guest, i.e. its memory should not be accessible from within the guest. This protection will be harder to accomplish without hardware support. Therefore, with joint monitor and guest, there is always the possibility for the guest to take over its monitor. This is especially critical if the monitor manages more than one guest.

Moving the monitor and the guest into separate domains has the advantage that the hardware will take care of protection between guest and monitor. An additional context switch is then required to enter and exit the guest. Basically, entering and exiting is the same thing as switching between regular processes on the host. The question remains whether there are many context switches between the host and the guest. Running a non-paravirtualized guest may cause many VM exits. For example, every time the guest accesses some of its virtual hardware, depending on the device, a considerable number of instructions accessing IO ports or memory-mapped registers may be expected. The virtual machine will exit on every such instruction, either through a memory fault, in the case of a memory mapped register, or through the interception of instructions such as `in` and `out` on the x86 architecture. Issuing such an instruction from the guest can be compared to executing a system call in a normal user-space process. However, a system call often encapsulates multiple such instructions that cause a virtual machine to exit. Therefore, the expected rate of VM exits differs significantly from the expected rate of system calls by an ordinary application.

I assume that once the virtual machine has entered guest mode, it runs at the same speed, i.e. instructions use the same number of cycles to execute, within the guest and the host. With this in mind, the dominant influence on performance overhead is the number of exits a virtual machine takes and the time needed to process them and enter the guest again. Of course, the time an exit takes depends largely on what the monitor does to process it, however, this does not change either way the monitor runs within or separated from the guest domain. What matters is the additional cost for the exit when changing to the monitor's domain. Therefore, the additional context switches needed will affect performance.

2.3 Hardware

Many of the problems one needs to solve when creating virtual machine can either be amplified or simplified by the hardware architecture. Some architectures are more tuned to virtualization than others. It depends on what the aim of the architecture designers was. The mainframes of the 70s and 80s were designed for virtualization. These architectures had a simple instruction set and used software-loaded TLBs. Every TLB miss is routed through the monitor, which can decide, through its own data structures, what to do with such a memory access. There is no need for shadow or nested paging. Trap and emulate works directly with such an architecture. Their virtual machine monitors were shipped with the hardware.

2.3.1 The x86 architecture

The x86 architecture started out in the 80s to be used in personal computers which are small, compared to mainframes. The architecture was not designed to run virtual machines. The resources such a computer offered were limited and sufficed to run a single operating system and some applications. Later on, protected mode and virtual memory was added and personal computers using the x86 architecture became enough powerful during the 90s such that running virtual machines seemed possible.

However, the x86 architecture features 17 instructions which are problematic with respect to virtual machines. These instructions behave differently when executed in privileged and in unprivileged mode. They are related to change or exposure of CPU state and memory access. A guest operating system expects to know the complete state of the CPU but since it is running as unprivileged code in the virtual machine it will only see the state a user-space process would see. In order to proper virtualize such an instruction it should cause a trap into system software when executed in unprivileged mode.

An example for such an instruction is `pushf` and `popf`, which let a process inspect and alter the current CPU flags. The CPU flags of the host most probably differ from the flags of the guest. Therefore, information can leak from the host to the guest through these instructions. Furthermore, the guest can alter the host's flags without it noticing that change.

The x86 architecture uses a hardware loaded TLB. In the case of a TLB miss, the CPU walks a predefined page table structure to resolve a particular virtual address. System software is only invoked if the page table does not map that particular virtual address. This event is called a *page fault*. Therefore, every instruction causing the TLB being consulted is part of the set of these 17 problematic instructions because the monitor can only control the entries loaded into the TLB by controlling the page table the guest uses. To control the guest page table shadow paging or nested paging can be used.

To provide a secure virtual machine on top of the x86 architecture one must at least alter the running guest either through dynamic or static translation such that all instructions related to state change or exposure and all instructions dealing with page tables cause the virtual machine to do a VM exit.

2.3.2 Secure virtual machine (SVM)

The main suppliers for x86 CPUs, AMD and Intel, released special versions of their CPUs with an extended feature set. The goal of this these features was to improve the architecture such that implementing virtual machines on x86 CPUs is simplified. The implementation Intel and AMD chose differ and are not binary compatible. However, the semantics of these two implementations are similar. *Secure Virtual Machine (SVM)* is AMD's implementation.

One of the main features of SVM [3] is the addition of a new CPU mode, the guest mode. Within guest, mode every CPU state is replicated. Guest mode by definition always runs in unprivileged mode, but the current protection level indicator may show the CPU being in privileged mode. It therefore can simulate privileged mode to a possible guest without its awareness. SVM offers atomic CPU state handling. It enables the monitor to capture and store all important CPU state to a designated area with one instruction. Another instruction is provided to load the same CPU state from a designated area into the CPU. When the CPU enters guest mode, it will save the host state to some memory location, and load the guest state from the area specified by the monitor. On a VM exit, the CPU will do the same operation in reverse. Therefore, no guest CPU state may be leaked into of from the host. The CPU supports a control area for each virtual machine defining its settings. This area includes flags to tell the CPU the conditions to exit from guest mode. At least all the unsafe operations, including the 17 un-virtualizable instructions, can be intercepted through these flags. These features provide the basic support to make the x86 architecture suitable for virtualization without having to modify the guest. The CPU will never run in privileged mode and guest mode simultaneously, and guarantees the separation of CPU states between guest and host.

AMD supports the use of a *tagged TLB* in their CPUs including the virtualization extensions. In a tagged TLB every entry is marked by an additional tag. In addition, every protection domain is associated with a tag. It is the responsibility the operating system to assign tags to different protection domains. If a virtual address is be translated into a physical address, the CPU will only consider those TLB lines which are marked by the same tag assigned to the currently running protection domain. Switching between domains with different tags no longer requires a TLB. Switching between domains with the same tag only flushes identically-tagged TLB lines. The speed of context switches can be improved using a tagged TLB, if the the two contexts in question do not use the same tag.

SVM provides nested paging as described above 2.2.4. The guest and the nested page table can coexist within the same TLB through the use of tags. However, the host and the guest must not use the same tag, since the architecture cannot distinguish between guest page table and nested page table entries within the same TLB.

2.4 Summary

This chapter introduces the virtual machine term and all its relevant components. It enumerates and explains different techniques which can be used to build virtual machines, the problems which need to be solved and how to solve them. The x86 architecture is introduced and the challenges and solutions using virtualization technology on x86 are discussed.

Chapter 3

Literature Review

3.1 Requirements for Virtualization

In 1974 Popek and Goldberg specified the requirements [26] hardware and software must implement to make trap-and-emulate virtualization possible. Their requirements to the hardware are that its instruction set shall be divided into privileged and unprivileged instructions, and the attempt to execute privileged instructions in unprivileged mode must trap the CPU into the operating system, running in privileged mode. There shall be no way that CPU state is exposed to the virtual machine in such a way a potential guest can distinguish between running on virtual or physical hardware.

They further define the concept of a virtual machine monitor, which is a piece of software immediately above the hardware and below guest operating systems. It shall be in control of all hardware resources. It shall be implemented in such a way that a statistically dominant proportion of the instructions a guest executes are run on physical hardware. Trap-and-emulate virtualization does not, except for trivial guests, allow every instruction to be executed on real hardware. Some instructions, such as those reporting CPU state, must be emulated by the monitor to separate host from guest.

3.2 Mainframe Virtualization

One of the first use cases for virtual machines was the multiplexing of mainframes. These machines were designed host virtual machines. The suppliers of these computers normally also distributed dedicated system software containing the virtual machine monitor designed for a particular mainframe. Through this homogenous setup they achieved acceptable performance with limited resources.

An of such a machine is the IBM System/370 [15]. This system was built with support for running virtual machines and appeared in the early 70s. In addition to the full trap-and-emulate support, special instructions were added to the hardware to help the virtual machine monitor simulate privileged instructions to the guest operating system. This enabled users of these systems to run unmodified guest operating systems with similar performance compared to running directly on the physical hardware. Similar support can today be found in the hardware virtualization extensions to the x86 architecture [3, 29].

3.3 Runtime Translation

As described in 2.3.1 the are challenges to solve when constructing a virtual machine for a machine using the x86 architecture compared to one where trap and emulate can be used directly [27], i.e. the x86 does not conform to Popek's and Goldberg's requirements [26]. Dynamic translation of guest code can be used to solve these problems.

Dynamic translation method was invented by VMware [13] in the late 90s. Their approach is to binary translate the guest code before it is executed on the physical hardware [2]. They implemented this by scanning the code to be executed until the next branch or special instruction. During the scanning, they translate the instructions in such a way that the virtual machine traps into the monitor if needed. Special instructions are all those which should behave differently than they actually do on the x86 platform to support proper virtualization. In the end, this system acts as a just in time compiler. It takes as input generic x86 instructions and compiles them into instructions suitable to run safely inside a virtual machine. Every distinct basic block of a guest must be translated once, if it is reached by the execution stream. VMware also uses run time translation to implement shadow tables. Every instruction which changes the guest page table location is replaced by a trap. To prevent the need for every memory-related instruction to trap into the monitor, they do not allow the guest to change its page tables in place unnoticed.

When a page fault occurs, VMware distinguishes between real page faults and hidden page faults. Real page faults are those which are raised like they would be if the guest ran on real hardware. They are propagated to the guest operating system which should handle these types of faults. Hidden page faults are those which are raised due to the fact the shadow page table is not synchronized to the guest page table. These faults are silently handled without the guest noticing. There are more page faults when executing the guest in a virtual machine compared to real hardware, but for every guest page table a distinct hidden page fault only occurs once, as long as the guest does not alter that particular part of the page table.

The QEMU project [7] uses a similar approach to achieve a different goal. QEMU supports a complete emulation of different virtual machines on top of a non-binary-compatible host machine. There exists a subset of all possible instructions which can be found on most CPU architectures. QEMU defined such a subset. For every supported host architecture QEMU knows the implementation of all instructions in that subset. Given a guest QEMU translates at run time, on a basic block level, the code into that generic instruction set for the host the virtual machine is running on. Since CPU architectures can be very different, QEMU needs to emulate more code than a virtual machine which is only capable of running binary-compatible code. QEMU tries to run as much code as possible directly on physical hardware, but experiences many more VM exits compared to, for example VMware. The overall virtual machine performance is reduced compared to directly running the guest on the physical hardware. The guest runs around 4-10 times slower than on physical hardware. The developers of QEMU introduced a set of optimizations, for example when the host and guest architecture are the same, the translation becomes much simpler and faster because the guest instructions do not have to be translated to that generic instruction set.

3.4 Static Translation

As described in section 2.2.3 it is possible to alter a guest at compile time to conform to the rules the virtual machine requires their guests to obey. Xen [5] is an implementation of such a virtual machine monitor. The developers of Xen created an interface for the operating system of the guest to communicate with the host on a higher level than privileged x86 instructions. Their interface is comparable to system calls an operating system offers to user-space processes, but is much simpler. As an analogy to the system call term they gave these monitor invocations the name *hypercalls*. A version of the Linux kernel was modified initially, which is able to run on top of Xen, called XenLinux. The developers of Xen claims [5] to have a copy of Windows XP capable of running on their virtual machine. Xen drives the major part of the physical hardware through a special version of a guest, called *domain0 (dom0)* [17] . Dom0 has elevated privileges within the entire system, like accessing physical hardware directly. The rest of the guests have access to a set of generic devices which are driven through channels and events, i.e. an upcall interface, and are securely multiplexed among multiple guests and dom0. The structure of Xen is summarized in figure 3.1. Through this high level interface provided to guests, Xen guests achieve almost similar performance using general virtual hardware, such as disks and network controllers, compared to using physical devices. However, the guest is of course aware that it is run within Xen. This contradicts traditional virtual machine requirements, but can be used to take advantage out of the tight coupling of monitor and guest. Xen uses ring buffers for a lot of the communication between guest operating system and virtual hardware. This enables implicit batching of requests. It can only be done because the guest is aware of these buffers and therefore their location in memory. Xen guests collaborate with their virtual machine monitor and therefore the Xen developers invented the term *paravirtualization* for this approach.

Another project using paravirtualization is Denali [30]. Denali's approach is to offer virtual machines for services rather than full fledged guests. Denali does not only provide a virtual machine monitor but also the guest operating system which should be used within the virtual machines. The guest operating system does not offer address space protection and should be used to run a single service, such as a web server. It uses the hypercall interface offered by the Denali virtual machine. The developers of Denali did these simplifications because they want to be able to run a large number of these stripped down virtual machines on one host machine. The virtual machines can talk to each other through sockets like different hosts do. Therefore, the security implications of running multiple services on one host are reduced to the same level as if those services were run on different physical machines, which is an improvement over running the same set of services within one operating system environment.

3.5 No Translation

In the background on hardware assistance for virtualization, in section 2.3.2, the extensions by AMD [3] and Intel [19, 29] were introduced. These extensions were already used to create virtual machines.

An example of a virtual machine running on x86 is the *kernel-based virtual*

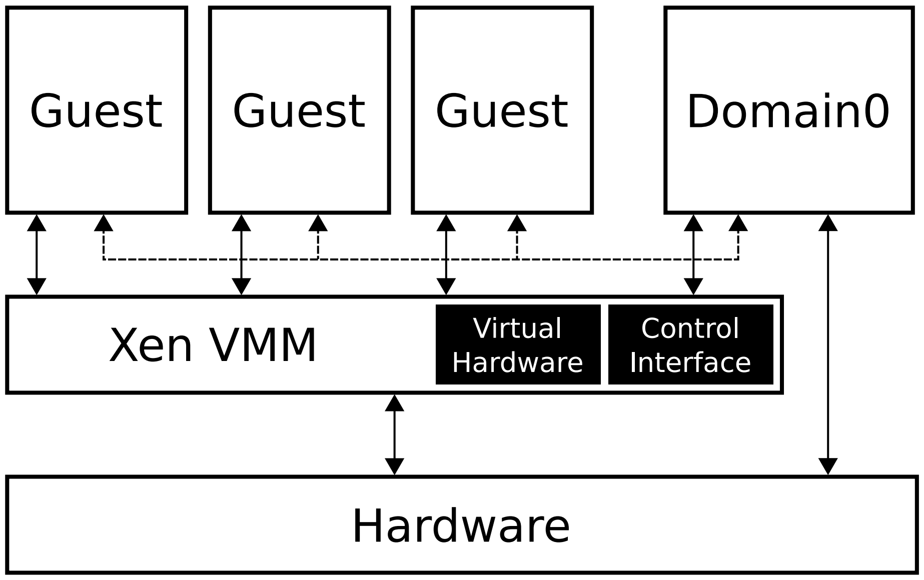

Figure 3.1: The Xen virtual machine. The guests are modified to use the hypercall interface provided by the monitor. There is a special guest, domain0, which has direct access to the system's hardware. It also has access to the control interface of the monitor to perform actions like starting or terminating a guest. The guests communicate with domain0 to gain high-level access to its generic hardware devices.

machine (KVM). The Linux KVM [20] is a small piece of code inside the Linux kernel which exposes a virtual machine interface to user space. It uses the hardware extensions, such as SVM, to run unmodified guests. KVM is tightly coupled to Linux. A Linux process can be in two modes, *kernel* and *user mode.* KVM adds an additional *guest mode* to each user-space proccess. When a certain Linux process is used as a part of a virtual machine, then the guest mode is used to execute the guest within that process. The monitor of that virtual machine is executed within user mode of the same process. The kernel mode is used to perform the privileged operations needed to execute the guest on the hardware. The code used for kernel mode is loaded into the Linux kernel. The virtual machine is exposed to user space via a traditional UNIX interface using a device node and designated `ioctl()` calls. Any capable user space application can act as a monitor to run guests via KVM. The KVM developers took a version of QEMU [7] and altered it to run its guest via KVM instead of its own virtual machine. They were able reuse much of the hardware emulation code of QEMU, which was already able to simulate PC hardware as a Linux process.

VMware also tried to take advantage of the newly-available hardware extensions, and created a modified version of their virtual machine monitor [2]. It turned out that their original virtual machine outperformed the new one in most evaluations. With unmodified guests every single instruction which is intercepted will result in a VM exit. A VM exit takes some time to happen because the CPU has to perform a *world switch* which is harder then a context switch. VMware implemented many optimizations which are specifically tailored to certain types of guest operating systems. Through their translation process they reduced the amount of VM exits needed. VMware further replaced some calls which would normally trap the CPU by calls into the monitor which is significantly faster than trapping. It turned theses optimizations improved performance more the loss they take when performing the translation on the guest.

3.6 Hybrid Approaches

An advantage of paravirtualization over dynamic translation is the increased performance the guest experiences due to the use of the high level interfaces to the virtual machine monitor. XenLinux running on Xen experiences similar performance to Linux running directly on the physical machine. However, the drawback of this approach is the tight coupling of monitor and guest. The guest needs knowledge about its monitor. Virtual machines using dynamic translation however, do not assume anything about the guest operating system but the ability to run directly on the physical machine. Hybrid approaches try to combine these two techniques. The intention is to achieve the same guest execution performance as Xen while staying compatible with as many guests as VMware or KVM.

One method to achieve this goal is *pre-virtualization* [22]. The argument of LeVasseur *et al.* against pure paravirtualization is an operating system built to run on Xen is not able to run on a virtual machine using dynamic translation nor real hardware. They built a system which enables a guest operating system to run on physical hardware and on different paravirtualization systems. They introduce a layer between the monitor and the guest. This layer is able to use the virtual machine's services. Pre-virtualization add annotations to the guest which do not alter the way the guest would run on real hardware but are recognized by this layer in between. Doing so, the guest may use the interface of, for example, Xen to improve performance. Guests without those modifications will still run at the same speed as they would on virtual machines offering full virtualization or dynamic translation.

Using static translation may pose problems because one might be unable to obtain a modified version of the guest operating system one wants to run. That is the main argument of another approach to this problem, called *optimized paravirtualization* [24]. In this approach the virtual machine monitor is capable of both performing full virtualization and offering a hypercall interface at the same time to the same guest. The guests are able to run on real machines, as opposed to Xen guests, and therefore may detect the availability of the hypercall interface of the virtual machine monitor and decide which of these calls it want to use. The guest can then trade-off between performance increase in certain subsystems and implementation effort to make these increases happen.

3.7 User-Mode Virtualization

User-mode virtualization is a special kind of static translation. Systems like Xen take a CPU interface, such as x86, and construct a hypercall interface around it.

In user-mode virtualization, guest operating systems are ported to the system call interface of another operating system. This interface is normally higher-level than a hypercall interface and is not specially adapted to a particular hardware architecture. Therefore, the potential guest operating system normally needs to be ported to a completely different architecture, which happens to be a system call interface.

An example of such an port is *User-Mode Linux (UML)* [16,18]. It is a port of the Linux kernel to its own system call interface. The UML kernel is then run as a normal user process. UML make use of Linux's `ptrace()` interface which was designed to debug user space applications. With `ptrace()` UML is able to catch every system call issued by a user process inside the guest. Once such a call is caught, UML forwards it to the guest kernel and change the original into a `getpid()` system call. From Linux's perspective, UML is a separate architecture like any other. It provides its own implementation of interrupts, virtual memory and other hardware services. Virtual memory is implemented through `mmap()`. A guest its monitor is represented through a Linux process. No action is taken to do additional protection. Everything running together with the operating system in the guest is able to compromise the whole guest. Interrupts and exceptions are implemented through UNIX signals. If an application inside the guests touches an unmapped region of memory, i.e. causing a page fault, the host kernel will emit a `SIGSEGV` to the process running the guest. The virtual machine monitor running within that process will identify this signal as a page fault and propagate it to the kernel inside the guest. It is then up to the kernel to decide whether this was a legitimate page fault, and allocate memory through `mmap()`, or generate a new `SIGSEGV` to the currently running user-space application. The primary use case of UML was architecture-independent Linux kernel development.

3.8 IO Performance

With the techniques available today, it is possible to multiplex the CPU to different guests with little performance loss. During non-privileged code execution, there are not many VM exits performed compared to the amount of instructions executed. However, when a virtual machine provides virtual hardware, every guest access will cause a VM exit. These exits have to be done because virtual hardware needs to be simulated by the monitor the lack of safety for the monitor if the guest accesses as physical device directly. One of the main reasons for this lack of safety is the use of direct memory access (DMA). DMA is a method which enables a device outside the CPU access to the main memory of the computer without the CPU overseeing or performing the operation. Therefore, it can be done concurrently to different code execution within the CPU. Many devices which involve much data being transferred between CPU and device, such as disk controllers and network controllers use this technique to chunk big portions of data into blocks exchanged asynchronously from the CPU and the device to main memory and back. The driver of such a device tells it where to read and write data from in physical address space. These accesses are not checked by the MMU. If a virtual machine guest would be led to drive a certain device with DMA support it would be able to alter any memory location within the whole system. Furthermore, a guest can in general not know the host-physical address-space locations of its guest-physical memory and physical devices are not aware of the guest-physical address space. Therefore, DMA cannot be used inside virtual machines.

Hardware designers came up with a new memory translation system, called IOMMU [4]. The IOMMU, initially designed to overcome the 32-bit address limitation for DMA in 64-bit systems, enables remapping of bus addresses to host addresses at page granularity. This mechanism can also be used to map the host-physical address space to the guest-physical address space to make the use of DMA from virtual machine guests more simple and secure [1]. The current IOMMU implementations only allow one page table to be active at all time. Therefore, the monitor will create one IOMMU page table to map the device registers of all devices directly used by the guests into the corresponding guest-physical address space. If a virtualization solution uses more than one monitor, then all monitors need to agree on a common IOMMU page table. Using this protection mechanism, one may allow a guest to program a device to write directly to physical memory. If the device touches memory which it must not, a hardware page fault will be raised before the memory access is performed, preventing the system of taking damage. This technology has been added to recent versions of the x86-64 architecture and was used to evaluate the impact to current virtualization implementations, such as to Xen [8].

VMware experienced the same problem before x86 provided any hardware acceleration for virtual machines. The performance losses using virtual devices were significant compared to real devices [28]. If a guest operating system is not changed, the only way a guest can use a device is by emulating that device within the virtual machine. However, the guest will access that device like a normal one which will cause a many VM exits, i.e. on every privileged instruction, such as `in` and `out`. VMware tries to reduce the amount of VM exits by providing a special driver to the guest operating system. This driver will not access the virtual hardware like a normal driver, but will use a higher level interface provided by the underlying virtual machine. Operations normally performed with several privileged instructions are batched in one call to the monitor. This method is comparable to a hypercall interface used in paravirtualization, but without the knowledge of the guest operating. The method has therefore also the same downsides. The supplier of the virtual machine must provide such drivers for all possible guest operating systems, to make them use these special interfaces.

3.9 Virtualization on NUMA Hardware

Computers can be built by assembling many CPUs and memory banks together and connect them via a set of buses. These computers behave much like a distributed system. Communication between nodes must be managed explicitly and the latencies and bandwidth can differ between different nodes. The memory-related symptoms are called *non-uniform memory access (NUMA)*. System software running on such machines must be aware of the performance implications when accessing memory from one CPU which is not local to that CPU. The locality of memory to a CPU is defined in the latency the CPU experiences when accessing a certain memory location. Data used by a CPU core has to be kept local to that core to get the maximum performance out of the system.

Disco [12] is an implementation of a virtual machine on top of a NUMA architecture. It predates work on x86 virtualization, such as VMware. The motivation for Disco is similar to the virtual machines built for mainframes. Disco's goal is to virtualize operating systems made for commodity hardware on NUMA machines. Commodity operating systems were not built to run directly on NUMA hardware. The goal of Disco was to run the guest operating systems faster on NUMA hardware by exploiting its special properties than on commodity hardware. Disco reached that goal by replicating the virtual machine monitor to each core of the machine and by implementing a clever page moving and coping system such that virtual machines have their memory close to them with respect to speed. The guest experiences uniform memory accesses. It does not need to be NUMA aware. Disco was implemented on the FLASH multiprocessor [21], a cache coherent NUMA architecture designed in 1994. The processor simulated inside the virtual machines was a MIPS R10000.

Another noteworthy project in this context is vNUMA [14]. It is in some sense the opposite to Disco [12]. While Disco simulates a machine providing uniform memory access on top of a NUMA machine, vNUMA simulates a NUMA machine on top of several machines with uniform memory access. vNUMA uses Itaniums (IA-64) connected by gigabit Ethernet links to establish a NUMA-like distributed system. It can be used to run NUMA-aware guests inside the virtual machine and simulate a big computer through the use of several relatively small computers.

3.10 Guest as Driver Provider

Virtualization techniques may be used to extend a host operating system in a special way. If one manages to virtualize an operating system with support for different types of hardware, one may use this guest as a driver provider and export the hardware it supports back to the host. As described earlier, Xen [17] uses a special guest, the dom0, for this purpose. This guest is given elevated privileges to operate hardware. Xen trusts this domain to perform correct and secure DMA accesses. The availability and use of an IOMMU would improve overall security. Without it, dom0 is part of the TCB. Dom0 exports its hardware through generic data and event channels to any permitted other guest.

A similar approach is taken by LeVasseur *et al.* [23]. This paper describes a project where virtual machines are entirely used to encapsulate unmodified drivers. Their motivation is to reduce workload from operating system developers to implement the same drivers again and again. The virtual machine monitor is implemented within a library which they claim is easy to include in any given operating system. Extra care is taken about the security and separation of the different drivers. Every driver is executed in a separate virtual machine which provides separation of the drivers from each other. Drivers may be restarted individually. The monitor also allows the guests to use DMA which, without the use of an IOMMU, moves all the guests using it into the TCB of the host system.

Chapter 4

Barrelfish

Barrelfish [6] is a research operating system developed by the Systems Group at ETH Zurich. It was designed to meet the requirements of today's and in particular future hardware designs. It is based on the assumption that tomorrows architectures will be more heterogeneous than they are today. Chips contain multiple cores where not all of them may support the same instruction set. Memory access latency is not uniformly distributed among the different cores. Caches do not have to be coherent and are not accessible by all cores. To some extent this is already case in recent mainstream architectures.

4.1 Structure

To cope with this situation Baumann *et al.* created the *multikernel* operating system architecture. Barrelfish is an instance of a multikernel operating system. In such an OS, every core hosts one kernel, called the *CPU driver* in Barrelfish. All kernels are essentially independent. State is not shared but replicated as needed among the different cores.

The CPU driver runs in privileged mode and is responsible for multiplexing the underlying hardware resources in a safe way to all user-space processes running on the same core as the CPU driver. Furthermore, it is responsible for scheduling and dispatch of these user-space processes. Since a CPU driver does not share any data structures with other cores, it does not need to lock its data structures to protect them from access. It is single threaded, completely event driven and nonpreemptable.

User-space processes are called *domains* in Barrelfish. A domain consists of a protection domain, a virtual address space, and a set of *dispatchers.* A domain which wants to execute code on a given core needs to set up a dispatcher registered to the CPU driver on that core. It will then, once the dispatcher is made runnable, enter the scheduling queue of that CPU driver.

On every core runs one special domain, called *monitor.* It is the user-space counterpart of the CPU driver. It encapsulates a lot of the functionality that would be found in the kernel of a traditional operating system. The monitor is responsible for the coordination of the system-wide state of the operating system. To that end some data structures need to be replicated over different cores. The different monitors employ an agreement protocol to keep all replicated data structures globally consistent. The monitor is further responsible for setting up inter-process communication channels and waking up processes waiting for a response from another core.

Barrelfish is vertically structured, much like an Exokernel []. The CPU driver is highly architecture dependent and exports means to interface with the bare hardware features of the x86-64 architecture in a safe multiplexed way. Drivers are implemented in their own domain like in a microkernel approach. Barrelfish's CPU driver is small, with less than 10kLOC.

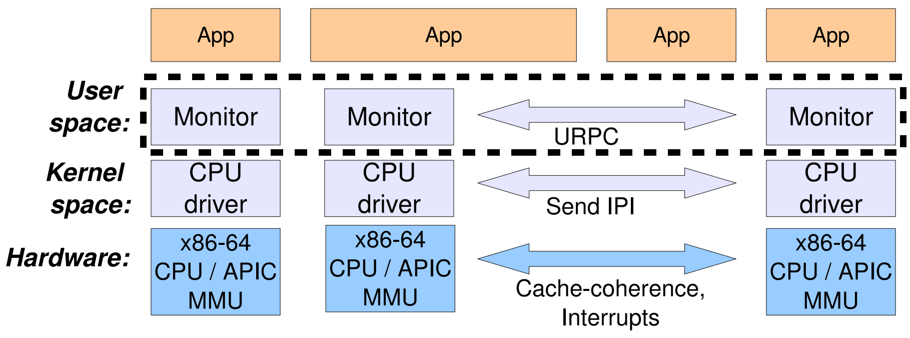

Figure 4.1: Barrelfish structure. Source: [6]

4.2 Dispatch

Every executable domain within Barrelfish features one or more dispatcher objects. A dispatcher object is a shared data structure between the CPU driver and the corresponding domain. It is local to the core on which the CPU driver runs and represents the required interface to execute a domain on that core. The dispatcher encapsulates an upcall interface which is called by the CPU driver every time the domain should run on that particular core. It is up to the dispatcher to decide what should be done with the available time slice on that core. This is different to traditional operating systems, which just continue a process once it is scheduled rather than calling it at a defined entry point. Threading within domains is entirely implemented by the dispatchers. They maintain all relevant data structures to make threading possible. The CPU driver has no knowledge about the concept of threads.

4.3 Capabilities and Memory Management

Barrelfish manages all its crucial resources with capabilities. A capability is an object with certain properties assigned to that object. It may represent some right or resource within the system. Capabilities may only be manipulated by privileged code. A process may own a reference representing such a capability. This reference can be used to invoke several operations on capabilities to modify them. The CPU driver is able, for each such reference, to find out whether it is valid and to which particular capability it points. Capabilities can be viewed as the opposite to access control lists (ACLs) which are used as a security primitive in many traditional operating systems. An ACL on an object defines who has which right over that object. In contrast a capability is a object which encapsulates rights over a set of objects.

Domains need to manage their memory for themselves to the extent of managing its page tables. Also, the CPU driver does not allocate memory, and, when a domain wants, for example, to create a data structure shared with the CPU driver, such as a dispatcher, it has to allocate that memory and tell the CPU driver about that memory region. Everything is implemented through capabilities. The complete memory region is represented by capabilities. It may be split into several sub regions, it may be mapped into a virtual address space. It can be used to store protected data structures, also by retyping memory capabilities.

A domain will normally request some memory capabilities from the memory server and use it to do everything needed for software to run within a virtual address space. It will map some parts of this memory into the heap and other parts to the stack of its threads. Domains have to handle page faults for themselves and may therefore implement a dynamic stack size for each thread.

4.4 Communication

Barrelfish employs explicit rather than implicit sharing of data structures. Implicit sharing denotes the access of the same memory region from different processes which usually needs to be protected through locks. Explicit sharing denotes several copies of the same data structures, one located at each process. Messages are exchanged with updates done to these data structures to keep them synchronized among all participating processes.

Messages can be passed through different channels depending whether the communication is intra- or inter-core. For communication on the same core, IDC or LRPC messages can be used. IDC is a CPU-driver-mediated message transport. It enables domains and the CPU driver to send messages of fixed sizes to each other. The message can optionally also unblock the receiver which will then take over the remaining time of the caller's time slice to process the request. LPRC [9] is offered as an optimization of synchronous communication for latency-sensitive operations.

Inter-core communication is done using URPC [10] in the current implementation. It exploits the cache-coherent memory of the x86-64 architecture. Messages are exchanged via shared memory. They are cache line sized. The implementation is optimized to reduce the communication needed on a possible interconnect between cores. Different hardware architectures may need different ways to implement the inter-core communication, especially if there is no cache-coherent memory available.

Barrelfish offers some abstractions to simplify the use of those communication primitives. A library, called *chips*, is offered which is based on services, clients and servers, and is built over those communication primitives. It hides the complexity whether the receiver is located on the same or another core. Furthermore, there is an IDL generator, called *flounder*, which can be used to define the interfaces between communication entities. The code generated will make use of chips.

Chapter 5

Approach

The background on virtualization and the literature review showed there are many different applications for virtual machines and many ways to construct them. Some of them are more bound to a specific purpose, like running drivers in a secure way, others are more general in their nature. VMkit shall provide a virtual machine solution which is general in the sense that there is no inherent predefined use but to gain advantage from running guests within Barrelfish.

VMkit shall be able to run unmodified guests. Moreover, guests should not need to know that they are running on a virtual machine.

5.1 Targeted Architecture

It is the aim of this work to build a virtual machine monitor for the x86-64 architecture. A lot of recent research on virtualization is based on this architecture, which makes comparison between different virtualization solutions easier. Also, Barrelfish has an implementation ready for the x86-64.

Section 2.3.2 described that Intel and AMD, the main suppliers of x86-64 CPUs, were not able to produce a standardized interface to their virtualization extensions. VMkit shall use these extensions, because they simplify the implementation of the virtual machine monitor. There is no scientific advantage to creating a monitor using both in the first place, but porting it to the other architecture should not be hindered. VMkit shall be implemented using SVM, the AMD version of these two extension types. The main advantage of SVM was the availability of the technology in the development machines available to me.

5.2 Structure

This section describes the general structure of VMkit.

First the structure of the guest and the monitor will be described, followed by the description of certain deviations from basic design principle to include some optimizations.

5.2.1 Monitor and guest

The importance of the separation of monitor and guest can be argued. As long as the monitor is protected by hardware means, i.e. running as a user-space process, and as long as one monitor only controls one guest, the possible damage caused by the guest accessing monitor memory can be limited to that particular guest only. However, the guest may, with appropriate knowledge, act as a normal user-space process within the host operating system, which may not be desired.

One of the key requirements of a virtual machine should be its protection for the host being jeopardized by the guest.Therefore, placing the monitor within the kernel needs a good argument why in that particular case this would be a secure solution.

The VMkit monitor is placed into a user-space process. There are also other reasons, besides the security concerns mentioned above, for the monitor to be situated within user-space. Barrelfish employs a microkernel structure. Placing too much functionality within the kernel would not be appreciated by its developers. The monitor process should encapsulate as much of the logic as possible to run the virtual machine. As little functionality as possible should go into the kernel.

The guest is placed into a separate user-space process, protected from the monitor. The guest process does not execute any code other than guest code. Any controlling activities are done through the monitor process.

This structure induces many additional context switches compared to the model where the monitor and the guest are located together within one user-space process, or when the monitor is part of the operating system kernel. With the use of a tagged TLB, the guest and monitor process can be assigned different tags and therefore do not need the TLB flushed when switching between them. This eases concerns about the security of the monitor without creating a large performance penalty. As we will see in the implementation, the design of Barrelfish, i.e. its vertical structure, also supports this decision. The monitor process can control the guest process in every way necessary. Furthermore, the monitor process needs not take extra precautions to protect itself from the guest, which makes it simpler and more understandable.

The monitor makes use of the hardware extensions provided by the CPU to manage the guests memory and the VM exits. It provides all the virtual hardware available inside the virtual machine, either through complete simulation, or the use of a driver within Barrelfish.

Figure 5.1 summarizes the global design of the system.

5.2.2 Kernel and fast path

The kernel part of VMkit is very small and consists only of the parts which need the CPU to be in supervisor mode when executed. It handles the state-related issues when entering and exiting the virtual machine such that neither the host nor the guest are compromised.

A monitor and its guest are closely-coupled pieces of software. They never run in parallel but always run as a result of either an event from the guest or the continuation of the guest through the monitor. From the CPU driver's perspective, these switches are similar to switches between any two user-space

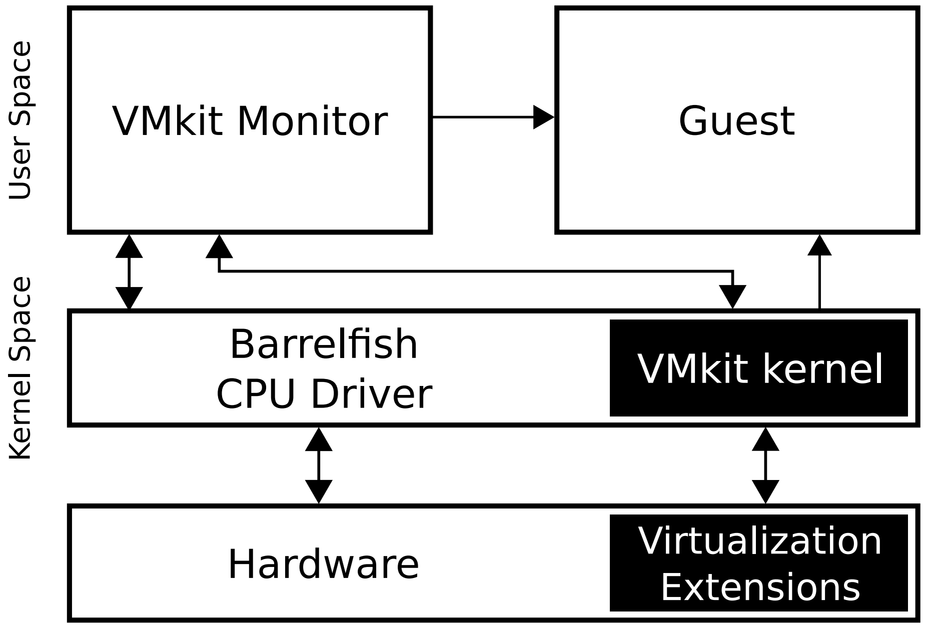

Figure 5.1: VMkit structure. The monitor and the guest are located in user space. The CPU driver is extended by means to operate the virtualization extensions provided by hardware. The arrows indicate means of communication and memory manipulation.

processes. Therefore, it is not defined which process will run next, which could impose additional latency when executing the virtual machine because it will yield its time slice every time it performs a VM exit. To improve performance, additional logic is placed into the kernel. When a guest takes an exit it would be favorable if the monitor could run and process the exit as fast as possible. The remaining unused cycles by the guest within its time slice can be used by the monitor to process the exit and eventually perform a VM enter again. Therefore, the kernel needs to know for any given guest, what the corresponding monitor process is. This fast path will reduce the overall latency between monitor and guest execution without taking any execution time away from other runnable user-space processes.

However, not all VM exits shall result in a monitor invocation. There are two general classes of exits a guests can take during execution: guest external and guest internal exits. Guest external exits are interrupts and exceptions raised by the real CPU. Exceptions during guest execution are always caused by the guest. Other than fatal errors, such as a triple fault which would cause a real machine to reboot, all these exceptions can be forwarded to the guest. Without the virtual machine managing real hardware through its guest, external interrupts will never be designated solely for the virtual machine. Calling the monitor on such a condition would increase the overall latency of the system in responding to the interrupt. Moreover, the called monitor would most likely not be able to handle the VM exit because it is not a driver for that device. The kernel part of VMkit shall have knowledge about some causes of VM exits, especially guest external causes, and execute the corresponding interrupt handler within the CPU driver directly instead of calling the monitor.

All other VM exits are not events by real hardware, are therefore related to the executing guest, and must be handled by the controlling monitor.

5.3 Memory Management

The virtual machine has to simulate the same memory environment to the guest it could expect running on real hardware. This includes at least a region of physical memory which it may use freely, and possibly some memory-mapped device registers. The guest operating system will manage this memory by generating page tables translating guest-virtual addresses to guest-physical addresses within these regions. The virtual machine monitor has to be able to distinguish between physical memory and device register accesses. In the latter case it has to execute device emulation code. As described in section 2.2.4 the performance of the virtual machine depends to a great extent on the MMU being virtualized. The monitor needs to manage the guest's physical memory either through nested or shadow paging, otherwise it would require to exit the guest on every memory access.

VMkit uses nested paging. Nested paging needs less effort to be implemented correctly than shadow paging. The monitor will be less coupled to the guest's page tables. Moreover, implementing a shadow paging mechanism which will able to yield the same or better performance as nested paging would require a lot of optimization and increase code size and implementation effort to an unacceptable level. I will not be able to test this hypothesis due to the lack of time to implement a shadow paging mechanism.

Nested paging is straightforward to implement using Barrelfish domains which manage their memory for themselves. The kernel will use the domain's page table as the nested page table when performing the world switch.

Chapter 6

Implementation

6.1 Host and Guest System

The virtual machine was developed and runs on AMD Opetron 8350 and AMD Opteron 8380 CPUs. The machines featured multiple CPUs and cores but only one core was used during development.

The emulated machine is either x86 similar to an AMD K6 or a generic x86-64 CPU. Most of the extra features which can be detected through the `cpuid` instruction are shown to the virtual machine as being unavailable, this includes for example the IOAPIC.

6.2 Structure

As described in the approach (chapter 5) the monitor is separated from the guest, and most of the parts of the monitor should be located in a user space process.

The kernel part of VMkit is located inside Barrelfish's CPU driver. The rest of the monitor is implemented through a Barrelfish user space domain. The guest runs in a separate user space domain. Currently a monitor controls exactly one guest.

6.3 Kernel and Guest Domain

The guest and monitor are are separated through different Barrelfish domains. The CPU driver knows for each domain whether it is a virtual machine guest domain or a normal domain. The kernel part and the guest domain are closely coupled. Most of the code inside the kernel is used to make the guest domain possible.

A domain hosting the guest does not use any of the Barrelfish features available for domains. Its sole purpose is to provide the protection desired and provide the nested page tables. If a common domain is scheduled, its dispatcher is invoked to process the invocation and decide what to do with the time slice. The VMkit kernel alters this behavior for all domains it knows to contain guests. If a the scheduler decides to run such a domain then VMkit code inside the CPU driver will be called. This code will, instead of calling the domain's dispatcher, save all the necessary host state and perform a world switch into the virtual machine running the guest, i.e. perform a VM enter. Performing the VM enter involves loading the guest state from a saved location and reestablishing that state onto the CPU. This state also contains the last instruction pointer used by the guest, or some initialization value if the guest never ran before. The CPU will then run in guest mode until a condition occurs which will force the CPU to exit the virtual machine, i.e. perform a VM exit. In this event VMkit's CPU driver portion will be executed again and perform the world switch back to the host, saving the guests state. It will then examine the reason why the hardware wanted to exit the VM and either call the corresponding monitor or, in case of an interrupt, call the corresponding interrupt handler. If the monitor is called, the guest domain is removed from the scheduler queue, i.e. made unrunnable, otherwise it will remain in the runnable state.

The VMkit kernel part has enough knowledge needed to identify and run a guest domain. No monitor interaction is needed to perform a VM enter. For example, in the case that the guest is exited because of a timer interrupt on the CPU, i.e. preemption, the monitor will not be invoked, but the interrupt handler will be. The handler will call the scheduler to identify the next domain to run. The guest domain will remain in the runnable state and therefore be scheduled again later. On this event, the mechanism described above will enter the guest. All this can be done without the monitor even noticing the guest was preempted. As long as there is no exit which requires the monitors attention, it will not be invoked, and the guest domain will be executed like any other runnable domain within the system.

6.4 Monitor

The monitor is implemented using a normal Barrelfish domain. Its duty is to create and destroy its guest and handle all possible VM exits originating from the guest. It is therefore basically an event processing machine.

Once started, the monitor will set up the necessary data structures the hardware needs to run the virtual machines. These data structures contain all settings regarding the guest's memory and intercepts, i.e. VM exit conditions. Furthermore, the initial machine state is established such that the same environment is simulated as one would encounter when running from a real machine. The monitor will copy some initial image the machine can execute into the guest's memory and mark the domain as runnable. Therefore the domain will be executed once the Barrelfish scheduler selects it.

The monitor will enter an event processing loop where it will wait for VM exits or events from other Barrelfish domains. Other Barrelfish domains happen to be drivers for similar devices the monitor wants to simulate to the guest. For each exit there is handler code to process the exit. This contributes a major part of the core monitor logic. If a driver sends a notification or data, then the corresponding simulator for that specific device is called within the monitor. This virtual device will most probably alter some of its sate and hardware registers, and eventually assert a virtual interrupt to the CPU.

6.5 Memory Management

The monitor is responsible for managing the guest's memory. With nested paging, it has to handle all nested page faults and resolve them accordingly by allocating memory, or call a corresponding virtual device.

Through Barrelfish's vertical design, each user-space domain has to manage its own memory including the required page table. Such a page table is represented by a set of capabilities. These capabilities can be held by any domain. Therefore it is possible for domain to maintain the virtual memory of another domain. The VMkit monitor uses this technique to maintain all the guest's memory. The page table that normally would be used when switching to the guest domain is used as a nested page table. The monitor can therefore simulate any physical memory environment to the guest.

The guest's memory is also mapped into the monitor domain enabling it to examine the whole guest-physical memory. The guest on the other hand is not able to access any monitor memory, which protects the monitor's memory from compromise by the guest. Page faults within the guest are not intercepted by the virtual machine. The guest needs to be able to handle them itself.

Every time a nested page fault occurs, the VMkit kernel will call the monitor. On a nested page fault there are three basic distinctions. Either the requested guest-physical address was within the range of the assigned guest-physical address range or outside of that range. If outside then it may be an access to a memory-mapped device register. If the faulty address is within the range of assigned memory then this memory is allocated through Barrelfish mechanisms and mapped into the nested page table, i.e. the page table of the guest domain. If the guest accessed a virtual device register then the code handling memory access for that device is called. In the third case, where the access is outside the acceptable range and not a device, a fault will be sent to the guest as if it had accessed illegal memory on a real machine.

Using the described mechanism, the majority of the available Barrelfish logic for virtual memory can be reused for the major part of the guest-virtual-memory handling. It is occasionally necessary to resolve a certain guest-virtual memory address through the guest's page table, i.e. every time some guest instruction is examined further by the monitor. However, general guest page table walks, as for example with shadow paging, are not required. Since the guest-physical memory is also mapped into the monitor, the resolution of guest-virtual addresses need not go through the nested page table. This page table is read solely by hardware.

6.6 Communication

VMkit uses Barrelfish's IDC mechanism as method for internal communication, i.e. communication between CPU driver and monitor. IDC messages can only be sent between dispatchers of domains running on the same core, or between the CPU driver and a dispatcher running on the same core. The latter is used by VMkit to implement events from the kernel side to the monitor.

Every time the kernel encounters a VM exit which needs monitor attention, it will construct an empty IDC message and send it synchronously to the guest's monitor. This message will, through the synchronous IDC mechanism, invoke the monitor directly, which then will run on the remaining time of the guest's time slice. Virtual machines can contain only one CPU. For a given pair of a monitor and its guest, only one of the two can run at any given time. Therefore, executing both on the same core does not degrade performance, and IDC messages can be used in all cases.

Communication with other domains within Barrelfish, such as drivers, is done using higher level abstractions of Barrelfish's communication mechanisms. This includes the use of *chips* and *flounder*. For most cases, the monitor does not need to take extra steps, but only uses the provided interface within the client libraries of the devices.

Between the monitor and the guest, communication is done only via virtual interrupts and VM exits. The monitor can notify the guest by sending it a virtual interrupt, which will trap the virtual CPU into an interrupt handler. The guest can, with or without its knowledge, end up in a condition generating a VM exit. Also, no special communication is done between the monitor and the guest's domain. The monitor owns the guest domain to the extent that it shares all its memory, so there is no explicit communication necessary on the host side. Explicit synchronization is also not needed, because the monitor and its guest never run simultaneously.

6.7 Virtual Hardware

The virtual machine provides all essential PC hardware such that a modern operating system can be executed on top of it. There is video adapter support. Only simple video BIOS calls are supported such that a boot loader can print messages before it initializes a serial controller. For some of the hardware, legacy versions of the device were chosen to simplify their emulation.

6.7.1 Interrupt controller

The interrupt controlling device is provided by a legacy *programmable interrupt controller (PIC)*. This device is present in all available PCs sold today. It is the predecessor of the *advanced programmable interrupt controller (APIC)* which is also available in most PCs today. The PIC is a simple device offering 8 interrupt levels, i.e. different interrupt types. It is directly connected to the CPU's external interrupt pin. A PC normally contains two separate PICs offering 16 different interrupt vectors. The second PIC is cascaded through an interrupt line of the first PIC leaving 15 distinct interrupt vectors available for the hardware to use. The PIC is responsible for managing all incoming external interrupts. It will monitor the different sources of interrupts and forward them in a well defined order to the CPU. It will wait for one interrupt to be acknowledged before asserting a new one. This exact behavior is simulated through the virtual device. This piece of software is responsible for all interrupts within the virtual machine and all other virtual devices will call the PIC to assert an interrupt.

In real hardware, a complete interrupt delivery would start with the PIC setting the interrupt request pin to high. When the CPU is ready to take the interrupt it will signal the PIC so, i.e. perform an *INTACK* cycle. The PIC will then tell the exact interrupt vector to the calling CPU. One of the difficulties is the fact that a certain pending interrupt can be replaced by one with a higher priority as long as the PIC does not receive the INTACK signal. Therefore, the exact interrupt number is only defined at the time the CPU PIC receives the INTACK. The same problem is preset with the virtual PIC. It could happen that during the time when the virtual CPU is signaled that there is a virtual interrupt pending a interrupt with higher priority is available. The difference between physical hardware in this case is that a real PIC will only raise an interrupt pin on the CPU when necessary but a monitor will pass all information to the virtual CPU, also the information a physical CPU will only know *after* an INTACK cycle, when it wants to signal that there is a virtual interrupt pending. One could work around this problem by inserting a VM exit just before the CPU would take a virtual interrupt, which would be equivalent with performing the INTACK cycle. However, additional VM exits should be avoided to help overall guest performance, especially if they are not needed as we will see. One advantage of the virtual machine is the fact that it knows at any exit whether the CPU will be ready to take an interrupt when the VM is entered again. Together with the fact that guest and monitor run mutually exclusive, the virtual PIC can *replace* the pending interrupt with a higher-priority interrupt as long as the previously-pending one has not been taken by the CPU. Therefore, the PIC maintains a queue of pending interrupts, and examines the current CPU state at well-defined points during simulation and then asserts a certain interrupt to the virtual CPU.

One of the defined points is when another device asserts an interrupt to the PIC. The PIC will immediately check whether there is an interrupt pending and whether it has lower priority than the newly arrived interrupt. In this case it will assert the interrupt to the virtual CPU. Otherwise it will be added to the queue of pending interrupts. Another point is when the interrupt handler within the guest performs an *end of interrupt* cycle, i.e. signals the PIC that it is finished with dealing with the current interrupt. The PIC will fetch the next interrupt with the highest priority from its queue and assert it to the virtual CPU, or do nothing if there is no interrupt. The same is done when the CPU performs a `hlt` instruction. The `hlt` instruction is used by the operating system to tell the underlying CPU that there is no work to do at the moment, and that it should wait for the next event to come from hardware. In virtual machine terms, this simply means the guest domain shall be taken off the ready list and not scheduled until a virtual interrupt arrives. Therefore, every time the `hlt` instruction is issued on a virtual CPU, the VM exits and the PIC is invoked to check whether there is an event pending. If not, then nothing needs to be done, and the systems sleeps until some driver produces a state change within the monitor.

6.7.2 Timing and counting devices

A modern operating system needs some type of timing device to implement its process preemption policy. Physical timing hardware usually uses an oscillator with a predefined frequency. This provides an accurate and contiguous time experience.

Within a system hosting a virtual machine, the monitor and its guest do not get all the time of the available CPU. Time will continue to run, if neither of the two are running currently. Therefore, the guest experiences jumps in time, i.e. time will not be contiguous anymore.

Programmable interval timer

The *programmable interval timer (PIT)* is a legacy device available in every PC. It offers three counters. The first of these counters can be configured to trigger a timer interrupt every time it reaches a counter value of zero.

VMkit provides a virtual PIT to the guest. The timer interrupt is implemented using a timer on the host system. Every time the guest reconfigures its virtual timer, the monitor will adjust the driver within Barrelfish to send it a message once the time the virtual timer has to wait for has elapsed. Once this happens, it will assert a timer interrupt to the virtual CPU.

Counters are implemented through a host-wide counter, which simply increments since the system was booted. This value is translated to the guest counter value each time the guest accesses the virtual counter registers.

Both virtual timer and virtual counter do not take the elapsed time into account when neither the guest nor the monitor is running, and therefore the guest's time experience will be different compared to experience on physical hardware. System software without real-time constraints should not have a problem with this.

Real time clock

The *real timer clock (RTC)* is used to provide a source for wall-clock time within a computer. It is normally connected to a battery and runs even if the system is turned off. The RTC version usually built in to PCs also offers some RAM which can be used freely by the operating system to store persistent data.

VMkit provides a virtual RTC to its guests but because Barrelfish has no notion of wall-clock time yet it simply starts at some predefined date and updates that date every second. The non volatile RAM (NVRAM) portions of the RTC are neither forwarded to the real RTC nor stored on another persistent storage device. Therefore, any data stored in the NVRAM will be lost when the virtual machine is shut down. The guest can choose to receive an interrupt on every RTC update or at some predefined alarm time.

6.7.3 Serial controller

The guests running on top of the virtual machines may be controlled through the use of the virtual serial port. To that end, the virtual machine offers a standard 16550 UART controller. It follows the specification of the National Semiconductor PC16550D UART [25]. FIFO support was left out. Transmitting FIFOs do not help to increase performance, since the virtual machine will experience a VM exit every time a byte is pushed into that FIFO so it can pass that byte along to the underlying real serial port or whatever device is virtually connected to the port, like the system console. The underlying device will take care of eventual real FIFOs. The same argument is true for receiving FIFOs. Every FIFO read needs a VM exit, and if the guest refuses to read serial input despite being notified of its availability, the underlying subsystem used to provide the serial input will take care of the buffering. Another argument is that, in contrast to real hardware where data can arrive at the serial port in parallel to the CPU executing unrelated instructions, it is not possible on a virtual machine toachieve that parallelism where either the monitor runs to simulate some hardware actions, or the guest runs.

The virtual machine offers four serial ports, all located at their legacy positions in IO address space and using their assigned interrupt vectors like in legacy PC setups. As a special case, the first serial port is hard wired to Barrelfish's console and will receive all input the user types into that console. Correspondingly, all output written to that serial port will show up on the user's console.

6.8 Real Mode and System Boot

VMkit does not boot a BIOS like a real machine would, but provides GRUB [11] as a entry into the virtual machine. GRUB is a universal boot loader capable of booting many different kernels. It is generic and provides a human-usable interface.

Most of GRUB's code runs in protected mode, but it relies rely on the BIOS to drive the most essential parts of the system, such as the hard disk controller. Instead of running a BIOS underneath GRUB, VMkit simulates all BIOS calls necessary to make GRUB work. The instruction raising a software interrupt at the CPU (`int`), is intercepted only if the virtual machine is running in real mode. The monitor then examines the exact parameters to this BIOS call and performs the desired action like a normal BIOS would but on a higher level, i.e. not by talking to low level interfaces of devices but using the high-level API provided by Barrelfish.

Many of these calls are reused once Linux boots. Linux also boots in real mode and fetches information from BIOS calls. This includes information about the memory structure of the system. The virtual machine tells the guest about the memory boundaries the monitor has chosen for that machine, which will later also be reflected in the nested page table. Early access to a VGA console is also provided through those calls to enable operating systems to print text before they could set up a serial connection.

Although SVM would be able to virtualize the real mode directly on hardware, an emulator is used during real mode execution. This should simplify the port to the Intel variant of CPUs with virtualization hardware extensions, since they do not support running the virtual machine in real mode.

6.9 Capabilites

VMkit uses Barrelfish's capability system to perform privileged operations on virtual machines.

In the current implementation there is a capability which is given to the monitor process. This capability enables the process to create virtual machines, i.e. set up memory regions to store the various control structures needed by the hardware or shared between kernel and the monitor. The monitor needs to tell the kernel about these regions which is also done through this capability. After setup the monitor may make the guest domain runnable and from there on the kernel will take care that the domain is dispatched correctly.

This model is simple but unfortunately insecure. The monitor stores the different data structures within different memory locations. These data structures contain host-physical memory-address references to each other. The monitor could use any address to reference them which could jeopardize the security of the system. Therefore, using this model, the whole monitor is part of the trusted computing base which is undesirable. It is however very convenient because the monitor can simply map all the locations holding those data structures into its memory and manipulate them directly.

A more secure approach would be retyping a RAM capability of sufficient size into a virtual machine capability for each virtual machine. This capability would hold all data structures which are either shared between kernel and monitor or needed by the hardware and therefore contain privileged information. An interface between the kernel and the monitor would enable the monitor to manipulate relevant portions of these structures, with the kernel always checking for their security. Using this technique, the privileged portion of the virtual machine control is not leaked into the monitor which will not be able to jeopardize the host with ambiguous memory references. However, the kernel portion VMkit would grow because some methods necessary to manipulate these data structures would be moved from the monitor to the kernel.

Chapter 7

Evaluation

In this chapter I evaluate how well VMkit meets the aim in section 1.2. VMkit should be simple and secure while staying competitive with regard to performance to other comparable virtualization solutions.

In the following evaluation, VMkit is compared against KVM [20], described in section 3.5. KVM is based on Linux and, like VMkit, is able to run unmodified guests using the hardware extensions provided by AMD. It is therefore the closest available virtualization solution of those enumerated in the literature review (chapter 3).

First the structure is evaluated for security and simplicity patterns followed by a analysis of VMkit's performance.

7.1 Simplicity and Security

VMkit is lightweight. It adds fewer than 500 lines of code to the CPU driver part of Barrelfish. Most of the code within the CPU driver is used to perform the privileged portion of the world switch and the monitor notification mechanism. A minor part implements the fast-path logic used to improve the performance of VM exits and external interrupt handling.

The user-space part of the monitor is about 6000 lines of code. VMkit does not yet support more virtual hardware than needed by operating systems designed to run on the PC platform. Therefore, one can say that the major part of the code is used to implement the core monitor logic. VMkit also does not support the Intel virtualization extensions. Adding them would increase the monitor and the kernel portion considerably.

KVM implements much of its functionality within the Linux kernel. This includes virtual interrupt handling and performance-critical devices, namely the interrupt controller, the programmable interval timer, and the memory management unit. All VM exits caused by accesses to these devices will not be forwarded to user-space code. The kernel part contains an x86 and x86-64 emulator to emulate instructions which are not virtualized. Summing up, KVM adds about 10k lines of code to the Linux kernel, which contains about the same functionality as VMkit provides with the complete implementation, including kernel and user-space parts.

The user-space part of KVM, QEMU, supports many more features than

VMkit's monitor. Its code base is therefore considerably bigger and not directly comparable to VMkit.

From a security perspective, VMkit's design is a lean solution where only a very small portion of the code is run in privileged mode. Guest and monitor are separated, and therefore the monitor is protected from corruption through the guest. KVM has, as described above, considerably more code running within privileged mode and is therefore more prone to software errors within its kernel part. Its guest is run together with QEMU in the same user-space process. Therefore, it may be possible for the guest, given a software error in QEMU, to destroy its monitor. This is however not fatal for the entire host's system security.

As described in section 6.9, the current implementation of VMkit's monitor moves the whole user-space part of the monitor into the trusted computing base of the system. The solution to that issue is also described within that section. But even with the monitor within the TCB, there are still fewer lines of code within the TCB than in KVM.

7.2 Performance

This section evaluates some of the performance aspects for virtual machines of VMkit against KVM [20].

First the test environment used to run all the benchmarks is presented. After that the relation of KVM and VMkit with respect to performance is evaluated through a microbenchmark measuring the cycles needed to perform a world switch, and a case study measuring the time needed to compile a Linux kernel.

7.2.1 Test environment

The system used to test VMkit contains 4 quad-core 2.5GHz AMD Opteron 8380 processors where each core has a 512kB L2 cache and each processor has a 6MB L3 cache shared by all 4 cores. The system has 16GB of main memory. Since the virtual machines only provide one CPU to their guests, only one of the cores of the test system is used.

The base system used to run KVM on is Linux kernel version 2.6.30.5 compiled to run on x86-64 CPUs. QEMU 0.10.5 is used to control KVM and provide the virtual hardware.

The virtual machine guest used to run the tests is also based on Linux 2.6.30.5. It is compiled to run on x86 CPUs. The reason why x86 was chosen over x86-64 is because x86 is easier to virtualize, and because guest operating systems assume the presence of devices, such as an APIC, which VMkit does not yet support. The guest is assigned 2GB of real memory.

VMkit does not yet support virtual PCI buses and therefore does not contain any virtual hard disk controllers. Fortunately, Linux has the ability to load its entire root file system out of a RAM disk which the boot loader puts into the guest's RAM prior to executing the Linux kernel. Using this method, the guest can run all tests using only the virtual hardware described in section 6.7.

Some results are compared to the performance when the guest is running on physical hardware. In this case, the operating system will also load its file

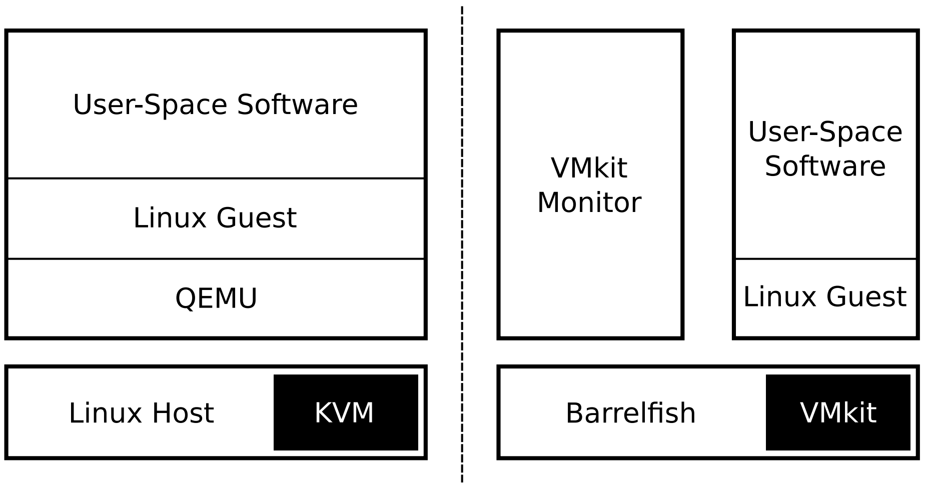

Figure 7.1: The test environment. The KVM test system uses Linux as host and QEMU as the controlling software within the guest. VMkit uses Barrelfish as host and the VMkit monitor as the controller of the VM. Both use the same virtual machine guest which consists of a Linux kernel and user-space software intended to run the desired benchmarks.

system into RAM, and will not use more devices than when inside the virtual machine.

7.2.2 Microbenchmark: World switch

The world switch, i.e. performing either a VM enter or a VM exit, is one of the operations which will degrade the performance of a virtual machine compared to a real machine, because a real machine can perform the same actions without the need of world switches. It is therefore not possible for a virtual machine running unmodified guests to achieve the same performance as the same guest would on a real machine. To get as near to that limit as possible, a virtual machine must reduce the total number of world switches needed and reduce the cost of each world switch.

To examine the cost of a world switch, one can measure the cycles needed to perform a VM enter or a VM exit. I measured the cycles needed to perform a `vmmcall` instruction within the guest. The `vmmcall` instruction can be executed within all SVM guests. It will immediately perform a VM exit and signal the monitor that `vmmcall` was executed. Neither KVM nor VMkit do anything else but perform a world switch back to the guest.

The test application will read the current time stamp counter, issue `vmmcall`, and read the current time stamp counter again. It will repeat the process ten million times and store for each `vmmcall` the difference between the starting and ending time stamp counter. The time stamp counter is a 64-bit register in the x86 architecture which is increased by one on each CPU cycle.

Table 7.1 shows the results of the measurements. The raw data shows a few large outliers which cause the standard deviation to be high. These outliers show up with the frequency of the timer interrupt. The host running Barrelfish uses

`vmmcall` latency (cycles)	Mean	Median	Standard deviation
VMkit (raw data)	7561	7510	2205
VMkit (cleaned data)	7538	7510	95
KVM (raw data)	2222	2139	1929
KVM (cleaned data)	2165	2139	573

Table 7.1: Latency of a `vmmcall` instruction. VMkit performs an IDC into user space on every `vmmvall`. KVM handles the call within the kernel. Cleaning the data, i.e. discarding the 1000 smallest and biggest latencies reduces the standard deviation significantly.

two CPU cores. As soon as there is more than one core running, the Barrelfish monitor, running on the same core as the VMkit monitor, will poll its URPC channels on every timer interrupt, causing the additional latency. Therefore, the raw data is sorted and the first and last 1000 entries are removed. After this clean up, the standard deviation is small within VMkit. KVM still shows an increased standard deviation. Most parts of the Linux kernel, including KVM, are preemptable and therefore the scheduling within the kernel cannot be predicted and will increase the variance of latency measurements.

VMkit, using its standard procedure for non-interrupt related VM exits, needs significantly more cycles than KVM to perform the two world switches. Due to the structure of VMkit, every VM exit which requires the monitor's attention will take the guest from the runnable queue and notify the monitor using an IDC. The monitor will not take a particular action on a `vmmcall`. It will make the guest runnable again and yield the remaining cycles from the current time slice to the guest domain. This adds a context switch from the guest to the monitor and one back from the monitor to the guest. KVM however handles the `vmmcall` instruction within the kernel and does not perform any context switch. The additional cycles used by VMkit to perform the world-switch are the cost for the separation of monitor and kernel.

7.2.3 Case study: Linux kernel compilation

Software compilation is a test used in operating system and virtual machine performance evaluation. It causes significant load on the file system to read the source files and write the results, the CPU to compute the results and the optimizations, and the memory to store the various data structures. In our case, the file system will be located in RAM and therefore cause fewer VM exits than if it would access some virtual device.

In this evaluation, the compilation time of a Linux kernel 2.6.30.5 is measured. The kernel configuration used enables nearly all features. This results in a test which runs for about 30 minutes. The same benchmark is performed on native Linux, inside a VMkit virtual machine, and inside a KVM virtual machine.

Since the virtual machines used to perform this test do access little of the hardware a standard PC offers, the virtual devices used most are the timers, interrupt controller, and, if the test produces output, serial controller. For per-

Linux kernel compilation	VMkit	KVM	Linux native
Elapsed time (s)	2022.295	1939.581	1664.627
Relative to native Linux	1.215	1.165	1.0
Number of VM exits caused	34051539		

Table 7.2: Time needed to compile a Linux kernel on native Linux, a guest running on VMkit and a guest running on KVM.

formance reasons, KVM implements the virtual timer and interrupt controller within the Linux host kernel, and therefore no switches to QEMU are necessary on access of them. However, for simplicity and security reasons, VMkit only puts the most relevant parts into the Barrelfish kernel which does not include any virtual devices. They are all provided through the monitor domain.

Table 7.2 shows the results of the measurements. The time needed to perform the compilation is smallest on native Linux. This was to be expected because the whole virtualization overhead, such as world switches and device emulation, are not necessary. The table shows that, compared to the run time on native Linux, KVM adds 16.5% and VMkit 21.5% to the total elapsed time to perform the compilation. VMkit needs roughly 82 seconds more than KVM. The table also lists the number of VM exits, and therefore monitor invocations, which occurred during the compilation. This number, multiplied by the difference of the world-switch latencies shown in table 7.1, will yield the minimum number of cycles used to perform all calls into the virtual machine monitor. Divided by the frequency of the CPU core, which is 2511.393MHz, this results in a contribution of roughly 72 seconds to the overall time. Out of the 82 seconds difference in elapsed time, 72 seconds are used for the increased world-switch latency. The remaining 10 seconds correspond to a 0.5% fraction of the total elapsed time, and are within the boundaries for other minor implementation differences between KVM and VMkit. Therefore, the additional time used by VMkit can be reduced to the additional cycles needed to perform a world switch, which is caused by the separation of kernel and monitor to increase system security.

7.3 Summary

This chapter compares VMkit to KVM [20] with respect to simplicity, security and performance considerations. It shows that VMkit adds fewer lines of code to the trusted computing base than KVM. It also shows that VMkit's performance is within the range of KVM.

Chapter 8

Conclusion

VMkit provides a virtual machine for Barrelfish. It is capable of running unmodified guests using the hardware virtualization extensions provided by AMD.

VMkit is designed to maximize the separation between its components, increasing the host system's security. It is implemented using simple mechanisms. It uses as much as possible of the available technology from Barrelfish, such as the protection model for guest domains, memory management, and communication. It makes use of the capabilities provided by Barrelfish to control virtual machine execution and all resources. The number of changes to the kernel is minimized to contain only logic which needs to be executed in privileged mode, and code which is used to implement the optimization to handle external interrupts and the dispatch of the monitor, which is invoked directly on a VM exit.

Currently, VMkit supports 32-bit x86 guests using a limited set of virtual devices. This includes am MMU, interrupt controller, programmable interval timer, and a real-time clock. It features one virtual CPU. To communicate with the world outside the virtual machine, a serial controller can be used. Currently, only Linux has been tested as a guest. The BIOS emulation is tailored to support GRUB. For this emulation, a simple virtual hard disk is provided.

The evaluation of the performance of these features has shown that VMkit does not add a considerable overhead to guest execution relative to comparable virtualization solutions. It was evaluated against KVM, which adds roughly 5% less overhead, compared to the execution on physical hardware, than VMkit.

8.1 Future Work

VMkit's usefulness is limited with the current implementation. First of all, it does not support a virtual PCI bus, which is essential to connect non-legacy devices, such as network cards, hard disk controllers, and video cards, to the system. Many of these devices also need a working DMA implementation, which can be achieved using the IOMMUs available today. The PCI bus is a complicated device, and its implementation would require some work, but it could easily be integrated into the current monitor and connected to the virtual interrupt controller.

VMkit currently supports the execution of 32-bit guests. The main reason for this decision is the fact that Linux allows more legacy hardware to be used in a 32-bit machine than in 64-bit. This is especially true for the local APIC and the IOAPIC. Apart from these missing virtual devices, all support needed to execute 64-bit guests is already included in VMkit.

Intel and AMD provide different virtualization extensions within their versions of the x86 CPUs. VMkit supports the AMD variant. Adding support for the Intel variant would require an extension of the interface between the kernel and user-space part of VMkit. In contrast to AMD, Intel offers special privileged instructions to manipulate the data structure holding all state and settings for a particular virtual machine guest. The current implementation of the monitor has the controlling data structures mapped into its virtual address space and manipulates them through ordinary memory accesses.

Intel support could be combined with fixing the security leaks present in the current use of capabilities, described in section 6.9. VMkit does not yet employ the standard Barrelfish technique to store structures critical to system security. Using this method, one would retype a capability referring to some region of RAM into a capability referring to a virtual machine control block, which must be protected from uncontrolled access by user-space domains. To enforce such protection, special invocations could be added to this capability type, which allow all modifications needed by the monitor. This additional interface between VMit kernel and monitor could be designed in such a way that the difference between AMD and Intel's version of the virtual machine control block is hidden as much as possible.

VMkit currently supports one virtual CPU. However, Barrelfish is designed to run on heterogeneous multi-core systems. Running only on one core simplifies memory allocation, since one can allocate the available memory regions with the best performance properties to the running core. Supporting multiple virtual CPUs would pose interesting problems. VMkit could try to hide all the effects, such as non-uniform memory access, and inter-core communication latency and throughput from the guest operating system, and try to find an optimal core and memory usage pattern to optimize the virtual machine performance under the hardware assumptions the guest is optimized for. Another possibility would be to provide an interface allowing the guest to gain knowledge about the latency and throughput properties of the physical hardware. Guests would have to be modified to make use of such an interface.

An important use case for Barrelfish would be using the large number of drivers available in commodity operating systems to control certain physical devices and export their features back into the host. It would reduce the effort needed to use devices which are not critical to performance. To this end, VMkit could be improved to offer guests special memory regions and communication channels to transfer data and events on a higher level than virtual hardware. The guest would need to be modified to export the hardware it manages back into the host over these channels. The monitor process would act as an ordinary driver within Barrelfish, offering interfaces to access the different devices. To gain the maximum performance, the guest would need to be able to access the physical hardware directly. The monitor could make use of the IOMMU features available in recent computers.

Bibliography

[1] D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu, and J. Wiegert. Intel Virtualization Technology for Directed I/O. *Intel Technology Journal*, 10(03):179–192, August 2006.

[2] K. Adams. A comparison of software and hardware techniques for x86 virtualization. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural*, pages 2–13, 2006.

[3] Advanced Micro Devices, Inc. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, chapter 15, Secure Virtual Machine. September 2007.

[4] Advanced Micro Devices, Inc. *AMD I/O Virtualization Technology (IOMMU) Specification.* February 2009.

[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 164–177, 2003.

[6] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The Multikernel: A new OS architecture for scalable multicore systems. In *SOSP '09: Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, 2009.

[7] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX 2005 Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[8] M. Ben-Yehuda, J. Mason, O. Krieger, J. Xenidis, L. Van Doorn, A. Mallick, J. Nakajima, and E. Wahlig. Utilizing IOMMUs for Virtualization in Linux and Xen. In *Proceedings of the Linux Symposium Ottawa*, July 2006.

[9] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. Lightweight Remote Procedure Call. *ACM Trans. Comput. Syst.*, 8(1):37–55, 1990.

[10] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. User-level interprocess communication for shared memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(2):175–198, 1991.

[11] E. S. Boleyn. GRand Unified Bootloader. `http://www.gnu.org/software/grub/`. [Online; accessed 31-August-2009].

[12] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. *ACM Trans. Comput. Syst.*, 15(4):412–447, 1997.

[13] E. Bugnion, S. W. Devine, and M. Rosenblum. System and method for virtualizing computer systems, 1998. US Patent.

[14] M. Chapman and G. Heiser. Implementing transparent shared memory on clusters using virtual machines. In *USENIX Annual Technical Conference*, pages 383–386, 2005.

[15] R. J. Creasy. The Origin of the VM/370 Time-Sharing System. *IBM Journal of Research and Development*, 25(5):483–490, 1981.

[16] J. Dike. A user-mode port of the Linux kernel. In *ALS'00: Proceedings of the 4th annual Linux Showcase & Conference*, 2000.

[17] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the Xen virtual machine monitor. In *Proceedings of the 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS)*, 2004.

[18] H.-J. Höxer, K. Buchacker, and V. Sieh. Implementing a User-Mode Linux with minimal changes from original kernel. In *Proceedings of the 2002 International Linux System Technology Conference*, pages 72–82, 2002.

[19] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide*, chapter 20–29, Virtual-Machine Extensions. June 2009.

[20] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the Linux virtual machine monitor. In *Ottawa Linux Symposium*, pages 225–230, July 2007.

[21] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH multiprocessor. In *ISCA '98: 25 years of the international symposia on Computer architecture*, pages 485–496, 1998.

[22] J. LeVasseur, V. Uhlig, M. Chapman, P. Chubb, B. Leslie, and G. Heiser. Pre-Virtualization: soft layering for virtual machines. Technical Report 2006-15, Fakultät für Informatik, Universität Karlsruhe (TH), July 2006.

[23] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz. Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In *OSDI '04: Proceedings of the 6th Symposium on Operating System Design and Implementation*, pages 17–30, 2004.

[24] D. J. Magenheimer and T. W. Christian. vBlades: optimized paravirtualization for the Itanium processor family. In *VM'04: Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium*, 2004.

[25] National Semiconductor, Inc. PC16550D Universal Asynchronous Receiver/Transmitter with FIFOs. June 1995.

[26] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. In *SOSP '73: Proceedings of the fourth ACM symposium on Operating system principles*, volume 7. ACM Press, October 1973.

[27] J. S. Robin and C. E. Irvine. Analysis of the Intel Pentium's ability to support a secure virtual machine monitor. In *SSYM'00: Proceedings of the 9th conference on USENIX Security Symposium*, 2000.

[28] J. Sugerman, G. Venkitachalam, and B.-H. Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 1–14, 2001.

[29] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. M. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith. Intel Virtualization Technology. *Computer*, 38(5):48–56, 2005.

[30] A. Whitaker, M. Shaw, and S. D. Gribble. Denali: Lightweight Virtual Machines for Distributed and Networked Applications. In *Proceedings of the USENIX Annual Technical Conference*, 2002.