Diploma Thesis

Lightwheight Virtualization on Microkernel-based Systems

Steffen Liebergeld

31. Januar 2010

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

Betreuender Hochschullehrer: Prof. Dr. rer. nat. Hermann Härtig
Betreuender Mitarbeiter: Dipl.-Inf. Adam Lackorzyński

Technische Universität Dresden
Fakultät Informatik

AUFGABENSTELLUNG FÜR DIE DIPLOMARBEIT

Name des Studenten: **Steffen Liebergeld**

Studiengang: Informatik
Immatrikulationsnummer: 3013853

Thema: *Leichtgewichtige Virtualisierung in Mikrokernsystemen*

Zielstellung:
Traditionelle Betriebssysteme sind aufgrund ihrer Komplexität und den damit einhergehend nicht auszuschließenden Sicherheitslücken keine geeignete Plattform für Anwendungen mit hohen Isolationsanforderungen. Mikrokernsysteme bieten mit ihrer potentiell kleinen vertrauenswürdigen Basis (Trusted Computing Base) eine Alternative. Entscheidend für die Anwendbarkeit von Mikrokernsystemen ist dabei, inwieweit Komponenten voneinander isoliert werden können und unter welchen Leistungseinbußen diese Isolation realisierbar ist.

Eine aufgrund ihres Nutzwertes sehr relevante Klasse von Komponenten sind komplette Betriebssysteme. Ihre Isolation ist eine besondere Herausforderung, da Betriebssysteme sehr spezifische Annahmen über ihre Ausführungsumgebung machen. Grundsätzlich sind Anpassungen auf drei Gebieten notwendig: des Kernsystems bestehend aus Prozessor und Speicher, der Plattform (Timer, Bootup, etc.) und der peripheren Geräte. Bisherige Lösungen konnten Isolation erreichen, taten dies aber auf Kosten von Komplexität und Leistungseinbußen. Die Verfügbarkeit von hardwarebeschleunigter Virtualisierung verspricht eine starke Vereinfachung bei der Prozessor-Speicher-Virtualisierung.

Gegenstand dieser Arbeit ist es zu untersuchen, wie Plattform- und Gerätevirtualisierung in Mikrokernsystemen realisiert werden können. Für einige Interaktionen, die das Gastbetriebssystem auf physischer Hardware tätigt, ist der Implementierungsaufwand im Systemvirtualisierer, dem sogenannten Virtual Machine Monitor, unverhältnismäßig groß. Im Hinblick auf eine Vereinfachung der Implementierung ist es möglich, eine von der physischen Maschine abweichende Schnittstelle zwischen Gastbetriebssystem und VMM zu wählen. Dabei ist darauf zu achten, dass die Größe der 'trusted computing base' für Komponenten mit hohen Isolationsanforderungen nicht anwächst. Der Entwurf ist zusätzlich so zu gestalten, dass zukünftige Erweiterungen, die die Kapselung von Echtzeitgastsystemen erlauben, mit geringem Mehraufwand möglich sind.

Der im ersten Schritt erarbeitete Entwurf ist anhand einer prototypischen Implementierung zu validieren. Als Gastsystem ist ein Betriebssystem zu wählen, das eine nennenswerte Anzahl von Anwendungen unterstützt. Leistungscharakteristika sind anhand von Messungen im Vergleich zu anderen Ansätzen und nicht virtualisierten Systemen zu ermitteln.

verantwortlicher Hochschullehrer: Prof. Dr. Hermann Härtig

Betreuer: Dipl.-Inf. Adam Lackorzynski
Institut: Systemarchitektur
Professur: Betriebssysteme

Beginn: 01. 08. 2009
einzureichen: 31. 01. 2010

Dresden, 20. 08. 2009 *Unterschrift des verantwortlichen Hochschullehrers*

Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig erstellt und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Dresden, den 25. Januar 2010

Steffen Liebergeld

Abstract

Microkernels were invented as a foundation for systems that can be tailored and adhere to strict security requirements. As with any new system, application availability is crucial to its adoption. Reusing a standard OS kernel on top of the microkernel is a pragmatic way of inheriting an OS'es API and with it its applications.

In the past, standard OS kernels were ported to run on top of microkernels. It turned out that substantial efforts were needed to implement the equivalent of a CPU, which was the model the OS originally assumed. Many OS activities such as context switches are security sensitive and have to be implemented with microkernel provided abstractions. Due to the microkernel involvement, rehosted kernels suffer from inherent performance penalties. Virtualization technology promises better performance with less changes to the standard OS. A port of KVM showed that virtualization on a microkernel-based system is feasible and standard OSes running in a VM perform better than rehosted OSes. However, this KVM port in turn depends on a rehosted kernel.

In this thesis, I present a solution that uses virtualization to host a slightly modified Linux on the L4 microkernel Fiasco. This solution shows a significant performance improvement compared to previous solutions, supports SMP, is well integrated and has a substantially reduced resource footprint.

Acknowledgements

I would like to thank Professor Hermann Härtig for allowing me to work in the operating systems group. I want to mention the tremendous help of my supervisor Adam, who always had an open ear whenever I asked him for advise. Thanks go to Torsten Frenzel as well, who provided me with ideas that helped me jump start me implementation. I owe Michael Peter a depth of gratitude, because he provided me with numerous hints for this thesis. Thanks go out to the people that proofread the text and helped me cope with the language. These people are Adam Lackorzyński, Torsten Frenzel, Michael Peter and Jean Wolter.

A good working atmosphere is crucial to my productivity, therefore I want to thank the students of the student-lab, for their companionship and their helpful remarks.

I also want to thank my parents for funding a large part of my studies. Stefanie deserves special thanks, because she was the one who suffered the most because she rarely got to see me due to my work commitment.

Contents

List of Figures

1 Introduction

In the past two decades, computers have evolved into an ubiquitous tool. They are used all over the world for everyday work, for managing finances and for communication. Unfortunately, computer systems are less reliable than they should be. They are vulnerable to attacks that –if successful– may result in the disclosure of confidential information, which, in turn, may lead to serious financial losses. An examination of actual attacks reveals that often weaknesses of the operating system (OS) are to blame. In many cases, attackers compromise a single application and from there widen the exploit to the whole machine. While an OS cannot protect individual applications from being compromised, it should prevent the attacker from reaching out further.

The issues that plague OSes are a consequence of their development history. Early on, users primarily ran applications they knew and trusted. Under the premise that the user trusts its application, there was no need for the OS to restrict the privileges of individual programme instances. Any application started by the user inherited all of his privileges, not only those needed to fulfill its task. Although this mechanism was sufficient at that time, today the usage scenario is different. As computers are hooked up to the Internet nowadays, users often face software the trustworthiness of which they cannot assess. Even worse, software can run without explicit consent of the user. Web-browsers, for example, automatically execute scripts downloaded from websites to enrich the web experience. Although the usage scenario changed, the assumptions that underlay the security mechanisms were never questioned. As such, modern systems have to cope with attacks that they were not designed to resist.

In addition to inadequate privilege mechanisms, the growing complexity in the kernel becomes problematic. Standard OSes were devised at a time when people did not foresee the complexity of today's systems. Over time, the computer evolved into the capable machines that we use today. With every innovation, the OS vendors had to implement new features, thus adding to the OS'es complexity. Complex systems tend to be more prone to bugs, which can reduce the reliability of the system by their own or even be exploited by attackers. Mechanisms that are employed to mitigate the risks of complexity at user level are not readily applicable to kernel components, as address spaces cannot be used to implement protection domains within the kernel.

These flaws are deeply embedded within the OS, which makes attempts to fix them difficult, especially if backward compatibility has to be ensured. Therefore, it is unlikely that the deficiencies of existing OSes will be removed in the near future. Instead, new systems will grow up that will follow new architectural paths.

Because of their architectural merits, microkernels are likely to serve as a foundation for new systems. In microkernel-based systems, the amount of code running in privileged mode is small enough to be thoroughly audited or even validated using formal methods. OS services, such as device drivers and protocol stacks, are implemented in user land

with address spaces providing hardware enforced isolation boundaries. In such a system, faults are contained, i.e. a faulty component such as a failing device driver, cannot bring down the whole machine. Accordingly, microkernel-based systems allow applications with tiny trusted computing bases.

Instead of ACLs or derivatives thereof, modern microkernels use capability based schemes for access control, which foster the adherence to the principle of least authority at system level. That is, applications can be configured with the minimal set of privileges needed for their correct operation. An attacker who seizes control over an application is therefore also restricted to the privileges that were selectively granted. The obvious goal then is, to structure the system into many small components each of which carries minimal privileges.

The improvements on security and availability of microkernel-based systems are paid for with giving up on backward compatibility. Accordingly, only few native applications will be initially available, which makes a migration path that allows gradual adoptions of existing applications desirable.

One solution is to establish an encapsulated legacy environment that can be instantiated multiple times. A pragmatic way of creating such a legacy environment is to reuse a standard OS kernel. Although an attacker may still succeed with its initial attack on a target in a legacy domain, he finds himself encapsulated in the latter. That gives other applications the chance to go unharmed, provided they reside in other domains.

In the past, standard OS kernels were adapted to run on top of a microkernel. The resulting three-tier architecture –microkernel, OS kernel, application– had to run on hardware that supported only two privilege levels. As a result, both OS kernel and application run unprivileged with the OS kernel relying on microkernel services to manage its processes. Security critical operations, such as memory management, have to be mediated by the microkernel. Therefore, porting a standard OS kernel involved in-depth kernel modifications, which require intimate knowledge of the kernel. Due to the microkernel involvement, rehosted kernels face inherent performance disadvantages compared to the native kernel.

The recent addition of hardware virtualization support to commodity processors creates the opportunity to reuse an OS kernel largely unchanged. That not only dispenses with the need for laborious changes but also overcomes the performance disadvantages of the previous approach. In previous work, virtual machines were a synonym of a duplicate of a physical machine (faithful virtualization). Para-virtualization, i.e. virtual devices with no corresponding physical implementation, was only employed if not doing so would result in severe performance degradations. Even though faithful virtualization can host unmodified OSes –an advantage that is of little importance for systems that are available with their sources–, its implementation introduces significant complexity, mainly due to device emulation.

In this thesis, I present a solution that uses virtual machines to run a slightly modified standard OS kernel, and thereby avoids the costs involved with faithful virtualization, both in terms of performance and complexity. In a second step, I will show how functionality of the para-virtualized standard OS can be used to support faithful virtualization as well.

1.1 Structure

This thesis is structured in the following way. First, in Chapter 2, I introduce the benefits of microkernels and their access control scheme. Second, technology that enables the reuse of standard OS applications on top of microkernels will be presented. Chapter 3 introduces important examples that are in use today, and discuss their strengths and weaknesses. In Chapter 4, I will present a design that is of less complexity and offers better performance than previous solutions. Chapter 5 gives insight into the implementation of a prototype that I developed. This prototype will be evaluated in Chapter 6. In Chapter 7, I present three solutions on how real-time (RT) tasks can be supported in the future. The thesis will be rounded up with a short outlook on future projects that are enabled with this work, and a conclusion and summary.

2 Background

In this chapter I will first give an overview of the importance of security in modern computing. Thereafter, I will introduce microkernels as a key technology on the way to more secure computing. The discussion of the security aspect will then be completed by an overview of access control. I will then proceed by introducing virtualization in both its terms. The chapter will then conclude with a short discussion on virtual machine monitors versus microkernels.

2.1 Security

Computers are taking up new importance in our lives. People use their computers to store and manage their private data such as digital pictures, diaries and even do banking. Companies rely heavily on the digital work flow and store huge amounts of mission critical data. The confidentiality of data, meaning that it may only be revealed to authorized persons or systems, has to be enforced at all times because its disclosure may cause substantial economic damage. Similarly, the integrity of data has to be ensured; For a company to rely on data, unauthorized modifications must be disabled or at least be detectable. Another major issue is the availability of data. If a trade in the stock market is delayed, the prices may already have risen, causing losses for the trader. All in all we can say that the demand for security of the information we manage on computers today has risen.

Most of the operating systems kernels in use today, such as Microsoft Windows, Linux and *BSD, are implemented as one single entity. Subsystems communicate by means of procedure calls and shared memory. There are no practical boundaries between subsystems, and we refer to such systems as being monolithic.

An analysis of of the software stack of current monolithic operating systems reveals severe problems concerning robustness and security: An attack to any subsystem such as a network stack can be used to gain control over the full machine. Even the latest incarnation of Windows is prone to these attacks as became visible to the general public when Microsoft issued a security bulletin describing a vulnerability in its file-sharing protocol stack [Cor]. A study on Microsoft Windows XP discovered that the majority of crashes was caused by bad drivers [GGP06]. Such crashes cannot be avoided beforehand because drivers are written by third parties that may not provide the source code for thorough analysis. A solution is to run drivers in an isolated domain, containing faults and errors and therefore preventing a breakdown of the whole system. Such a system may restart failed drivers and thus continue its operation.

In widely used operating systems isolation between applications is insufficient; To trust an application we have to trust the whole software stack, which is unnecessarily large

and makes applications depend on functionality they do not need. We need a system that allows applications with a custom tailored trusted computing base (TCB).

Studies have shown that even well-engineered and reviewed code contains an average of two bugs or possible attack vectors per 1000 source lines of code (SLOC) [MD98]. A minimal Linux kernel already comprises 200.000 SLOC, standard configurations being much bigger [PSLW09].

Asides from the kernel, the X-Server running on top of it contains 1.25 millions SLOC, all of which are executed with root privileges, and –if compromised– allows the attacker access to all data on the system. Applications, such as web browsers are also quite complex; For example Firefox comprises 2.5 millions SLOC. Any bug, such as a buffer overflow, in this huge stack may be used not only to compromise the browser and the web banking session therein, but can also escalate to compromise a user's data [PSLW09].

Because all OS subsystems run in privileged mode without isolation from one another, a monolithic kernel is not a suitable basis for secure systems. Furthermore, it is difficult to retrofit modern access control schemes that would allow more fine grained access control, which might render a user's data less prone to attacks, without breaking backward compatibility. Asides, further technology, for example support for real-time applications is hard to add, as well.

Instead we should seek to create an architecture that enables us to build decomposed systems in which applications with a small TCB can be built and access permissions can be controlled at a fine granularity.

2.1.1 Building Decomposed Systems with Encapsulated Components

One proven way of dealing with the software complexity is modularization and restricting interaction to well defined interfaces. However if all components run in privileged mode no improvement in security is achieved because there are no means of isolation between them. While we have to trust all code running in privileged mode, the situation is different for unprivileged code: Address spaces provide a boundary that can be used to establish strict isolation between processes [Hei05]. By moving subsystems into user-land processes, we can safely decompose the system in a way that faults and errors can be confined to one subsystem only.

The base system, or kernel, that establishes the required isolation between components must be small, because to make sure that the isolation property holds, the workings of the kernel have to be verified either per extensive testing, by code reviews, or even by formal verification. Verification techniques available today scale poorly and are typically limited to hundreds or at best thousands of lines of code [Hei05].

The idea of building a system on top of a small kernel, a **microkernel**, came up in the 1980ies. Early solutions were built with Unix in mind. The most important representative of such a first generation microkernel is Mach. Mach has a number of system calls that geared towards supporting Unix. Moreover, its task and thread structures are pretty much aligned to those of Unix. Inter-process communication (IPC) is asynchronous, which requires in-kernel complexity such as buffer allocation and queue management.

At that time, first generation microkernels were pretty much en vogue in the systems research community. Numerous different operating system personalities such as Unix and

DOS were implemented on top of them, essentially proving the versatility of microkernels. While drivers remained inside the kernel, a new concept was introduced: user-level pagers. With this new concept memory management could now be done in user land [ABB+86].

Unfortunately, Mach and other microkernels showed a significant overhead compared to monolithic systems. Consequently, they did not catch on in the marketplace, and general interest in microkernels ceased [Lie96].

At that time, Jochen Liedtke devised a new generation of microkernels. Contrary to the designers of first-generation microkernels, he started from scratch following the rationale:

> A concept is tolerated inside the microkernel only if moving it outside the kernel [..] would prevent the implementation of the system's required functionality." [Lie95]

He proposed three basic abstractions: tasks, threads and IPC. Tasks are used as isolation domains and are usually implemented using address spaces. All activity inside of tasks is called thread [Lie95]. Threads communicate with each other using synchronous and therefore unbuffered IPC, which proved to be much faster than asynchronous IPC as used in first generation microkernels [Lie94].

According to Jochen Liedtke a microkernel avoid implementing policy wherever possible to retain flexibility. Therefore, user land can run multiple policies in parallel. Such flexibility allows the implementation of a number of different operating system personalities, each with its own policy, and they all can run side-by-side [Lie96]

Moreover, interrupts and exceptions are translated by the microkernel into IPC allowing hardware drivers to be implemented in user land and therefore to run encapsulated with respect to faults and errors [Lie96].

Memory management is done in user land as well: Address spaces are constructed in a hierarchical fashion. Page faults are exposed to the user. A secure mechanism allows page mappings to be created, and an *unmap* operation ensures that memory can be revoked any time without the mappee's consent. Memory management operations except unmap require consent and are therefore piggybacked onto IPC.

The first implementation of such a second generation microkernel written by Jochen Liedtke was called **L4** [Lie96]. Today the name *L4* is used for a family of successors that implement this interface [AH].

Microkernels of the second generation proved to be a suitable platform upon which highly decomposed, secure systems can be built [PSLW09]. Microkernel based systems are regarded as being highly modular, providing good fault isolation, being flexible and tailorable [Lie96].

Microkernels impose a slight overhead compared to monolithic systems: Instead of calling subsystems by method invocations, communication is done using IPC. An IPC call from one user land server to another requires switching from user privilege level to kernel privilege level, possibly switching address spaces and switching back to user level. Thus microkernels introduce transition overhead as well as increasing the TLB and cache footprint. Because there are a number of servers communicating with each other to provide the operating system's functionality, IPC performance is very important. Härtig and colleagues were able to show that a port of Linux onto a modern second

generation microkernel undergoes an overhead compared to native Linux of only 5 to 10% [HHL+97].

2.1.2 Access Control

Microkernels establish isolation between tasks with separate address spaces and allow those components to communicate with IPC; To enforce an access control policy, mediation of IPC is needed. In the following I will describe some of the challenges therein.

In a secure system a software component should be given the minimal set of rights that is needed for operation. Granting processes the minimal set of permissions to fulfill their task is called the **principle of least authority**.

Under certain circumstances processes can be fooled into using the permissions they have in an unforeseen way; The problem was first described with a compiler that was granted the permission to write to a file A for accounting purposes. If the name of file A is handed to the compiler as the target for debugging output, the compiler will overwrite the accounting data. The problem here is not that the compiler exercised an operation it was not allowed to, but that it was fooled into using its permissions in the wrong way. This problem is known as the **confused deputy** problem [Har88].

Imagine a company A such as a bank that wants to buy software from another company B to process confidential data. Now company B does not allow company A to view the source code of the software because doing so would violate its trade secrets. But without being able to fully analyse the software, how can company A be confident that the software will not leak the confidential information? What is needed here is **confinement**, a property of a system that enforces that a process cannot leak information to third parties. So if company A uses a system that implements confinement, it can be confident that the software bought from company B cannot leak the confidential information.

Such problems can only be mitigated with an adequate access control mechanism.

Early L4 microkernels implemented a simple access control scheme: clans and chiefs; Every task belongs to a clan, and may freely communicate with any task therein. One task of the group is the chief. Communication between tasks of different groups must pass through their corresponding chiefs. Therefore IPC has to be dispatched and redirected by the microkernel, which places a significant performance burden on inter-clan IPC. Clans and chiefs is no longer actively used because while being flexible, this access control scheme was costly in terms of run time overhead.

In current microkernels such as OKL4 and Fiasco a new mandatory access control mechanism is used: object-capabilities. With object capabilities access to an object is granted only if the task holds the appropriate capability. A task may grant an access permission to another by mapping the corresponding capability. Similar to memory pages, capabilities can be unmapped without the mappee's consent. Thus an access control policy can be implemented. A user land can be built that implements the principle of least authority, which means that all tasks have the minimal set of permissions needed to fulfill its commission [MI03].

Object-capabilities unify access permissions and designators. Thus systems using object-capabilities are less prone to the confused deputy problem than systems that employ two different mechanisms such as Access Control Lists [MI03].

Communication also requires capabilities. Therefore the confinement problem is solved, because a task can only communicate with tasks that it has been given the capability [MI03].

2.2 Virtualization

In the last years virtualization has become popular in the low-end server and even in the consumer market for a number of reasons, most of which are due to inabilities of current operating systems:

The workload on typical server machines does not fully utilize the machine and demand for computation time comes in bursts. Therefore virtualization is used to consolidate several underutilized servers into one physical machine, whereby energy is saved and space in the data center is freed. Virtual machines can be easily deployed and removed, which gives the operators more flexibility, and allows product testers to set up a number of machines each with different configurations.

With live migration techniques improved load balancing and fault tolerance can be achieved. By moving a virtual machine away from an overloaded or faulty server. Fault tolerance may also be improved by check pointing virtual machines; If the virtual machine fails, it can be restored from an earlier checkpoint.

Operating system developers may use the inspection mechanisms provided by virtualization to develop and debug systems. The same inspection mechanisms can be used to monitor the virtual machines behaviour and alert the operator on any misbehaviour [CN01].

End users use virtualization to run applications written for different operating systems concurrently. Virtual machines may also be used to run outdated operating systems developed for architectures that are no longer physically available. This may enlarge the lifespan of important legacy applications. Furthermore virtualization enables legacy applications to run on new operating systems, which may be a way to provide a migration path and thus foster the adoption of the new operating system.

In the following chapters I will give an introduction to virtualization including definitions on terms that will be used throughout this thesis.

2.2.1 Virtualization Basics

One early attempt at properly defining the term *virtualization* was done by Popek and Goldberg in 1974[PG74]. As the basis for their considerations they used a processor architecture with two different modes of execution: Privileged mode that has the full set of instructions available and unprivileged mode that only allows a subset of instructions. This setup is typical of microprocessors used in machines intended for multi-user operation.

They defined a **virtual machine** (**VM**) as an *efficient, isolated duplicate* of the real machine that is established by a control program, which they call a **virtual machine monitor** (**VMM**). According to their definition a VMM has three important characteristics: First, it provides an environment for programs that is essentially identical to the original machine; Second, programs running in that environment shall have little decreases in execution speed; Finally that the VMM is in full control of the system's resources.

The first characteristic means that programs running in the virtual environment must produce the same results as when run on a real machine. Temporal behaviour however

cannot be recreated because the timing of the program flow is altered by all interventions of the VMM. The efficiency requirement demands that a statistically dominant subset of the virtual processor's instructions runs directly on the physical processor. The third characteristic, resource control, means that the program running inside the VM should be able to use only those resources that it has been explicitly granted access.

Further definitions by Popek and Goldberg include:

Trap: A *trap* is an unconditional control transfer from unprivileged to privileged mode executed by the processor.

Privileged instruction: A *privileged instruction* may only be executed in privileged processor mode. If its execution is attempted while the processor is in an unprivileged mode, the execution must cause a trap.

Sensitive instruction: The term *sensitive instruction* refers to any instruction that has influence on the authority of the control program in one of the following ways:

- changes the processor mode without trap
- attempts to alter the amount of resources available
- produces different results when run in different processor modes or in different locations

With these definitions in place they draw the following conclusion:

> "For any conventional [..] computer, a virtual machine monitor may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions"[PG74].

In other words, a computer architecture is *virtualizable* if all sensitive instructions trap.

If an architecture is virtualizable, a VMM can be built in the following way; Guest code, both privileged and unprivileged, runs in unprivileged mode on the host. When sensitive instructions trap, the VMM steps in to emulate them and thus recreate their native behaviour. Thereafter it returns control to the guest. This scheme is called **trap and emulate**.

2.2.2 Nomenclature

All code running inside the VM will be called the **guest**, as opposed to the **host**, which hosts the VMs.

All host code running in privileged mode will be referred to as the **hypervisor**. All code controlling the VM execution will be called the VMM. VMM and hypervisor must not necessarily be different entities, but can be combined. The combined system will also be called hypervisor.

2.2.3 Virtualization of the x86 Architecture

The x86 instruction set includes a number of sensitive instructions that are not privileged and do not trap when executed in unprivileged mode. Therefore privileged code of a VM cannot run in unprivileged mode without loss of equivalence. Therefore the x86 instruction set is not virtualizable[RI00].

The remainder of this chapter will focus on virtualization on such a platform, and will be followed with an overview of hardware extensions that were created to aid virtualization.

2.2.3.1 Software Virtualization

There are essentially two approaches virtualization on architectures that are not virtualizable: Either privileged code has to be emulated or the VMM has to analyse guest privileged code on the fly and replace sensitive instructions that do not trap with instructions that do. I will classify the first solution as *emulation* and *binary translation*, and the second one as *patching*.

Unfortunately both techniques incur a serious performance deterioration: Emulation of code is typically by factors slower than its native execution. Causes for this slowdown are inherent: Under emulation the whole CPU state has to be duplicated in software and the emulation of an instruction must produce the same result as when run on the physical CPU. Typically, to emulate one instruction, a number of host instructions have to be executed, a fact known as *instruction inflation*. Instruction fetch requires extra effort inducing *loop overhead* that can account for a fair portion of the overall overhead. As such, emulation does not fulfill the *efficiency* criterion by Popek and Goldberg.

Binary translation is a dynamic translation of guest binary code into host code that takes place at run time. The result is host code that often is a subset of the x86 instruction set, for example user mode instructions only. Translation is done only when code is to be executed (lazy), and cached to speed up subsequent executions. The cache for translated code has to be invalidated upon writes, thus overhead arises from cache management.

In both emulation and binary translation host CPU registers have to be used both to keep administrative data of the emulator and register contents of the guest forcing *register pressure* that requires additional memory accesses to store and load register content.

Today's hardware is able to do concurrent computations for example in pipe-lined execution and simultaneous address translations. In emulation these operations are run serially.

Patching techniques achieve better performance than emulation but require thorough analysis and replacement strategies that make the technology more complex. In practice shortcuts were implemented to increase performance. However, such shortcuts deviate from the x86 interface and do not work in the general case.

Operating systems have to be in control of the memory management of the machine and rely on segmentation and paging to achieve memory protection. A VM however is not allowed to manipulate host page tables and segments. Page table and segment

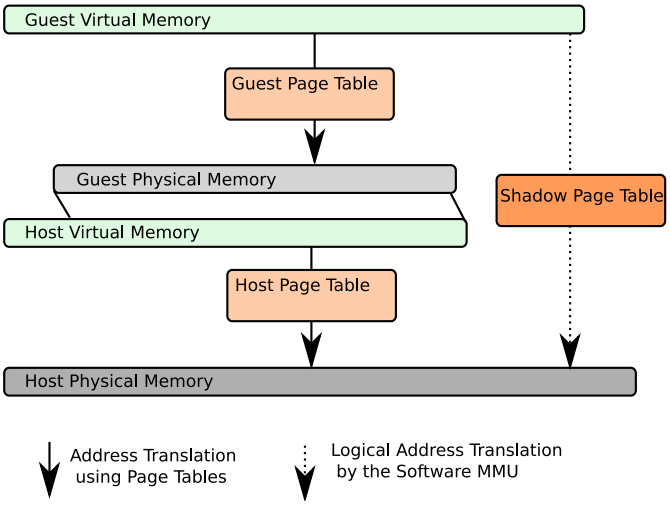

Figure 2.1: Illustration of a shadow page table that maps guest virtual to host physical addresses.

manipulations are privileged operations that cannot be delegated to a VM because it would enable the VM to manipulate kernel memory an thus seize control. Therefore VMMs on x86 have to emulate the memory management unit (MMU).

The guest memory is usually virtualized by multiplexing host address spaces in a way that provides the VM with the illusion of being in control of the memory management. This requires the VMM to be involved in any memory relevant operation of the VM and to keep track of mappings of VM memory to host memory. In order to do this, the VMM has to implement an additional page table that maps guest virtual addresses to host physical addresses. Such a page table is called a **shadow page table**, an example of which is shown in Figure 2.2.3.1.

2.2.3.2 Hardware Virtualization

During the last decade, together with the increasing power of CPUs, virtualization technology gained significance; VMware was the market leader because they were the only ones who could do efficient virtualization of x86. The entry barrier for other contenders in the x86 market was exceptionally high. When the x86 was enhanced to 64bit, legacy mechanisms like segmentation were intentionally left out. Unfortunately that also broke VMware's virtualization technique, which used segments to protect the hypervisor [pag]. Intel and AMD, two major contenders in the x86 market, therefore investigated hardware extensions to aid virtualization and thereby lower the entry barrier to the virtualization market. They came up with similar, but differently named solutions. In this thesis, I use the notation of AMD.

Modification of the instruction set was not an option as it would break backward compatibility. A possibility would have been to introduce a new processor flag that, if

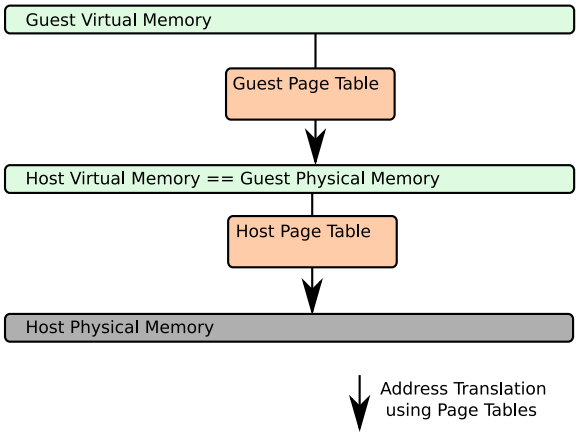

Figure 2.2: MMU virtualization without VMM intervention: Nested paging.

set, causes all sensitive instructions to trap. This would enable VMMs implementing the trap and emulate scheme as proposed by Goldberg and Popek.

Instead, the CPU state was duplicated with the introduction of a less privileged processor mode that is called the **guest mode** in contrast to the standard processor mode, which is now called **host mode**. A switch between guest and host mode is fairly complex because the processor state that is not or only indirectly accessible, for example segment registers, has to be stored and exchanged. A transition to guest mode is initiated with the privileged instruction **vmrun**.

The CPU can be configured to intercept certain instructions, faults and interrupts while executing in guest mode. An intercept results in an unconditional fallback to host mode (**VM exit**) that is augmented with information about its cause. Intercepts may be used to let the host receive and handle interrupts to remain in control over the machine. They can also be used to let the VMM assist in operations that are not handled by hardware virtualization (for example device virtualization). A **vmmcall** instruction is available in guest mode to voluntarily initiate a VM exit.

To aid in removing the performance bottleneck imposed by emulating the MMU, both AMD and Intel created another hardware extension that adds another stage in hardware address translation and thus mitigates the need for the VMM to intervene on guest page table manipulation. This hardware extension is called *nested paging*. An host address space is used as guest physical memory upon which the guest can build address spaces: The address resolution is done by the MMU, and involves parsing of the guest page table to translate a guest virtual address to the guest physical address and subsequently parsing of the host page table to find the corresponding host physical address (see Figure 2.2).

2.2.4 Platform Virtualization

Virtualizing the CPU and memory management falls short of virtualizing a whole machine; An operating system relies on the services of a number of tightly coupled devices that need to be implemented in the VMM because of performance considerations.

Such devices include an interrupt controller and a timer device. Both are of less complexity compared to peripheral devices and can therefore be implemented without increasing the VMM complexity significantly.

2.2.5 Peripheral Device Virtualization

Providing the VM with peripheral devices such as a keyboard, a mouse, network interface cards, hard disks and graphics cards is essential to the usability of the VM. In contrast to platform virtualization, which has to be implemented in software by the VMM, peripheral devices may be virtualized in several ways.

One solution would be to fully emulate an existing device. In order to do so, the VMM has to implement a full device model and to intervene on any device access. Such a solution allows guest operating systems to run unmodified by using their native drivers.

Such a solution increases VMM complexity, because it requires the full device model to be implemented. Another source of complexity comes from the means needed to safely emulate device memory: In x86, device memory may be mapped straight into the physical memory space of the machine. It is then accessed using regular memory operations. A write to device memory is usually unbuffered and interpreted by the device as a command, which is immediately acted upon. To provide the VM with the illusion of direct device memory access, the VMM has to recreate its behaviour. However, there is no way to trap on memory accesses other than page faults. Therefore, the VMM has to make sure that accesses to device memory regions in guest memory cause a page fault and thus enable the VMM to gain control. It may then emulate the instructions accessing the memory region, update the device models accordingly before returning to the VM. This requires the VMM to provide an instruction emulator, which comes with the risk of bad performance and increases the VMM complexity. In the literature this technique often called **full-** or **faithful virtualization**.

Device operations often consist of many commands that update the device state incrementally until the actual operation is executed. Because in faithful virtualization each IO operation has to be handled by the VMM, such behaviour causes significant numbers of world switches, thus causing substantial performance penalties.

A different approach is to present the VM with a custom device interface that does not resemble a physical device. Interfaces of physical devices are often inefficient, and require incremental device state updates for complex device commands, which cause numerous traps to the VMM and subsequent instruction emulation. Instead, a custom interface can make use of efficient shared memory communication, which increases the expressiveness of commands and thereby reduces the number of VMM interactions. The interface used by the guest to command the device can be implemented using calls to the VMM that resemble system calls used by applications to request services from the OS. Instead of requiring the implementation of a device model of an existing device,

an abstracted device model can be created, which, because it is not restricted to the limitations of the machine interface, may be of less complexity than a model of a physical device.

In summary, such a solution is of less complexity than faithful virtualization and may provide better performance [BDF+03]. However, providing the VM with an interface that is different from the physical machine breaks the equivalence criterion of classical virtualization and is therefore called **para-virtualization**. Para-virtualization requires custom drivers for the guest OS to make use of virtual devices.

Both faithful and para-virtualization require the VMM to implement back ends to the virtual devices. This can be done in several ways, for example by vesting the VMM with device drivers that directly access the machine and are multiplexed by the VMM, or by relying on services of the host operating system.

The third approach is to give direct device access to the VM. However, great care is needed to avoid security pitfalls: Direct memory access (DMA) allows devices to directly read and write from main memory without OS intervention, bypassing its isolation mechanisms such as paging. If the VM is allowed to directly drive devices using DMA, it must be counted to the TCB of the system. This huge increase of the TCB is not wanted for most scenarios. Hardware innovations that mitigate this problem are already in preparation: IO MMUs can be used to provide devices with a virtual view on main memory and thus restrict access [AMD07].

Another solution would be to use the VMM as a proxy for direct device access; The guest OS uses an adapted device driver that instead of directly writing to device memory issues a call to the VMM, which can then inspect and validate the command and issue it to the device memory on behalf of the VMM.

Devices such as network interface cards (NICs) even support multiplexing in hardware and can be used as-is (as long as they do not facilitate DMA).

2.3 Discussion - Are VMMs Microkernels Done Right?

Hand and colleagues [HWF+] initiated an interesting discussion by boldly claiming that virtual machine monitors –while striving for different targets– achieve many of the goals of a microkernel based system. Their arguments are that VMMs establish a more versatile interface than microkernels do and mitigate the need for fast IPC. The microkernel community was quick to respond and refute most of Hand's arguments [HUL06].

However I think that the discussion is worthwhile because despite comparing apples with oranges it sheds light on important similarities of both microkernels and VMMs: While microkernels surely represent the smallest possible kernels, VMMs may also be small in size compared to monolithic kernels. Both systems multiplex memory and CPU time for their clients. The difference lies in the interfaces they present; VMMs present the full machine interface, whereas microkernels provide an abstracted interface.

Virtualizing an operating system alone does not automatically provide stronger security; VMMs can only provide isolation as strong as separate physical machines do. However, it

is worth looking at lessons learned from microkernel systems research: With microkernels it is possible to build applications that have a tiny TCB.

Given the premise that commodity operating systems are inherently insecure and it is not possible to make them more secure without compromising application compatibility, how can we use the untrusted applications of a commodity operating system for tasks that demand strong security?

The solution may be a combination of a microkernel and a VMM in a split-application scenario [Här02]. The VM may communicate with secure applications on top of the microkernel through a well-defined secure channel.

An example of such an application is secure online banking. The user runs a commodity browser in a virtual machine. When the user is prompted to provide his private key to authenticate himself, the VM sends a message to a small application running on top of the microkernel. This secure application must show the information from the website and allow the user to input his private key, encrypt it and send it back to the browser in the VM. The browser than sends the encrypted data to the bank and the user is authorized. Thus, the untrusted software stack in the VM has no chance to compromise the authentication process for example by sniffing the private key.

2.4 Summary

In this chapter I motivated why secure computing is of importance and introduced microkernels as basis for secure systems. I then introduced virtualization and its building blocks. This was completed by a discussion about the similarities and differences of virtual machine monitors and microkernels, and why it may be a good idea to combine both.

3 Related Work

This chapter gives an overview on projects related to this thesis. Presented systems include the microkernels Fiasco and L4Ka::Pistachio. Additionally I will give a brief introduction to the capability-based EROS operating system to show that a secure and robust operating system can be built using object-capabilities. I will also present the rehosted operating systems L4Linux, Wombat/OK:Linux and User-Mode Linux as well as important examples of VMMs.

3.1 Microkernels

In this chapter I will introduce two second generation microkernels. Beforehand, I will shortly revisit the major design features that constitute second generation microkernels.

When we refer to second generation microkernels, we usually mean microkernels of the L4 family. The L4 specification was devised by Jochen Liedtke and includes three basic abstractions: Address spaces are represented by **tasks** and used as isolation domains; Activity inside of tasks is abstracted as **threads**; Communication between threads is done with synchronous **inter process communication** (IPC).

Address spaces can be constructed with three operations [Lie95]:

Grant One task may grant a page to another task if the recipient agrees. The granted page is removed from the granter's address space and included into the grantees address space.

Map A task may map a page to another task on agreement. Thereafter, the mapped page is visible in both the mapper's and the mappee's address space.

Unmap A task may unmap a page without consent of the mappee. The page will be recursively unmapped from all address spaces that contain mapping of the page.

With these operations address spaces can be constructed recursively by user land servers. In an L4 system a special task, sigma0, initially owns all physical memory. Upon that a hierarchy of pagers can be built.

In L4 IPC is the basic mechanism for communication between threads. IPC is unbuffered and synchronous: Only if both sender and receiver agree and are ready, the kernel does a rendezvous of both and delivers the message. This modus operandi proved to perform well [Lie94]. Upon this mechanism other techniques such as remote procedure calls may be implemented.

3.1.1 Fiasco

The Fiasco microkernel is a project at TU-Dresden that arose from an effort to create a real-time capable microkernel. It is written in the high level programming language C++, and runs on x86, x86/64, ARM and PowerPC platforms. A Linux user land port is also available for development purposes. Support for symmetric multi-processing (SMP) is available, and the kernel has been shown to offer good performance with a port of Linux [HHL+97]. Its sources are provided under the terms of the GPL to encourage community participation. Originally, it implemented the L4 specification and has since been used as a research vehicle to explore and evaluate evolutions in the field of microkernel research. It includes refined support for standard monolithic operation systems and is platform for a ported version of Linux. Fiasco is co-developed with its run-time environment.

Fiasco sports support for secure virtual machines, which is another embodiment of protection domains alongside tasks. Contrary to tasks, its interface does not consist of system calls and virtual memory, but of a virtual CPU and virtualized memory including a virtual MMU. VMs are subjects to the same address space construction rules as tasks.

Recently support for object-capabilities was added to the kernel. On top of it, a capability based run time environment was developed that follows the principle of least authority and aids developers in building applications. This run time is named L4 Runtime Environment (L4Re).

3.1.2 L4Ka::Pistachio

L4Ka::Pistachio is a microkernel developed by the System Architecture Group at the University of Karlsruhe in collaboration with the DiSy group at the University of New South Wales, Australia. It is written in C++ and runs on x86, x86-64 as well as PowerPC machines. The microkernel provides SMP support and fast local IPC [pis].

3.1.3 EROS

The Extremely Reliable Operating System, or EROS for short, is an attempt to build a system that uses capability-based privilege management for any operation without exception. It has a small kernel and uses a single level storage system for persistence, which is transparent to applications. The EROS system is a clean-room reimplementation of its commercial predecessor KeyKOS written in the high-level language C++ for the x86 platform. Address spaces are used for isolation and object-capabilities for access control. Early microbenchmarks show performance comparable with Linux.

EROS and its predecessors showed that efficient and secure capability based systems can be constructed.

3.2 Rehosted Operating Systems

Implementing a rich user land on top of a microkernel is a challenge of its own as can be observed with the Gnu Hurd project [WB07].

A pragmatic solution is the port of a monolithic kernel to run as a user land server that executes unmodified applications. Such a port is called **rehosting**. The rehosted kernel has to be modified to run as an application on the host OS. That means, it has to be ported from the machine interface, where it is in full control, to the application binary interface (ABI) of the host operating system. To run unmodified applications, the rehosted OS must behave like the original with respect to user land. For example applications make assumptions about the layout of their address space and would not run if it is altered. Similarly they rely on the correct behaviour of synchronisation mechanisms. Therefore a rehosted OS has to be in control of its applications. In contrast to running on a real machine the rehosted OS must employ mechanisms of the host; For example the rehosted OS must not manipulate hardware page tables directly, but has to employ host mechanisms to create the address spaces of its applications.

During the next paragraphs I will introduce three important examples of rehosted operating systems: L4Linux, Wombat/OK::Linux and User-Mode Linux.

3.2.1 L4Linux

L4Linux is a rehosted monolithic kernel. Its basis is the Linux kernel and it runs as a user land server on the Fiasco microkernel.

L4Linux executes in its own address space, and runs its applications in separate L4 tasks. L4Linux applications can be forced to give control to the L4Linux server with a system call that has been enhanced for this specific purpose [Lac04]. Similarly the Fiasco microkernel was enhanced in a way that it supports special threads that are not allowed to do system calls directly, but whose faults, exceptions and system calls are reported to their corresponding pager. L4Linux uses these special threads to run its applications and remain in their control. To administer client address spaces, L4Linux uses L4 map and unmap operations, which increases the costs for memory management.

Native Linux maps itself in the upper gigabyte of its applications address spaces. An entry into the Linux kernel, for example a system call, requires a privilege level switch only. In L4Linux, the upper gigabyte of the application's address space is occupied by the Fiasco microkernel. When an application wants to voluntarily enter the L4Linux kernel, it executes an INT 0x80 (like in native Linux), which is intercepted by Fiasco. The microkernel synthesizes a message on behalf of the application and switches to the L4Linux address space to deliver the message. L4Linux receives the message and can now run the operation as requested by the application. In summary, entering the kernel from an application requires two privilege level and one address space switch in L4Linux in contrast to one privilege level switch in native Linux. This is one example where the L4Linux setup incurs overhead. L4Linux is a highly optimized rehosting effort, and despite inherent performance disadvantages performs well in many workloads.

L4Linux runs encapsulated and is binary compatible to Linux. An illustration of L4Linux is given in Figure 3.1. L4Linux runs Linux applications side by side with L4 applications with a small TCB.

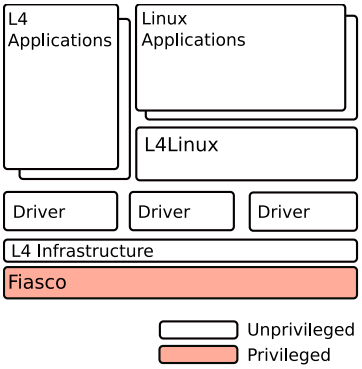

Figure 3.1: L4Linux running as a user land server on top of the L4 microkernel Fiasco.

3.2.2 OK:Linux aka Wombat

Wombat is a port of Linux to the L4Ka::Pistaccio microkernel. The basic architecture is similar to L4Linux, with the Wombat kernel residing in its own address space. The main emphasis on this port was to efficiently support Linux on L4 on embedded platforms. The work was done at NICTA [LvSH05].

The Wombat project is currently stalled, but its intellectual properties live on in the Open Kernel Lab's OK:Linux project, which runs on top of the OKL4 microkernel. The currently available version of OK:Linux is based on Linux 2.6.24.9. Unfortunately no further information on the project is available.

3.2.3 User-Mode Linux

User-mode Linux (UML) is a port of the Linux kernel to the Linux API. The UML binary is mapped into the address spaces of all its applications. UML remains in control by using the ptrace system call trace facility to keep track of system calls as well as signals to remain in control of faults and exceptions [Dik00]. Being a full fledged Linux port, it runs the same binary applications as the host [Dik01].

For any UML application a tracing thread is employed that intercepts system calls. This mechanism allows only interception but not canceling of system calls. Therefore the tracing thread nullifies the original system call by issuing the getpid() system call, which results in an unnecessary kernel entry. UML uses signals to force control to the UML kernel during a system call or interrupt. Signal delivery and return are slow and impose a noticeable performance deterioration [ska].

To protect the UML binary from its applications a *jail* mode was introduced that remaps the UML binary in a read only fashion while executing an UML application, and remapping it with write permission on context switches. This modus operandi slows down UML context switches considerably and does not protect the UML binary from being read, which can be used by attackers to tell whether or not this attacked system is running UML.

To improve both UML performance and security a patch was developed to enhance Linux support for UML. With this patch UML can run in a separate address space and is therefore inaccessible to its applications, which solves the security problems. Additionally the patch obviates signal delivery on system calls and thus improves performance considerably. The patch is called SKAS, which is short for Single Kernel Address Space [ska].

Another patch further improves performance by enhancing the ptrace mechanism with a new command that allows system calls to be canceled. This patch is called User-Mode Linux Sysemu [sys].

Both patches have not yet found their way into the main Linux source tree and are currently outdated.

UML uses virtual devices which are constructed from software resources provided by the host [Dik01]; UML may use any host file as a block device and attach to any host network interface.

UML was created to show that the Linux system call interface is sufficient to host itself. It is now used as a tool for operating system developers, who want to make use of general purpose debuggers. It can also be used for sandboxing and jailing hostile code as well as untrusted services. Another application is consolidation of physical machines. Ports of UML to other host operating systems may be used to provide the Linux interface on such systems.

A running instance of UML is illustrated in Figure 3.2.

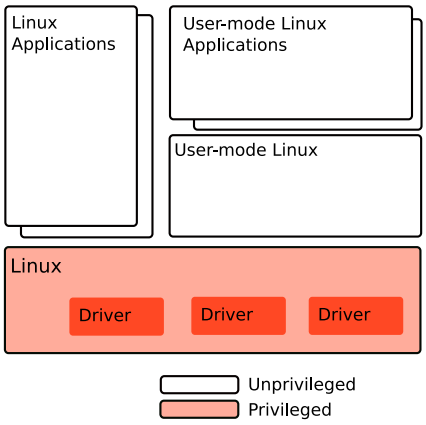

Figure 3.2: User-mode Linux running on top of Linux.

3.2.4 Pre-virtualization

Pre-virtualization is a form of automated OS rehosting that attempts to achieve the performance of paravirtualization with little porting expenditure. Pre-virtualization allows an adapted operating system binary to be run on bare hardware as well as on a number of different hypervisors such as L4 and Xen.

To achieve good performance the guest operating system kernel is modified. Modifications are largely automatic, and include the rewrite of assembly code to pad sensitive instructions, the analysis of high level source code to find sensitive memory accesses, as well as structural modifications such as the introduction of function-call overloads and an empty region within the guests virtual address space. A hypervisor can decide whether or not to modify the guest kernel to make it benefit from para-virtualization. Possible run time modifications include the injection of VMM code and the replacement of sensitive instructions as well as replacement of kernel functions. The injected VMM code can, for example, implement mode switches of the virtual CPU, batch page table updates, and issue efficient calls to the hypervisor where appropriate.

A prototype implementation supports L4 kernels as well as Xen as back end, and supports the Linux operating system as guest. This implementation already showed good performance, with only 5.5% performance degradation with the Xen back end and 14.6% with the L4 back end, both compared to native Linux [LUY+08].

3.2.5 Summary

Rehosting an operating system requires substantial work, both in adapting the OS to the host OS's interface and in implementing optimizations in the host OS to aid in rehosting.

With lots of optimization applied, rehosted operating system achieve good performance. However, achievable performance is limited because of the overhead of mapping operations to host mechanisms.

3.3 Virtual Machine Monitors

3.3.1 Xen

Xen is a virtual machine monitor for the x86 architecture. It provides an idealized virtual machine abstraction to which operating systems such as Linux, BSD and Windows XP can be ported with minimal effort [BDF+03]. Both systems with and without hardware virtualization extensions are supported.

Guest operating systems must be ported to run on top of Xen. The required modifications are reasonably small for Linux: Only about 3000 lines of code had to be added, which is about 1.36% of the total x86 code base. Xen provides good performance, with little virtualization overhead of only a few percent [BDF+03].

Xen is designed to be in full command of the machine, with virtual machines running in less privileged mode. The Xen kernel is a combination of a VMM and a hypervisor. It is usually referred to a **hypervisor**.

The idealized virtual machine abstraction implemented by Xen has a number of restrictions compared to the full machine interface: A guest cannot install fully-privileged segment descriptors as segmentation is used to protect Xen from guest operating systems and segments must not overlap with the top end of the linear address space. Updates to page tables are batched and validated by the VMM. The guest OS must run at a lower privilege level than Xen. Usually the guest kernel runs on Ring 1, with guest applications

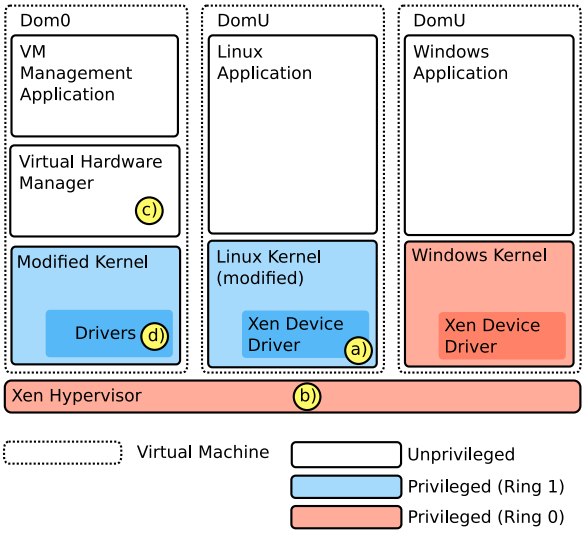

Figure 3.3: The Xen hypervisor running one VM to drive the hardware (Dom0) and two VMs with no direct hardware access (DomU). Windows is run using hardware virtualization, whereas Linux is run paravirtualized.

in Ring 3 (ring compression), using the host MMU to provide isolation between guest applications.

With the advent of hardware virtualization technology, Xen was enhanced to make use of it. Therefore, Xen can now also run unmodified OSes.

A Xen setup includes one para-virtualized VM that runs the devices of the host machine and is called Dom0. It also runs a management application, which can be used by the administrator to manage VMs.

All other VM instances –called DomU– have no direct hardware access, but communicate with Dom0 for multiplexed device access. Devices for all VMs besides Dom0 are implemented with a custom interface; Data is transferred using asynchronous IO rings, and interrupts are delivered using an event mechanism. Because Dom0 has full device access, it belongs to the trusted computing base of all VMs [MMH08].

A typical Xen setup is depicted in Figure 3.3.1. Xen runs in Ring 0, with a para-virtualized guest driving the host devices (Dom0). It also runs a VM with a para-virtualized Linux, and a VM with an unmodified Windows (using hardware virtualization). Both DomU VMs do not access physical devices. A device access of the DomU running Linux is initiated by the Xen device driver (a). The Xen device driver communicates using an efficient protocol (b) with a management application in Dom0 (c) which multiplexes the device for all DomUs. The management application then uses mechanisms of the guest OS in Dom0 to access the physical device (d).

If Dom0 drives devices with DMA memory accesses, both the Xen kernel and the OS running as Dom0 belong to the TCB.

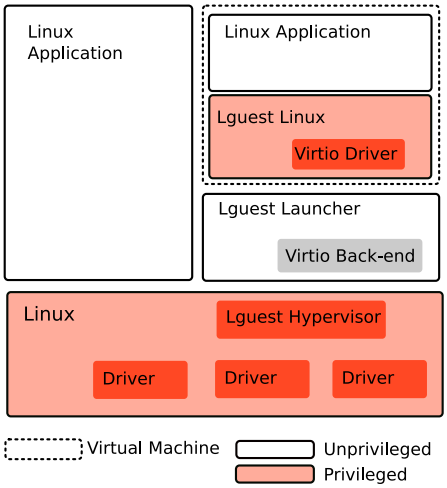

Figure 3.4: Linux running a para-virtualized Linux in a VM using the Lguest hypervisor.

3.3.2 Lguest

Lguest is an extension to Linux that allows it to act as a hypervisor by leveraging hardware virtualization extensions. Contrary to KVM it does not support faithful virtualization, but runs para-virtualized Linux kernels exclusively. For device virtualization it resorts to the virtio framework. It is meant to be of minimal complexity and comprises only 5000 lines of code, which makes it easy to modify and extend. The hypervisor functionality is implemented with a kernel module, and as such can be loaded at runtime. A tiny launcher application initiates new VMs and implements the virtio interface for para-virtualized device access[Rus]. The VM is controlled by the Lguest kernel module. A setup of lguest is depicted in Figure 3.4.

3.3.3 NOVA OS Virtualization Architecture

NOVA is an attempt to create a secure virtualization environment with a small TCB from scratch. It consists of *NOVA*, a novel microkernel with hypervisor functionality (a **microhypervisor**), which aims to be small and efficient. NOVA supports capability-based access control, and has a minimal interface for hardware virtualization. On top of NOVA runs a multi-server user environment. Aside from drivers, it also runs the VMM *Vancouver*, which does faithful virtualization. Vancouver runs entirely unprivileged. Each VM is coordinated by its own instance of Vancouver [Ste]. The NOVA setup is illustrated in Figure 3.5.

3.3.4 KVM

The Kernel-based Virtual Machine (KVM) is a virtualization extension to Linux that enables it to act as a hypervisor. It strives to implement faithful virtualization and uses a

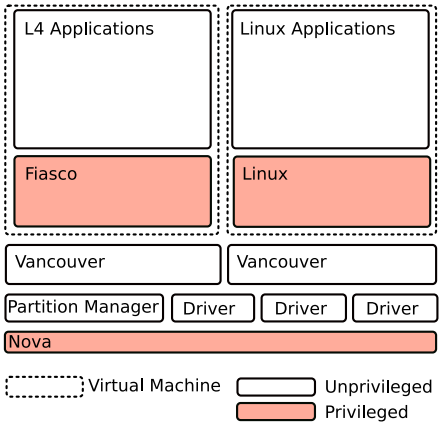

Figure 3.5: NOVA OS Virtualization Architecture, running a VM with Linux and one with Fiasco.

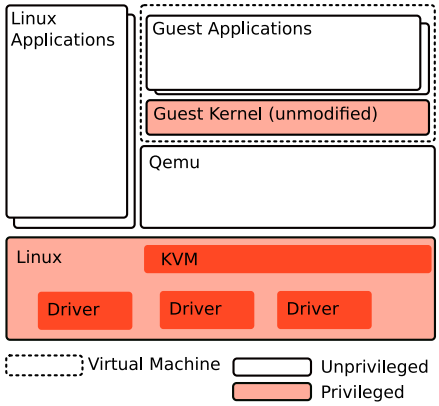

Figure 3.6: KVM running a VM side by side with normal Linux applications.

slightly modified instance of Qemu for emulation. KVM requires hardware virtualization extensions and supports nested paging.

In a KVM setup the full monolithic Linux kernel runs in privileged mode and –in terms of security– counts to the TCB of all applications and VMs running on top of it.

A KVM setup is illustrated in Figure 3.6. It runs one VM with the help of Qemu side by side with Linux applications.

3.3.5 KVM-L4

Peter and collegues ported KVM to run on an L4 system. The KVM kernel module was ported to run in L4Linux. Its communication with the hardware was modified to make use of Fiasco's support for VMs. The microkernel is the only privileged component and enforces isolation between VMs and applications, KVM-L4 does not increase the

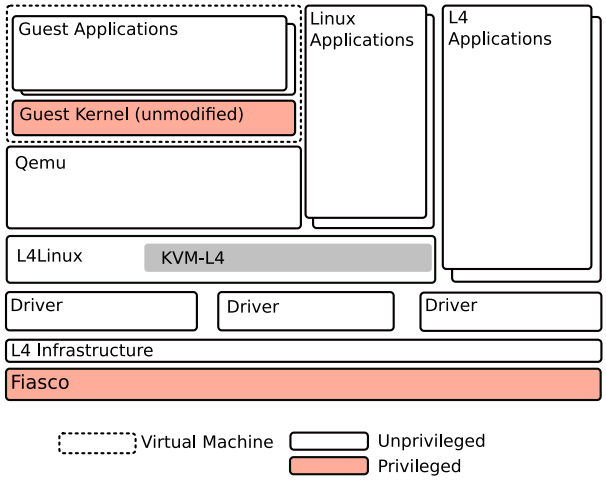

Figure 3.7: KVM-L4 running one VM and L4 applications with a small trusted computing base side-by-side.

system's TCB. Therefore this setup allows applications with a tiny TCB to run side by side with virtual machines. This setup is a huge improvement compared to native KVM in terms of security. KVM-L4 does not degrade performance compared with KVM [PSLW09].

In Figure 3.7 you can see a setup with KVM-L4 running a VM on top of the Fiasco microkernel.

3.3.6 Hardware Assisted Virtualization for the L4 Microkernel

In his Diploma thesis, Sebastian Biemüller introduced hardware assisted virtualization on top of an L4 microkernel [Bie06]. He proposed an extension to the L4 API to support efficient virtualization. For this purpose the VMM was split into two components: A hypervisor that runs privileged and the monitor that runs in user land. The prototype implementation used Intel's VT-x virtualization extension and implements a software memory management unit.

The prototype implementation of this VMM, as presented in the thesis, does not have multiprocessor support for virtual machines and supports only an L4 microkernel as guest OS. It implements an interrupt controller, a timer device and a serial console. Similar to KVM-L4, this work does not significantly increase the TCB of applications running side-by-side. Conversely, it does not have a capability based system as substrate. Unfortunately no evaluation information about this work has been published.

3.3.7 VMware Workstation

VMware Workstation is a hosted virtual machine architecture. It runs on commodity operating systems such as Windows and Linux and supports virtualization both with and without hardware extensions. VMware Workstation installs a driver into the host

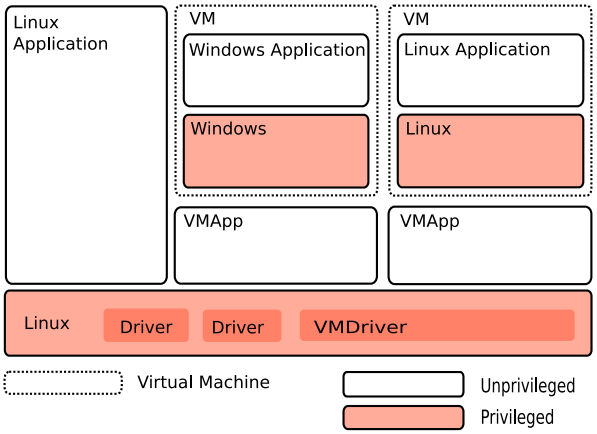

Figure 3.8: VMware Workstation running two virtual machines on a Linux host.

operating system that implements the privileged part of the virtual machine monitor (VMDriver). A second component runs as an application and is used to translate IO operations of the VM to host primitives (VMApp) [SVL01].

VMware Workstation has a fixed set of virtual devices that may be used by the guest. Host drivers are used as a back end. A set of custom drivers can be installed into guest operating systems to make it aware of virtualization and use more efficient IO interfaces.

An example VMware Workstation running on a Linux host is illustrated in Figure 3.8. In this example it drives two virtual machines. In a VMware setup, both the hosting monolithic kernel and the VMDriver component belong to the TCB.

4 Design

Innovations in operating systems often imply breaking backward compatibility. However, the availability of applications is a crucial prerequisite for the adoption of an OS. As the development of complex applications is usually a slow and expensive process, providing a stable interface is an important aspect in the development of new operating systems.

Linux is a free, open source operating system kernel. It is used in many applications in the embedded, desktop and even server domain. During the years many companies started to use Linux and add to its already large application basis with their software. Because of this variety of available software I chose Linux as the target interface of my work.

On top of Fiasco, Linux applications are supported with two projects; L4Linux and KVM-L4. L4Linux (see Chapter 3.2.1) is a version of Linux that has been rehosted to run directly on the L4 API. The rehosted Linux kernel has inherent performance disadvantages compared to native Linux: As illustrated in Figure 4.1, a kernel entry in native Linux (a) requires only one privilege level switch. In L4Linux (b) the same system call needs to be mediated by the microkernel, requiring two privilege level and one address space switch.

Virtualization extensions recently added to mainstream processors simplify the implementation of virtual processors. When the guest runs in a VM, the microkernel is not involved with the guest's operations such as system calls, as shown in Figure 4.1 (c). Therefore, the performance disadvantages of rehosted kernels related to processor virtualization do not apply to OSes running in VMs.

Virtualization extensions are used on top of L4 with KVM-L4. Its VMs already yield good performance. However, KVM-L4 relies on the services of L4Linux and Qemu, and thus has a substantial resource footprint.

In this thesis I want to make Linux applications usable on top of a microkernel. They should run with as much performance as possible. The solution shall require less resources than KVM-L4.

For its security properties, such as a tiny TCB and strong isolation between user land components, a microkernel is an ideal basis for secure systems. In this thesis, I will present a solution that uses a VM to run a Linux kernel on a microkernel.

I will proceed by defining the requirements of this work. Thereafter, I will present the overall architecture. The architecture will be refined in Section 4.3 with a discussion about the device interface. Section 4.4 will introduce the different attack scenarios that I consider. The design of the VMM will be introduced in Section 4.5.

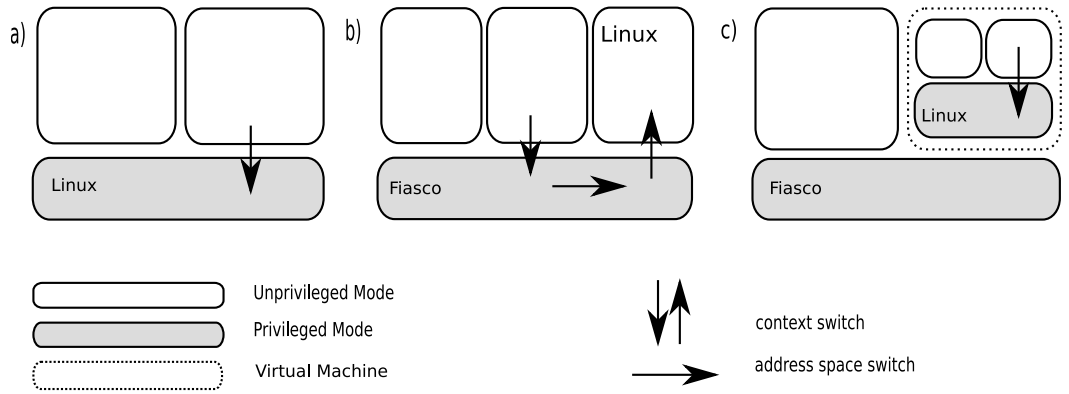

Figure 4.1: System call execution in (a) native Linux, (b) L4Linux and (c) Linux in a VM.

4.1 Requirements

My work has to fulfill the following requirements:

Functionality The whole range of Linux software shall be supported. Therefore Linux applications must not need to be adapted. Interaction through standard interfaces such as graphical use interfaces, mouse and keyboard as well as networking shall be available.

Isolation Linux applications must run encapsulated and not interfere with L4 applications. Security properties of the underlying microkernel system, as described in Section 2.1.1, must still hold.

Resource Footprint The amount of resources that is needed to run the VMM itself should be small.

Performance Application performance should match their native performance as closely as possible.

Complexity The solution shall be of low complexity. This keeps the costs for maintenance low and helps future researchers to adapt it for their own projects. It can improve the security because small systems are suited to a more thorough revision process.

4.2 Architecture

As described in Section 2.2, virtualization requires core, platform and device virtualization. Core virtualization is concerned with CPU and memory virtualization, both of which are done by hardware extensions.

Following Jochen Liedtke's microkernel rationale, all uncritical functionality has to be implemented in user land (see Section 2.1.1). Core virtualization is concerned with

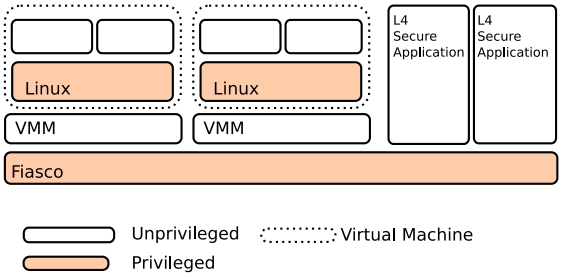

Figure 4.2: Setup with two VMs.

security critical operations: The *vmmcall* instruction to switch the CPU to guest mode is a privileged instruction and thus can only be executed by the microkernel.

The microkernel must retain control of the machine at all times; More specifically, it has to receive and handle all system interrupts. For example, if the guest were able to consume the system timer interrupt, the host would not be able to schedule and thus loose control over the machine. That is why the microkernel must enforce that host interrupts occurring during VM execution are intercepted and delivered to the host.

Similarly, the VM's memory must be constructed using the microkernel's secure mechanisms, which ensure that a VM can only access memory that has been allocated for its operation. In a high-level view, VMs are protection domains similar to tasks. Therefore, the same memory construction mechanisms that are used for tasks also apply to VMs. VMs are allowed to communicate with their VMMs only.

Peter and colleagues [PSLW09] were able to implement secure VMs by adding only 500 lines of code to the Fiasco microkernel, which is an insignificant increase of the TCB.

As described in Section 2.2.5, object-capabilities are a mechanism that allows systems that adhere to the principle of least authority. With support for object-capability based access control, a VMM can be built in user land that has the minimal set of permissions. With both object-capability and secure hardware virtualization support Fiasco is a viable choice for this thesis.

Coordination of the VM execution, platform and device virtualization are up to a VMM that is implemented in a task. The VMM runs as an ordinary task and is subject to the strong security properties of the microkernel.

For added security, different critical Linux applications can be run in separate VMs. It is therefore desirable that VMs are isolated from each other. In a scenario where the VMM is shared among several VMs, a malicious VM can attack the VMM and thereby render all its VM siblings compromised. This scenario can be avoided if a VMM drives one VM only. Consequently, I will employ one VMM per VM as illustrated in Figure 4.2, which shows a setup with two VMs.

Figure 4.3 gives an impression of the setup of Linux (a), L4Linux (b), KVM-L4 (c) and the envisioned architecture (d). Linux runs in privileged mode, with its applications running unprivileged. L4Linux and its applications run in user land on top of the microkernel. KVM-L4 uses L4Linux and Qemu to do faithful virtualization. The envisioned architecture is similar to KVM-L4 in that it employs a VM. Contrary to

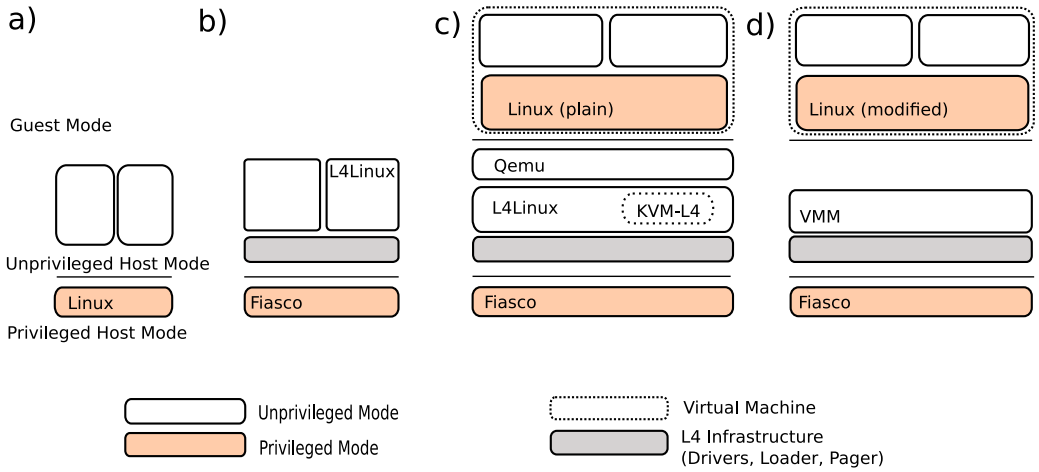

Figure 4.3: Architectural overview of (a) Linux, (b) L4Linux, (c) KVM-L4 and (d) the envisioned system

KVM-L4, it depends neither on L4Linux nor Qemu, which promises a much reduced resource footprint.

4.3 Para-Virtualization

The VMM has to provide the VM with (virtual) devices (platform and device virtualization). One option is to provide the VM with virtual duplicates of physical devices. In general, this requires both an implementation of the device model, and an instruction emulator (as described in Chapter 2.2.5). Another option is to provide a custom device interface. Such an interface may use efficient shared memory communication and obviate an instruction emulator. A custom device interface requires custom drivers for the guest.

KVM-L4 does faithful virtualization, and therefore provides a guest with an exact duplicate of a physical machine, including a complement of devices. As described in Chapter 2.1.2, faithful virtualization requires instruction emulation and device models. Both are implemented in Qemu, which runs as a separate application. KVM-L4 achieves about 92 percent native performance [PSLW09], which is on par with native KVM.

KVM and Qemu are widely used, and actively developed by a number of skilled people all over the world. Therefore I assume that the emulator has received careful tuning. The remaining overhead of about 8 percent results from inherent costs for device virtualization: IO intercepts, instruction emulation and update of device states. Many workloads cause frequent device interaction, and are therefore limited by the performance of device virtualization.

Implementing device virtualization inside the VMM is one possible optimization that spares the overhead incurred with switching to an external component, but the costly IO intercepts and performance penalties caused by emulation cannot be avoided.

Therefore, to achieve better performance than KVM-L4, I decided to go without full hardware backward compatibility. Instead, I will provide the VM with custom device interfaces, thereby avoiding emulation altogether. The device interface will use calls to the VMM, using the *vmmcall* instruction (**hypercalls**). Hypercalls will facilitate shared memory communication to increase the expressiveness of individual commands and thus significantly reduce their number.

16bit real mode is a relic from the early days of x86. It is used by all current OSes for their early set up only. During operation, protected mode (32bit) is used exclusively. Whereas AMD SVM supports hardware assisted virtualization of real mode code, only the most recent Intel processors with Intel-VT have that capability. With the prospect of future Intel-VT support, I decided to not support real mode code. As with 16 bit mode, the BIOS is a remnant of former times. Its only purpose is to provide a backward compatible environment during the boot up stage. With VMs starting directly in 32bit mode, the BIOS is rendered unnecessary as well.

A number of OSes cannot be modified, either because they are distributed without sources, or their license does not allow modification. These OSes cannot be adapted to the target VM and therefore cannot run para-virtualized. The most prominent example is Windows, which runs a huge variety of commercial applications. In a second step I facilitate Linux functionality to establish virtual machines that do faithful virtualization. I will later revisit fully backward compatible VMs in Section 4.6.

4.4 Security Considerations

In this section I will shed light on the security implications of VMs on microkernel-based systems. There are a number of attack scenarios that are of no concern in this thesis: Attacks on the Linux kernel originating from malicious Linux applications or from the network may target guests in the VM as well. Although virtualization might be used to increase security, for example with analysis of the VM's behaviour, these scenarios are out of scope of this work.

A VM running on a microkernel-based system poses a potential threat both to the microkernel and to its applications. For this thesis I assume that the microkernel itself is resistant to attacks. This assumption draws on the microkernel's properties as described in Section 2.1.1. Especially for VMs, I assume that the mechanism used to construct the VM's protection domain is valid and does not allow the VM to access memory without permission. The kernel ensures that it receives physical interrupts and thus remains in control at all times.

I also assume that the microkernel mechanisms work to protect tasks from each other: The secure memory construction mechanism ensures that tasks can only access memory to which they were granted permission, and the capability system is correct and does not leak access rights. Furthermore, I assume that the peers used for services are well behaved, and safeguard the interests of all their clients.

Another class of attacks involve the VMM: Because the VM can only communicate with the VMM, attacks originating from the VM are limited to the VMM. One scenario is that a guest application might attack the VMM and thereby compromise its Linux

kernel. Under the assumption that the world switch path is immune to attacks, the only attack vector is the hypercall interface of the VMM. Configuring the VM such that the *vmmcall* instruction is privileged and thus hypercalls are allowed to the the guest kernel exclusively, this class of attacks can be avoided. Another interaction between the VMM and guest applications are preemptions. However, these preemptions are transparent to the VM and cannot be exploited.

The guest kernel may attack the VMM though the hypercall interface. However, even if it succeeds, the security situation of the remainder of the system is unchanged. That is because the VMM is encapsulated and assumed to act on behalf of the VM. Because the VMM exercises control over the attacker VM only, the only privilege that an adversary could gain is access to its own VM. As the attacker runs in privileged mode in the VM, it holds full control over the VM regardless whether it controls the VMM or not.

Another source of attacks are external attackers. The input to the VM is mediated by device managers. It is up the device manager to do protocol multiplexing and to fend of attacks on the protocol.

4.5 Virtual Machine Monitor

The VMM coordinates the VM execution and provides it with resources such as virtual devices and memory. It runs encapsulated in an L4 task and manages resources for the VM. Also, it includes an implementation of platform devices such as an interrupt controller and provides the VM with access to peripheral devices.

The following sections will introduce the design of these components.

4.5.1 Core Virtualization

CPU and memory virtualization are handled with hardware virtualization extensions, which are under control of the microkernel. The VMM uses the microkernel's interface to command its VM.

With para-virtualization, the guest can voluntarily leave the VM execution with the *vmmcall* instruction. This mechanism is used to implement calls to the VMM that work similar to system calls: Register contents are used to select commands and to pass values to the VMM. Additional information can be provided in shared memory. As stated before, these calls are called **hypercalls**.

Asynchronous communication from the system to the guest is done with virtual interrupts. Once injected, the guest's execution is preempted and its interrupt handler is called according to the guest's interrupt descriptor table. From that point on the regular interrupt handling procedures apply.

The VMM's **control loop** coordinates the VM's execution; It instructs the microkernel to switch to the VM with a system call. Upon return, it checks the VM state to decide on the next action, for example to react on a hypercall. The control loop also injects virtual interrupts. It must not block as that would halt the VM execution.

Upon start up, the VMM allocates a memory region of appropriate size and access permission, and maps it into the virtual machine to be used as guest physical memory.

During the whole VM lifetime, the VMM has access to VM (physical) memory, which can be used for efficient shared memory communication between the VM and the VMM.

An alternative to allocating all memory at once is demand paging. Whenever the VM accesses a page that has not been used before, the VMM receives a page fault and maps a fresh page into the VM.

In this thesis I chose the former approach because it allows the mapped memory region to be contiguous in physical memory. Whereas this allocation scheme is more wasteful on system memory, it simplifies the address translation if the guest is allowed to directly drive DMA enabled devices.

4.5.2 Platform Virtualization

As described in Chapter 2.2.4, the VMM has to implement platform devices such as an interrupt controller (IC) and a timer device.

For the design of the IC, it is reasonable to align it with the high level abstractions of Linux. If done so, the Linux operations correspond directly to IC ones. As for a virtual device, interrupts are flagged by helper threads in a shared data structure, and delivered to the VM by the VMM's control loop.

The timer thread periodically blocks with a timeout, upon which it flags the timer interrupt at the virtual IC.

4.5.3 Peripheral Devices

For a VM to be useful, it has to be provided with virtual devices. The L4 system includes device managers, servers that multiplex access to hardware. For example *Ankh* implements a virtual network switch, thereby multiplexing the physical network card. Similarly, we have a secure console that securely multiplexes the system's frame buffer and maintains a secure input indicator, which cannot be forged by clients. These device managers consist of a device driver and multiplexing logic. Client communication with device managers is done with IPC and shared memory. These services are a natural choice for the back ends of virtual devices.

The VMM makes use of to the device managers to present the VM with a custom interface. The guest OS needs special drivers to communicate with the virtual devices. Because these drivers implement an abstracted interface only, I will call them **stub drivers**. The VMM's task is limited to translating the stub driver's commands into IPC messages, which are forwarded to the device manager.

Asynchronous events, such as the availability of packets, are signaled using IPC or asynchronously using User-Irqs. For each device manager, the VMM employs a blocking helper thread to receive the notification and flag the event as interrupt at the virtual IC.

Virtio provides a para-virtualization interface, which is used by Lguest and by KVM (optionally). It implements device drivers for a network card, a serial console and a generic block driver [Rus08]. It is up to the VMM to implement back ends to the virtio device interfaces. By using the virtio interface, I could benefit from existing drivers, but would have to implement complex mappings to the L4 device manager's interfaces, which would significantly increase the complexity of the VMM.

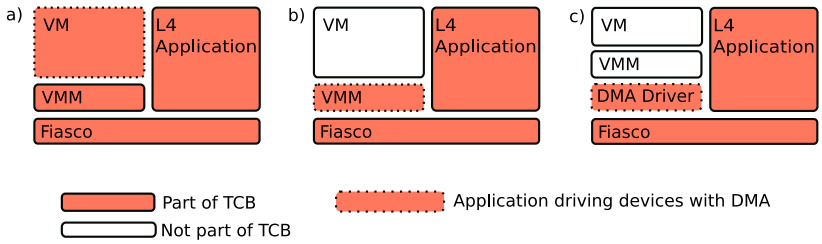

Figure 4.4: TCB of an L4 application where (a) the VM drives the hard disk, (b) hard disk commands are mediated by the VMM and (c) hard disk commands are handed out to an external server.

Instead, I opted to adapt existing L4Linux stub drivers. Because these already fit well to the device manager's interfaces, the translation to IPC should be much easier, which would help in keeping the VMM simple.

Unfortunately, not all devices can be shared using L4 device managers; For example, there is currently no device manager for hard disks available. For such devices one VM in the system may be allowed to directly operate on the physical hardware.

Secure device discovery is enabled by the *IO* server, which presents its clients with a virtual bus infrastructure. IO allows device access to be restricted to clients in a configurable fashion: It has an interface to map device memory into client address spaces and grants access to IO ports. In this thesis, I want to use IO to present the guest kernel with a virtual bus containing only the hard disk. The guest can then use its device discovery algorithms as usual. Depending on the intended use of the system, the VM may be allowed to access device memory directly.

However, allowing the VM to directly access devices that employ direct memory access (DMA) is problematic with respect to security: A malicious guest may use DMA to manipulate physical memory of the host and thus seize control of the machine. Consequently, this particular VM belongs to the TCB of all applications in the system. In Figure 4.4 the TCB of an L4 application is illustrated for this setup (a).

To mitigate this problem, the VM must not have access to device memory or IO ports. Instead, guest driver could be modified to issue hypercalls to the VMM. The VMM could then validate the command and make sure it accesses only memory regions that belong to the VM. After successful validation the VMM writes the command into device memory. As illustrated in Figure 4.4, this removes the VM from the TCB of an L4 application (b).

That still leaves the VMM in the TCB of the system, which might be prone to attacks from the VM. To further increase security, this checking functionality might be implemented in an even smaller external server, which further decreases the TCB (c).

As discussed in Chapter 2.2.5, IO MMUs allow DMA main memory accesses to be restricted in a secure fashion. Therefore, a VM running on a system using an IO MMU can drive DMA enabled hardware without having to be counted to the system's TCB.

4.5.4 System Environment

On x86 systems, OS kernels are usually loaded by a boot loader, which typically operates in real mode. It is installed in a predefined location and started directly after BIOS initialization. Its task is to prepare the machine for the OS (bootstrap) according a boot protocol. Upon receiving control, the OS queries the BIOS for system information, initializes itself and switches to protected mode, which is then used throughout system operation.

I decided not to support real mode code, which is no restriction for most standard OS apart from booting. However, with no support for real mode code, no standard boot loader can be used. Instead, it is the VMM's duty to bootstrap the guest. Guests are started directly in protected mode. BIOS calls are available in real mode only, which is not available in the VM. Instead, BIOS information such as the physical memory layout has to be provided by the VMM.

4.5.5 Multiprocessor Support

One of the goals of this thesis is to make multiple CPUs available to a VM (SMP). As described in Section 4.5.1, a virtual CPU is implemented with a thread running the control loop.

To run multiple CPUs, the VMM employs multiple control loops, each with its own thread. If control loops are mapped onto different physical CPUs, the VM can make use of real concurrent execution.

In SMP systems, CPUs communicate with one another using inter-processor interrupts (IPIs). Operating systems make use of IPIs to start processors, trigger remote function execution and to do synchronization. name=IPI,description=Inter-Processor Interrupt

For this thesis, I decided to equip each virtual CPU with an advanced programmable interrupt controller (APIC) back end, which contains facilities for IPI delivery.

4.6 Staged Virtualization

Many operating systems do not come with source code access and can therefore not be adapted to run directly in a VM established by the VMM. Microsoft Windows is the most prominent example. It is in wide use and gained a huge amount of available commercial software over the years. Another class of operating systems are legacy OSes that are no longer supported by its vendors but still run important legacy software.

It is desirable to make this huge body of software available. Faithful virtualization, the recreation of an existing physical machine, is a solution to this problem as it enables virtualization of unmodified OSes.

Peter and colleagues showed that KVM can be ported to run on top of the Fiasco microkernel [PSLW09], and that its performance is on par with native KVM. This KVM port requires an instance of L4Linux and Qemu.

I plan to do a similar setup that, instead of requiring L4Linux, uses a VM with a para-virtualized Linux. To describe this, I will use the following notation: The VM that runs the modified Linux instance will be called **first-stage VM**, whereas the VM

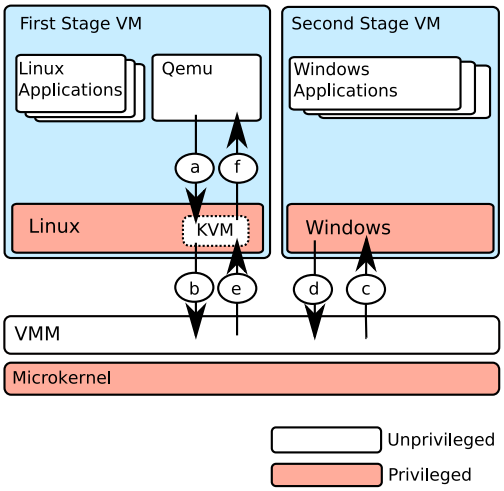

Figure 4.5: Staged Virtualization.

established by KVM, which does faithful virtualization, and runs unmodified OSes will be called **second-stage VM**.

To support staged virtualization, KVM needs to be modified. Instead of building guest physical memory using Linux mechanism, and running the *vmrun* instruction itself, it needs to use hypercalls. The VMM translates these calls into the appropriate L4 system calls.

Figure 4.5 gives an impression of this setup. The second-stage VM is run in the following way: VM execution starts in Qemu, which does its initialization. As soon as possible, it switches from emulation to execution using hardware virtualization. It therefore instructs KVM (a) to do a world switch. Contrary to native KVM, which does the world switch itself, in staged virtualization, KVM does a hypercall to instruct the VMM to do the world switch (b). The VMM then does the world switch on behalf of KVM using the microkernel interface (c). On intercepts, the second-stage VM is left (d), and the VMM resumes execution of the first-stage VM (e). Upon receiving control, KVM analyzes the exit reason of the second-stage VM, and switches back to Qemu (f) if needed.

4.7 Summary

The system architecture is illustrated in Figure 4.6. The VMM uses the microkernel's interface to control the VM execution (control loop) and memory. It uses L4 device managers as back ends for virtual devices. The guest's stub drivers communicate using hypercalls, which the VMM translates into IPC messages that are sent to the corresponding device managers. Asynchronous system events are received by blocking threads, flagged at the virtual IC, and eventually injected into the VM by the control loop.

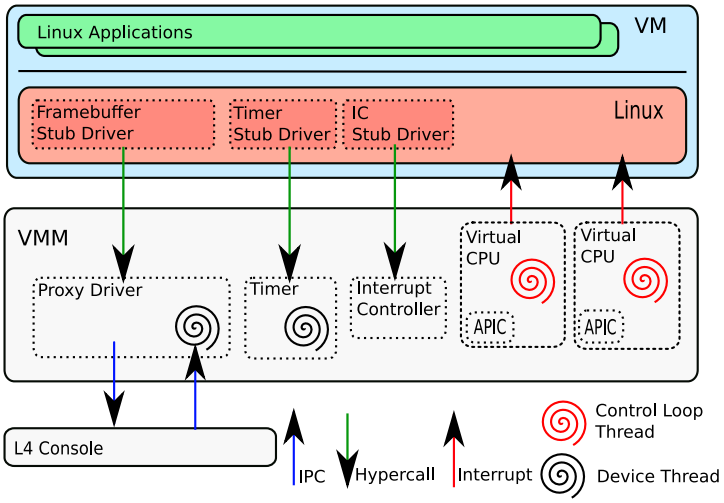

Figure 4.6: Envisioned system architecture.

In summary, the VMM contains the following components:

Boot Loader The guest is bootstrapped, provided with required system information, and finally started by setting the instruction pointer to the guests kernel entry. Instead of using a standard boot loader, these operations are done by the VMM.

Control Loop The VMM runs one control loop per virtual CPU. The control loops coordinate the execution of the VM, handle hypercalls, and inject virtual interrupts.

Platform Devices Platform devices, the global interrupt controller, APIC, and a timer device are implemented directly in the VMM.

Peripheral Devices Peripheral devices are accessed using a custom hypercall-based protocol that uses shared memory. The VMM maintains threads that receive external events and flag them at the IC. Guest stub driver's commands are forwarded to the corresponding device manager

5 Implementation

The implementation of the VMM started with the decision to use the C++ programming language. I did this in the hope that an object oriented design will both simplify the implementation and help to produce clean and easily extensible code. The resulting VMM, which will be called **Karma**, can serve as a starting point for further research.

In this chapter, I describe details of the VMM implementation. I will begin by explaining the core system, including core virtualization. Then, the system environment is presented, which includes platform virtualization, infrastructure needed to boot Linux, as well as SMP support. In the next section, I present how device virtualization is done. The chapter will be completed with a description of staged virtualization.

5.1 Core System

The Fiasco microkernel has support for the hardware virtualization extension SVM. In this chapter I will describe how the interface for this extension looks like and how it can be used to implement a VMM in unprivileged user land. First, I will describe the coordination of CPU virtualization and how the VM is provided with memory, looking at both the hardware and the Fiasco interface. Having provided this knowledge, I will explain the workings of the control loop.

5.1.1 Hardware Virtualization Interface

In x86 hardware-assisted virtualization the whole CPU state is duplicated for a VM. A world switch from host to guest mode is initiated with the *vmrun* instruction and left with an *vmexit* as described in Chapter 2.2.3.2. I will now give more details on hardware virtualization and how it is implemented in Fiasco.

In AMD SVM, an in-memory data structure, the *virtual machine control block* (VMCB), contains information about the virtual CPU: The virtual CPU's state is saved in the *state save area*. The state save area does not save general purpose registers, which need to be saved explicitly. Information that is used to control the behaviour of CPU in guest mode is saved in the *control area*.

The control area is used to control which instructions and events (e.g. host interrupts) are to be intercepted. It also contains information about the cause of the VM exit.

Memory virtualization is handled by nested paging, which works upon a regular host address space that is used as guest physical memory. A guest address is translated from guest virtual to guest physical using the guest's page table. Subsequently, it is translated from guest physical to host physical using a host page table. Both operations are done by the MMU and do not require VMM interaction.

5.1.2 Fiasco Hardware Virtualization Interface

The Fiasco hardware virtualization interface provides system calls to initiate a world switch and to create a host address space to be used as a host page table for nested paging.

A new *VM* kernel object was devised, which inherits the memory management interface of L4 tasks. Therefore, the address space for nested paging is created in the same way, like a task's address space : It is constructed by the VMM using map operations on the VM capability.

The VMM allocates a VMCB for the virtual CPU, and initializes it in a way that the guest can be bootstrapped. A world switch is initiated with the *l4_vm_run_svm* system call on the VM object capability. The VMM passes both the VMCB and the general purpose registers of the guest, which are not handled by the VMCB, along with the invocation. As described in Section 4.2, Fiasco enforces some of the values of the VMCB for security reasons.

The VM executes in the scheduling context of the thread that executed the *l4_vm_run_svm*. Therefore, it behaves like the calling thread with respect to scheduling. Host interrupts occurring during VM execution cause a VM exit, which are visible to the VMM as return from the *l4_vm_run_svm* system call.

5.1.3 Control Loop

I will now explain how the VMM's control loop works. In an abstract view, a control loop consists of a world switch and subsequent handling of VM exit conditions. Exit conditions can either be *asynchronous* or *synchronous*: Asynchronous exits occur for purposes that may be unrelated to the VM state, such as physical interrupts. Synchronous exits happen either to handle VM state transitions or voluntarily by the VM, for example for hypercalls.

In the most basic control loop two exit reasons need to be handled, that is asynchronous exits such as host interrupts and the basic synchronous exit reason of an error. Possible error conditions are for example invalid CPU states and guest initiated shutdown events

In the next refinement step, I implemented support for virtual interrupts: The back ends of virtual devices contain a blocking thread. Whenever a device manager has new input, it notifies this thread with an IPC message, or a User-Irq. The thread then registers the event at the virtual IC as a pending interrupt, a mechanism, which will be described in Section 5.2. On each VM exit, the control loop checks the IC for pending interrupts. If an interrupt is pending, it is injected into the VM. Interrupt injection will be described in more detail in Section 5.1.4.

Stub drivers issue hypercalls to send commands to driver back ends. Hypercalls are issued by providing required information in registers and shared memory, and voluntarily leaving the VM execution with the *vmmcall* instruction. The control loop handles hypercalls by analyzing the predefined designation register and forwarding the command to the back end component for processing. Driver back ends will be explained in detail in Section 5.3.1.

```
loop_forever
begin
    if(interrupt_pending())
        inject_interrupt();
    vmexit_reason = l4_vm_run_svm(VMCB, registers)
    switch(exit_reason)
    begin
        case Vmmcall:
            send_command_to_driver_back_end()
            break
        case Asynchronous_exit:
            break
        case Hlt:
            wait_until_interrupt_pending()
            break
        case Error:
            print_error_message()
            shutdown_vm()
            break
    end
end
```

Figure 5.1: Illustration of the Control Loop.

When an OS is idle, it executes the *Hlt* instruction, which, on a physical machine, suspends the processor until an interrupt occurs. In the VM, the Hlt instruction is handled by the control loop, which yields the processor for other activities that are ready to execute, until a virtual interrupt is flagged as pending by an event thread. The complete control loop is sketched in Figure 5.1.

At the moment, yielding the CPU is implemented with a timeout IPC of 1ms and subsequent polling, which is problematic with respect to interrupt latency. In this implementation interrupt latency can be as large as 1ms. I also implemented an alternative that uses asynchronous User-Irqs. In this implementation, the control loop blocks on a User-Irq that can be triggered by event threads on incoming events. Once the event thread has triggered the User-Irq, the control loop immediately continues execution. This implementation does not use polling, and in theory should keep the interrupt latency low. However, in my measurements the first implementation performed better. The reasons for this counter-intuitive behavior are unknown and subject to future research.

In summary, the control loop has to handle the following exit reasons:

1. Asynchronous Exits:

   Interrupt The VM was left because of a host interrupt. After the host OS returned control to the control loop pending interrupts are injected into the VM.

2. Synchronous Exits:

   Vmmcall The VM explicitly called the VMM. Such hypercalls are done by stub drivers to send commands to device managers.

Hlt When the guest executes a Hlt instruction, the CPU is yielded to other activities that are ready to execute.

Error An error condition occurred. Possible errors include invalid VM states and shutdown events.

5.1.4 Virtual Interrupts

The control loop can inject only one interrupt per *vmrun*. However, after a leaving VM execution, multiple event threads can receive notifications from their device managers and flag them as pending interrupts. Whereas the the control loop injects the first interrupt immediately, the next pending interrupt can only be injected as soon as the VM is left again. Thus, if the VM is left upon physical interrupts only, the interrupt latency for virtual interrupts can be rather large. Even worse, the VMM could receive at a faster rate, than it can inject them, which would result in outstanding interrupts that are never injected.

Therefore, depending on the number of pending interrupts, two different injection schemes are used: If one interrupt is pending, it is injected via the SVM virtual interrupt injection mechanism. If two or more interrupts are pending the first interrupt is injected as virtual interrupt, and virtual interrupt intercepts are enabled. The virtual interrupt intercept causes a VM exit at that exact point in time before the virtual interrupt is taken by the guest (e.g. when the guest enables interrupts). Upon virtual interrupt intercept, the VMM injects the first interrupt via the event injection mechanism of SVM and the second one as virtual interrupt. When only one interrupt is pending, it is injected as virtual interrupt and virtual interrupt intercepts are disabled. With this scheme we can make sure that the interrupt latency is short and all interrupts are injected.

5.2 System Environment

My implementation draws on work done by Torsten Frenzel, a PhD student at the chair for operating system research at TU-Dresden, who is working on a VMM that targets the ARM platform with the ARM TrustZone architecture [Lim]. At the time I started working on my thesis, he already had implementations for both a timer and an interrupt controller (IC). I used his code to jump start my VMM implementation, and modified it where necessary.

The IC implementation consists of a stub driver residing in Linux and a back end residing in the VMM. The back end consists of a shared data structure holding the interrupt state, and access functions both for the interrupt threads to trigger interrupts and for the control loop to poll for pending interrupts.

The timer back end resides in the VMM and contains a blocking thread that uses timeout IPC to get a notion of time. It periodically flags timer interrupts at the virtual IC, which are subsequently injected by the control loop.

Timer interrupts are the basis for scheduling and timeouts both in the host and the guest kernel. Because the VMM's timer interrupt is driven with timeouts of the host, the guest timer interrupt granularity is limited by that of the host. In this thesis, I configured Linux to use 100HZ timer granularity on a host Fiasco with 1KHZ granularity.

Instead of implementing a stub driver, I adapted the existing i8253 timer driver to communicate with the back end instead of the physical device.

5.2.1 Boot Loader

In this chapter I will first describe the boot process of a typical PC. Thereafter, I will describe how Linux boots in Karma.

After powering the system up, the processor starts execution from a fixed state; The processor starts in real-mode (16 bit) at the fixed address 0xffff0. At this address, it starts to execute the basic input-output system (BIOS), which is custom tailored to the machine. The BIOS does initial checks, finds and initializes installed devices. It then copies a block from the specified boot location (e.g. the first hard disk) into memory. This block is called the master boot record and contains the boot loader. After loading the master boot record the system starts the boot loader therein.

The boot loader sets up the system according to the boot protocol of the operating system. It then loads the operating system kernel and finally leaves the system to the operating system's boot code.

The operating system then further configures the system for its needs: It sets up the segment descriptor tables and the interrupt descriptor table. It also calls BIOS code for system information such as the layout of the systems physical memory. This system initialization is done in real mode to prepare the system execution in protected (32 bit) mode.

Once in protected mode, the OS sets up the initial page tables and turns on paging. Thereafter, it discovers devices and initializes them with the corresponding drivers. It then proceeds to start the first user process.

The initial boot process of PC hardware is (nearly) unchanged since the first PC that had an 8086 processor running 16bit code. This means a lot of the complexity of the boot process stems from these early systems and has the only function to prepare the system for protected mode.

For a VM we can remove this legacy code: When a VM is set up, the host system is already up and running. The VMM sets up the system, initializes the virtual devices, sets up the physical memory and configures the CPU for guest mode. This means the VMM can do both the tasks of the BIOS and the boot loader and start up the system in protected mode.

Doing so dispenses with the need for a BIOS emulation and thus reduces the complexity of the VMM. For Intel-VT systems, booting the system directly in protected mode also obviates an instruction emulator.

The Linux kernel supports booting from machines that employ the EFI [1] for machine initialization and booting. EFI boots systems in protected mode. It defines the layout of physical memory as well as the initial values of segment registers. Information about the 32 bit boot process can be found in the implementation of the Lguest virtual machine implementation in arch/x86/lguest, as well as in the official kernel documentation.

[1] extensible firmware interface

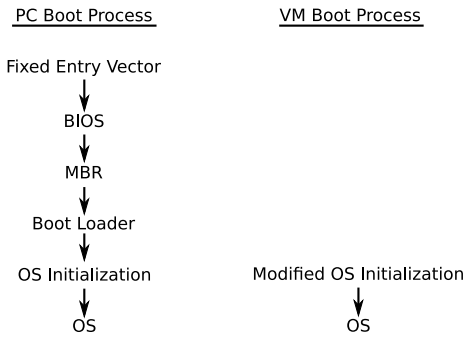

Figure 5.2: A comparison of the boot process of PCs and VMs.

The VMM has to do all system configuration that is normally done in real mode on behalf of the VM. This includes setting up the VM's initial segment descriptor table (GDT): Linux requires all segments to have the base address zero and the range of 4GB.

The information that is normally queried from the BIOS can be provided to the kernel through a data structure. It should include the video mode of the built in graphics card and the layout of physical memory. Additionally it includes the kernel's boot arguments and configuration that is normally provided by the boot loader.

An initial user land may be loaded from a file system image, which is loaded into physical memory. Such a file system image is usually called a ramdisk. For Linux the ramdisk is placed into the uppermost memory region,

A comparison of the boot process of PCs and VMs can be see in Figure 5.2.

5.2.2 SMP Support

In this section I will introduce how I implemented support for multiple CPUs.

I will follow the naming conventions that Intel laid out in the MP specification [Int]: The system boots from the *bootstrap processor* that is elected either by the BIOS or by the hardware. The bootstrap processor is used to boot the operating system. All other CPUs in the system are called *application processors* and are activated by the operating system.

An SMP system consists of a bootstrap processor and one or more application processors. Each processor in the system has its own advanced programmable interrupt controller (APIC).

The APICs have their own IDs and are able to send inter-processor interrupts (IPIs) to other APICs in the system. They also deliver interrupts from the system's peripheral devices.

On x86 systems with multiple CPUs, the BIOS provides the OS with information about all available processors and their status (enabled, disabled). This data is either stored in a data structure called the MP table, or in ACPI tables.

The PC boots with the bootstrap processor. During boot time, the operating system reads an in-memory data structure, the MP table, or ACPI tables to find other processors,

and initializes the APIC. It then calibrates and activates the APIC timer and sends IPIs to activate the application processors.

The Linux in the VM has the same boot sequence. The VMM starts up with one control loop, which is used as the bootstrap processor. Inside the VM, Linux does all initialization as described before.

There are two solutions to provide Linux with the information about how many (virtual) CPUs are available. Either the VMM creates an MP table (like described before), or Linux retrieves this information with hypercalls. I decided to use the former solution because the algorithm to create an MP table is simple, and I did not have to alter the early boot process of Linux, which can be tricky.

Once the bootstrap processor has done its initialization it activates the application processors. On a bare machine processor activation is done with IPIs, which are sent by the bootstrap processor's APIC to the target processor's APIC. On x86, the application processor is initialized in 16 bit mode, which I wanted to avoid. Therefore, I paravirtualized the activation of application processors with a hypercall. The VMM creates a new APIC instance and a VMCB that contains the CPU state of the application processor that is started. The application processor starts up in protected mode, and its segment and interrupt descriptor tables are set up in the same way like on the bootstrap processor. The VMM then creates a new thread that runs a control loop using the newly created VMCB.

The APIC implementation is similar to the IC implementation; An APIC can trigger an IPI at another APIC (both direct IPIs as well as broadcasts are supported). The control loop belonging to that APIC polls for interrupts at each VM exit, and injects them into the VM. Therefore, to keep IPI latency low, the target processor has to be forced to leave VM execution, which can be done by triggering a physical interrupt on that processor.

In Fiasco, IPIs are used to signal remote CPUs to deliver cross-CPU IPC and User-Irqs. IPC is synchronous, and would require the sender to wait until the receiver is ready. User-Irqs instead are asynchronous and allow the sender to continue execution immediately. I employ an additional event thread per CPU to wait for an incoming User-Irq. Because this thread runs directly on the target CPU, a User-Irq sent to this thread forces the target processor to leave the VM execution. Its control loop can then immediately inject the IPI.

The control loop of the bootstrap processor additionally polls at the IC for device interrupts. Therefore, in this implementation interrupts of peripheral devices are injected into the virtual bootstrap CPU only. Linux usually controls interrupt affinity, for example to do interrupt load balancing. In this thesis, I decided to ignore the interrupt affinity to decrease the APIC implementation's complexity. This is in conformance with the physical APIC's default behaviour, and in my testing Linux was able to handle it without problems. However, support for interrupt affinity can be added in the future, if needed.

APIC Timer Calibration Linux calibrates the APIC timer by disabling interrupts and reading the CPU's timestamp counter and APIC timer value two times in a row and comparing the difference in both values. While that algorithm works great on real

machines, it did not work in the VM. The APIC clock granularity is limited, because it can only be a multiple of the host timer granularity. During the timer calibration, a host interrupt may occur. Host interrupt handling increases the timestamp counter of the CPU, but may be short enough that the APIC timer does not change until control is given back to the VM. Because host interrupts cannot be disabled by the VMM, timer calibration in the VM gives arbitrary results even though the APIC timer has a fixed frequency. Therefore I disabled the timer calibration in Linux and instead provided it with a fixed value.

5.3 Peripheral Devices

Peripheral devices are needed for usability. For example, it is desirable for the VM to have access to the network, hard disk and to use a graphical console, including keyboard and mouse for input. To support these devices, I made use of existing device managers that multiplex the device functionality. L4Linux contained stub drivers for these device managers, which I used as a starting point for my implementation.

In the next section the main design pattern of the device driver stubs will be introduced. I will then give more details on the implementation of the serial line, L4con and Ankh driver stubs. Thereafter, I will explain how I let the VM access the physical hard disk.

5.3.1 Design Pattern

I will now explain a pattern that is common to all device driver stubs. In the following sections I will give more details about the specific drivers.

As described in Section 4.5.3, a stub driver is implemented in the guest operating system. It contains all driver logic and implements the interface to the guest operating system. More specifically, the stub driver glues together the logic of the corresponding device manager's IPC interface and the guest OS interface.

In the Fiasco hardware virtualization interface the VM cannot directly execute IPC. Instead, the stub driver's commands are mediated by the VMM.

Therefore, each stub driver is accompanied by a driver back end in the VMM (thereafter called **proxy driver**), which executes the IPC operations on its behalf. The driver back end contains a blocking thread that waits for incoming notifications. Notifications can be issued by device managers in the event of input, or by the microkernel to forward interrupts from hardware devices[2]. This thread will be referred to as **event thread**.

Upon receiving a notification, the event thread registers a virtual interrupt as pending at the IC, which is then injected into the VM by the control loop. To make sure that the latency between a device notification and the subsequent injection is low, event threads have a higher priority then the thread running the control loop.

Once the guest OS receives the interrupt, it executes the interrupt handler that belongs to the stub driver. Figure 5.3 shows a comparison of a stub driver in L4Linux and the driver setup in Karma.

[2] On Fiasco such notifications can be implemented either per IPC, or asynchronously with (User-)Irqs.

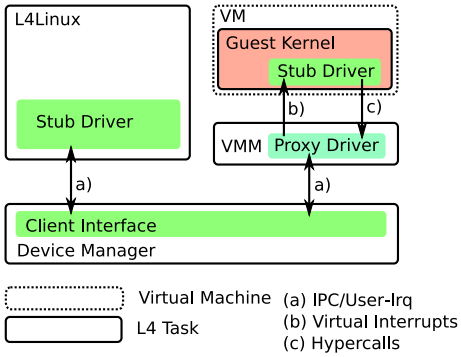

Figure 5.3: One the left hand side there is a standard L4Linux kernel with a stub driver. The stub driver consists of the driver logic and the L4 IPC interface for communication with the L4 device manager. On the right hand side both Karma and a Linux VM are depicted. The stub driver in the Linux kernel has been split so that the driver logic remains inside of Linux, and the L4 IPC interface resides in the VMM. A custom hypercall-based protocol connects both parts.

5.3.2 Serial Line

A serial line is the most simplistic interface for input and output. For a serial line only two operations are needed; A read operation reads a byte from the line and a write operation writes to the line. Availability is signaled to the client. Such a simple serial line interface is implemented in Fiasco and is used for the console. I implemented a small driver back end in the VMM to make use of this interface.

The stub driver was derived from the L4Linux serial driver by Adam Lackorzyński. The Linux serial line interface has operations for IO of single characters, but also for whole strings. Instead of implementing the single character interface only and breaking strings up to their characters, I implemented a hypercall that allows the VMM to print the whole string directly from guest memory.

5.3.3 Graphical Console

A graphical console is provided by the L4 device manager L4con. Clients get a shared memory area that they can use to draw their graphical output (frame buffer). L4con composes the final display from these shared memory regions. In L4con only one client is visible at a time, and an indication about which client is active is drawn to a special screen area that cannot be manipulated by clients. Operations such as screen refreshes are initiated by clients over the IPC based protocol L4Re::Framebuffer.

L4Linux contains a stub driver that implements the L4Re::Framebuffer interface, and thus acts as a client of L4con for graphics output. I used this stub driver with modifications: As described before, the VM cannot make use of IPC directly. Therefore

I split the L4Linux stub driver so that the IPC interface of L4con is implemented in the proxy driver in the VMM and the driver logic resides in Linux.

I mapped the shared memory region for graphical output, offered by L4con, into the VM's physical address space, and made it visible to Linux as device memory. Drawing an image to the screen is done by the stub driver by writing the image data directly into the device memory and then sending a command to the proxy driver to refresh the screen area. The proxy driver forwards the command to L4con to execute the refresh operation. A similar scheme is used for console input: Input events are stored in a shared memory region, and the proxy driver's event thread translates the notifying IPC into a virtual interrupt, which is then injected into the VM to be processed by the stub driver.

The L4Re::Framebuffer interface used by L4con is also used by DOpE, which is a console multiplexer that draws clients in floating windows. Therefore the VMM can make use of both console multiplexers without modification.

5.3.4 Network Interface

Network interface cards are multiplexed by Ankh. Ankh provides each of its clients with a shared memory area for in- and outgoing traffic. Each client gets its own MAC address and all routing between clients and the outgoing interface is done transparently by Ankh. IPC is used both by Ankh to notify clients of new incoming data and by clients to notify Ankh of outgoing data. Figure 5.4 depicts Ankh multiplexing a network card with both an instance of L4Linux and a VMM with Linux in a VM as clients.

When I started this thesis, no Ankh stub driver was available for L4Linux and I started the implementation of the stub driver by using the stub driver of Ankh's predecessor as template. Like for the graphical console, I made the shared memory region visible to Linux as device memory. The stub driver implements the Ankh client interface, with the exception of IPC, which is done by the in-VMM proxy driver.

5.3.5 Hard Disk

At the time this thesis was written, there was no working device manager for hard disks. Linux has a large driver base and therefore supports many devices out of the box. So one way of supporting devices that do not have L4 device managers is to allow one VM to drive the device. The driver VM could export a device interface to other VMs, but that is out of scope for this work.

A Linux driver communicates with devices either with IO ports or by reading and writing directly to device memory. In this thesis I allowed the VM to access the hard disk, which is connected to the system with serial ATA (SATA). The SATA controller implements the AHCI interface, which makes use of memory mapped IO.

To access the hard disk, Linux needs to be provided with device information, such as the interrupt number and the location of the device memory.

In the L4 system access to devices is managed by the L4 server *IO*. IO iterates over the PCI configuration space and ACPI tables to find attached peripheral devices. It allows the administrator to construct virtual busses that present clients with only the devices

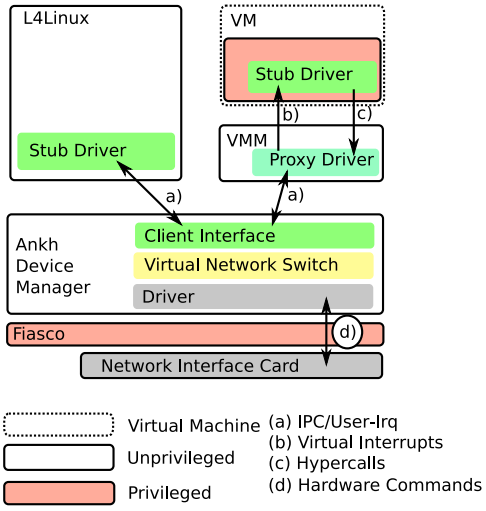

Figure 5.4: The Ankh network multiplexer with two clients: L4Linux and a VM running Linux.

the client should be able to directly access. As described in Section 5.3.1, I implemented both the stub driver and the back end to make use of this mechanism.

The AHCI interface relies heavily on DMA both to send commands to the device and to receive data from the device. DMA requires special handling because it requires host physical addresses. The VM does not know about host physical devices. Therefore I implemented a hypercall that makes the VMM query the data space provider for the host physical address of the guest physical address zero (base address). Linux uses this hypercall once on during initialization, and saves the base address. Later on, when setting up a DMA transfer, Linux uses this base address to calculate the host physical addresses without VMM involvement.

When I mapped the device memory directly into the VM to allow the Linux driver to command the device I encountered a problem. Linux accesses this memory through the nested paging address translation. To correctly use device memory, writes must not be cached. Although I made sure that the memory was mapped correctly on the host side (with the cache disable bits), the writes done by the Linux drivers were cached, and the device did not work. I could verify that caching is a problem by modifying the driver to explicitly flush the appropriate cache line (using the *clflush* instruction) after each IO memory write and the device worked as expected. I could then fix the problem by modifying Linux to set different cache attributes.

Letting the guest OS drive a device directly is security sensitive if that requires DMA, as described in Section 4.5.3.

Therefore, I also implemented another way to command the hard disk: In that scheme, I did not map the device memory into the VM's address space but only into the VMM's. I modified the Linux driver to call the VMM to both read values from device memory and write commands to device memory. This has the advantage that the VMM can verify the command, and the VM is removed from the system's TCB. This scheme can be extended,

to hand DMA commands out to an external server, which would remove the VMM from the TCB as well. However, external analysis comes at the cost of performance, as will be discussed in the evaluation chapter.

5.3.6 Summary

In my solution Linux has access to the most important peripheral devices. In my work I support the following devices:

Serial Line Interface A virtual serial line is provided with an interface that translates to the *log* interface of L4Re. Both the *vcon* Fiasco interface or any other service implementing the log interface can be used as back end.

Network Interface Card In this setup Linux contains a stub driver to make use of the multiplexing capabilities of the Ankh NIC multiplexer. In my measurements I found that this setup is able to saturate the physical interface. At the time of this writing the Ankh NIC multiplexer was experimental and I was not able to do reproducible benchmarks.

Graphical User Interface A graphical user interface (GUI) is provided with an interface to the *L4Con* protocol; both L4Con and DoPE can be used as back ends. The Linux stub driver implements the Linux framebuffer abstraction. As such it also supports the X11 window system for graphical output.

PCI Bus I added a stub driver that allows Linux to make use of virtual PCI busses as provided by the *IO* server.

Hard Disk With no device multiplexer for block devices in place, I chose to implement support for accessing a physical hard disk. This access must be restricted to one VM. Allowing the VM to make use of DMA transfers is sensitive with respect to security, and adds the VM to the TCB of all applications. Therefore, I implemented two methods to drive the device: Either the VM accesses the device's IO memory directly, or these accesses are handed out to the VMM. By handing the device commands out to an external server, the TCB might be further reduced.

5.4 Staged Virtualization

Support for a second-stage VM that does faithful virtualization is done with KVM. Running KVM in a VM upon the Karma VMM required the following adaptions:

Karma implements a back end, which provides the guest with a hypercall interface to initiate a world switch, to create and destroy a VM and to set up the VMs memory.

On initialization of a second stage VM, KVM does a hypercall to make the KVM back end acquire a fresh VM capability. A world switch into the new VM is done with a hypercall as well: The hypercall is augmented with the address of the second-stage VM's VMCB and the general purpose registers. Using this information, the back end

```
44 files changed, 2697 insertions(+), 133 deletions(-)
```

Figure 5.5: Output of diffstat

executes the *l4_vm_run_svm* system call, which makes Fiasco switch the CPU to the second-stage VM. Like in ordinary VMs, Fiasco ensures that it remains in control at all times. Therefore, the second-stage VM's execution is left upon physical interrupts. Thereafter the Karma control loop takes control and returns to the first-stage VM.

Initially the second stage VM's address space is empty. Upon VM page faults, KVM does a hypercall to map a page into the VM.

I could do the adaptions to KVM rather quickly, because the points in the KVM code that needed to be adapted, are the same as in KVM-L4.

5.5 Summary

Complexity I created the VMM with its future use in mind. Therefore, I wanted to keep its overall complexity low so that other people can easily understand the application logic and start extending it. An indication of the complexity of an application can be given with its total of lines of code. My VMM implementation has about 3800 lines of code. The line count alone can be misleading, so here are more numbers:

- Lines of code: 3800
- Files: 24 headers, 13 implementation files
- Classes: 21

An important aspect is the complexity of the modifications done to the Linux kernel. The code encompasses the stub drivers, adaptions needed to make use of DMA transfers using the physical hard disk and the modifications I made to the start up process of application CPUs (see Section 5.2.2).

The complexity of the patch to the Linux kernel is visualized with the output of *diffstat* in Figure 5.5. The stub drivers comprise about 2300 lines.

I was able to port the patch to a new Linux version in about one hour. I think that the effort to keep the virtualized Linux up-to-date is reasonably small.

Resource Footprint Karma VMM needs 284KB of memory for an instance[3], which is an improvement compared to KVM-L4. KVM-L4 requires L4Linux and Qemu, which at a rough estimate need about 16MB of memory on their own.

[3] The resource footprint of 284KB has been measured for an instance with 3 virtual CPUs. Each virtual CPU requires memory on its own. Therefore, instances with more than 3 virtual CPUs would require slightly more memory.

Additional resources are needed to run the L4 device managers. There is one device manager instance per physical device. Every device manager needs resources on a per client basis. The L4con secure console for example allocates a memory area with the size of the framebuffer that each client uses to draw its graphical output. Additionally, some memory needs to be available as main memory for the VM.

Limitations At the moment IO ports are not supported; writes to an IO port are ignored, and reads return a fixed value. In Linux, applications can request the rights to directly access IO ports from the kernel. An example is the *hwclock* application, which can read the current time directly from the system's clock. In turn, it can set the clock to another value by writing to its IO ports.

In my solution such applications do not work. In the future, IO ports can be emulated, if required.

6 Evaluation

The goal of this work was to run unmodified Linux applications on top of the microkernel Fiasco. I created a solution that is of low complexity (5.5), has a small resource footprint (5.5), and runs unmodified Linux applications, with few exceptions (5.5). An important aspect of my work was that the overhead in terms of execution time should be comparable with those of native Linux.

System The measurements were done on the following system:

- Machine:
  - CPU: AMD Phenom 8450 X3, 3 Cores, 2100Mhz, 128KB L1-, 512KB L2- and 2048KB L3 cache
  - Memory: 4GB
  - Hard Disk Controller: ATI Technologies Inc. SB700/SB800 SATA Controller
- Fiasco:
  - Version: 20 October 2009
- L4Linux:
  - Version: 12 November 2009
  - Linux Version: 2.6.31
  - Graphics: VESA, Mode 0x117
- Native Linux:
  - Version: 2.6.31.5
  - Graphics: VESA, Mode 0x117
- Karma:
  - Version: 4 January 2010
  - Graphics: VESA, Mode 0x117
- KVM Setup:
  - Nested Paging: Enabled
  - VGA: Disabled
  - Hard Disk Access: IDE
  - Version: KVM-84, Qemu 0.9.1
  - Host: Linux 2.6.31 PAE

Time Sources Time inside a VM is not guaranteed to be in compliance with the wallclock time. Instead, it may drift, and therefore be inaccurate. In my measurements I took such drift into account by using an external clock source.

Linux running in a VM on top of Karma issues hypercalls to get time stamps. The Karma VMM receives the hypercall and creates a time stamp from the CPU's time stamp counter. A time stamp was taken immediately before and after the benchmark, and the delta denoted the benchmark measurement.

In KVM, I used *ntpdate* immediately before and after the benchmark to make sure clock drift inside of KVM are evened out. Ntpdate is a tool that synchronizes the wall clock to an external server using the network time protocol (NTP).

Structure In the next sections, I will present the measurements I did to evaluate my work. First, I will present the benchmarks I used to give an impression of the existing implementation. These benchmarks are concerned with CPU virtualization, memory virtualization and overall performance. Second, I tweaked system parameters and measured how they affect the overall system performance. I did this to get an idea where further optimization might be needed. This chapter will be completed with measurements that evaluate the performance of second stage VMs.

6.1 Performance Benchmarks

In this section, I will evaluate the performance of Karma. First, I measured the performance of the CPU virtualization with a compute benchmark. Second, I devised a custom benchmark that stresses the OS interface as well as the performance of both the memory virtualization and –for SMP– the IPI implementation. Third, I measured the compilation of a Linux kernel from source to binary, to evaluate the overall system performance.

6.1.1 Compute Benchmark

To measure the performance of CPU virtualization, I used an application that uses a brute force approach to find prime numbers. The measured run time is given in Figure 6.1. As expected, CPU virtualization imposes an insignificant overhead to compute-intensive workloads.

	Time	Percent
Native	5m 4s	100
Karma	5m 5s	99.67
Nested VM	5m 8s	98.70
KVM	5m 7s	99.02

Figure 6.1: Compute Benchmark

```
#!bin/sh
sh test.sh eins &
EINSPID=$!
sh test.sh zwei &
ZWEIPID=$!
sh test.sh drei &
DREIPID=$!
wait $EINSPID
wait $ZWEIPID
wait $DREIPID
```

Figure 6.2: bench.sh

6.1.2 Custom Benchmark

I devised a small custom benchmark to inquire about the performance of the memory virtualization. It stresses the Linux system call interface by creating a huge number of processes that do few computation. In compliance with its file name, I will call it the bench.sh benchmark. Bench.sh and its sibling test.sh can be seen in Figures 6.2 and 6.3.

This benchmark causes Linux to send a huge number of IPIs to reschedule and migrate processes, and is therefore ideal to measure the performance of the IPI implementation. Process creation and destruction also stresses the memory subsystem, address spaces need to be created and destroyed. Additional computation is needed for synchronization in the file system layer of Linux. Again, I compared native Linux, Karma Linux, L4Linux and KVM Linux.

If the benchmark is executed with one virtual CPU, its runs faster in a VM established by Karma, than it does in native Linux. I think that this is a result of scheduling anomalies caused by the virtual clock, which is currently running slightly slower than the physical clock, and the difference in the amount of injected timer interrupts correlates with the speedup.

```
#!bin/sh
FILE=$1
count=0
while [ $count -le 100000 ]
do
    count=$[$count+1]
    rm -rf $FILE.txt
    touch $FILE.txt
    dd if=/dev/zero bs=500K count=3 of=$FILE.txt 2>/dev/null
    gzip -9 $FILE.txt
    cat $FILE.txt.gz > /dev/null
    rm -rf $FILE.txt.gz
done
```

Figure 6.3: test.sh

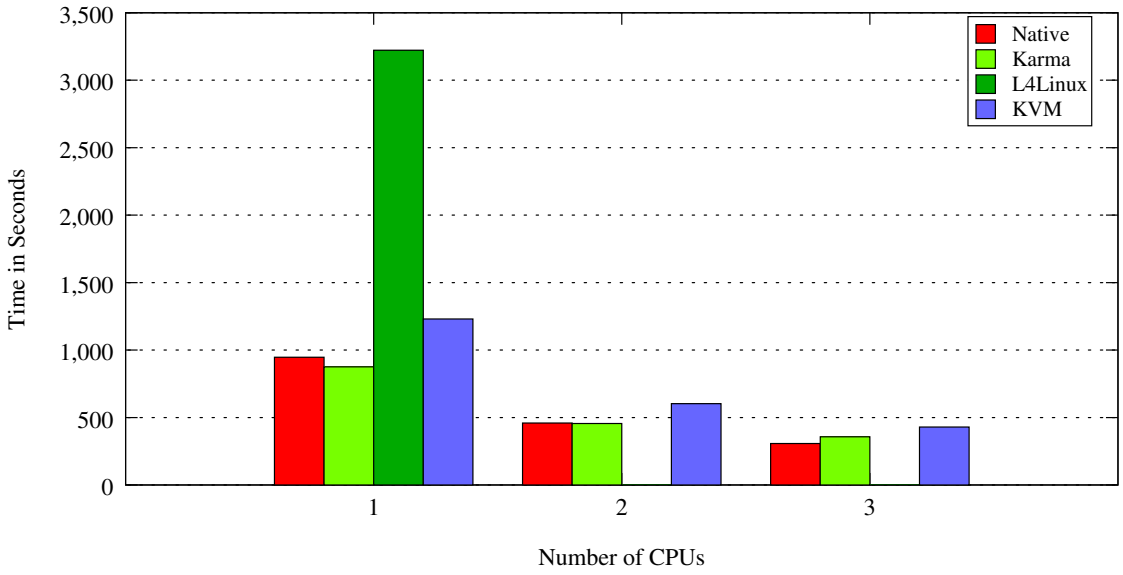

Figure 6.4: Bench.sh benchmark

The bench.sh benchmark is a pathological case for L4Linux, which shows bad performance. The bad performance of L4Linux is caused by the frequent interaction with the Fiasco microkernel to create tasks, to populate them with memory and subsequently to destroy them.

6.1.3 Kernel Compile

The performance of a Linux system can be measured in many ways. A benchmark that stresses a lot of operating systems services is compiling the Linux kernel.

In the measured scenario, I used an install of the Linux distribution Debian in version 5 (Lenny) as basis. The compiler version is gcc 4.3.1. The compilation time is measured with an external clock, as described in in the previous section, and therefore does not rely on virtual time, which may drift. The measured results can be seen in Figure 6.5.

To reduce the performance impact of L4con, which is used for in- and output, and therefore minimized the amount of text output of the kernel compile. Furthermore, I disabled the Ankh network multiplexer. All data, including the compiler binary and the kernel source code, are located on the physical hard disk. I ran the kernel compile twice and measured the second cycle only. Thus, I can ensure that all caches used by Linux are warm and the influence of hard disk latencies is minimized. In all scenarios, the VMs were equipped with 420MB of main memory.

The performance of the Karma VM is compared to a natively running Linux and L4Linux. All measured kernels had the same kernel configuration, with the exception of Karma and L4Linux specific settings. All configurations used a framebuffer device for output. Additionally, I measured native KVM, whose performance should closely match that of KVM-L4. The results can be seen in Figure 6.6. For clarification, the overhead compared to native Linux is depicted in Figure 6.7.

```
#!/bin/sh

tar xjf linux-2.6.30.5.tar.bz2
cd linux-2.6.30.5
make i386_defconfig
make -j 3
make clean
./time
make -j 3 >/dev/null 2>&1
./time
```

Figure 6.5: Script used for measuring the run time of a kernel compilation. The ./time binary is a custom binary which does a hypercall to make the VMM take a time stamp. The difference of the two printed timestamps is the actual run time. In KVM ./time was replaced by ntpdate, which synchronized the VM's clock to an NTP server, and the time stamps were taken after the synchronizations.

The Linux X11 framebuffer drivers do not refresh areas of the screen, for example to keep track of mouse-pointer movements or video playback. Therefore, both L4Linux and Karma employ a thread, which refreshes the whole screen repeatedly to make sure all drawing operations become visible. For this benchmark, I disabled the periodic redraw operation because it is not needed for console output and seriously degrades performance (I will revisit the refresh overhead in Section 6.2.3).

I expect the virtualized Linux to have only slightly reduced performance compared to native Linux. Overhead occurs because of the execution time that is needed for both the VMM and its transitions to the Fiasco microkernel to execute a world switch. Since both the VMM and the Fiasco world switch path have low complexity I expect this overhead to be small.

The KVM setup is included only to give an impression of the performance of a well-established virtualization platform. However, the measured setup differs in important properties: Contrary to all other measurements, which use AHCI for hard disk access, the benchmarks in KVM used IDE. Furthermore, I disabled graphical output and instead used a serial line. Because of these differences, the numbers measured in the KVM setup are not directly comparable to the other benchmarks.

The measurements met my expectations: With one CPU, Karma Linux ran the kernel compile with a negligible performance overhead. For SMP, the overhead is slightly larger, which results from the IPI implementation, which leaves room for optimization. In a more general view, the Karma scales well for multiple CPUs.

In my measurements KVM performed worse than expected. I measured the virtualization overhead of KVM to be about 15 percent. I expected the overhead to be about 8 to 10 percent, which are caused by the switches between the VM and Qemu. The measured overhead can be a result of the different setup, for example, KVM used IDE to access the harddisk, whereas Karma, L4Linux, and native Linux used SATA. I suppose that with more thorough tuning the KVM performance can be significantly improved.

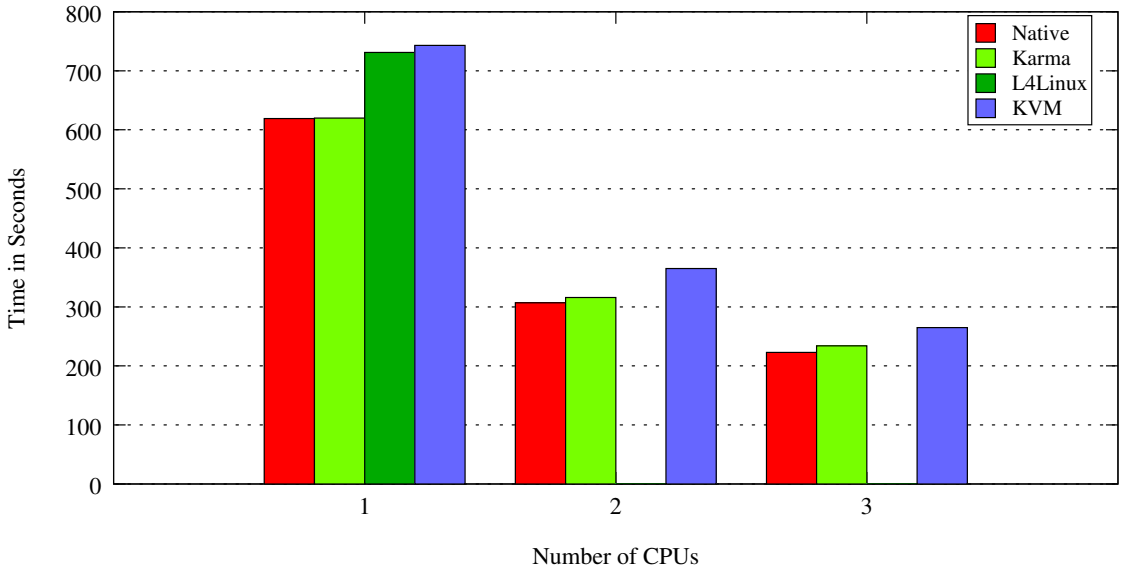

Figure 6.6: Kernel compile times

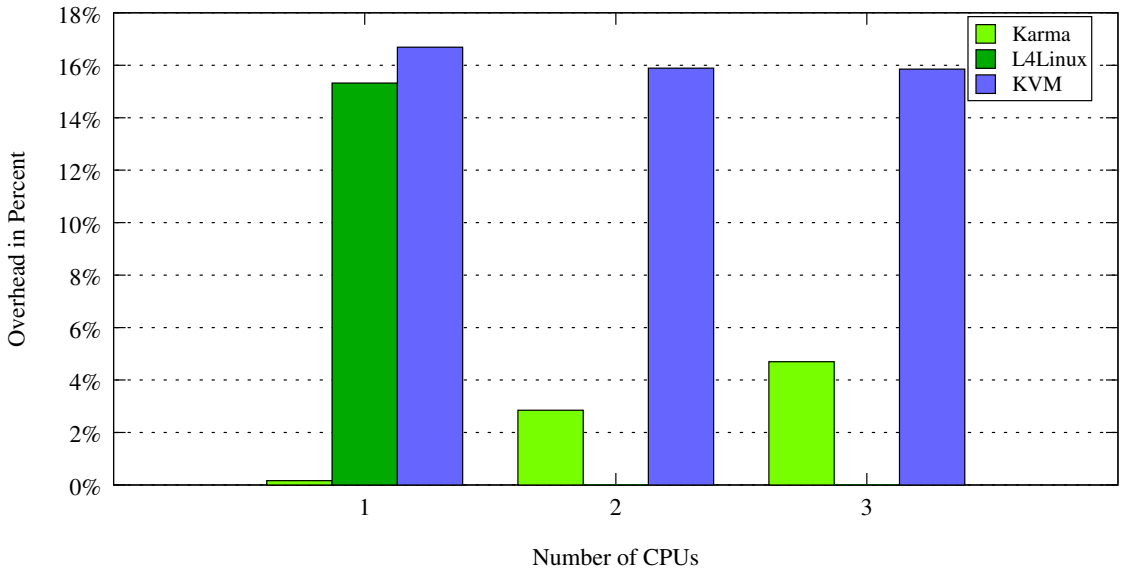

Figure 6.7: Kernel compile overhead relative to native Linux. No data is available for L4Linux on 2 and 3 processor setups.

CPU	0	1	2
Hypercalls	85275	12206	11617
Preemptions	1583361	191852	194254
Pending Interrupt Injections	85	16	16
Hlt	8736	9591	9250

Figure 6.8: Vmexit reasons during kernel compile.

6.1.3.1 Vmexit Reasons

An interesting aspect is why and how often the VM execution is left during a real-world load scenario. Therefore, I measured the kernel compile: the VM was configured to use three CPUs, 512MB Ram and to use the L4con Framebuffer-interface. The numbers are to be seen in Figure 6.8.

Additionally, I wanted to know which device back ends received the most hypercalls:

- IC: 66572
- APIC: CPU0: 11594, CPU1: 8098, CPU2: 7712

These numbers indicate that the IC interface and the APIC interface are used very often. As an optimization, these interfaces could be implemented with a data structure in guest memory. The guest would write the commands to this data structure without having to issue a hypercall. The back ends would then query the data structure to update their state when needed. For example, if the guest would mask an interrupt, it would put a flag into the data structure. When a device thread wants to flag and interrupt as pending, the IC would query the data structure and find out the interrupt has been masked.

6.1.3.2 Idle VMs

To estimate how much computation time is lost on idling VMs, I measured the kernel compile with different numbers of idle VMs running side by side with the measured VM.

The measured numbers are visualized in Figure 6.9. As expected, the overhead of idle VMs is small. Interestingly, the kernel-compile times did not increase linearly with the number of idle VMs. Instead, for odd numbers of idle VMs the kernel compile took longer than the same benchmark with one more idle VM. I guess that this slowdown is a result of a scheduling anomaly.

As an optimization, the VMs could be configured to use one-shot timers to reduce the frequency of timer interrupts when there is little load, which would increase the time of hibernation.

6.1.3.3 Handing IO Memory Accesses to the VMM

To reduce the TCB of the system it is desirable to not let the VM directly execute any DMA transfers. To do this, I implemented an alternative; The Linux AHCI-driver

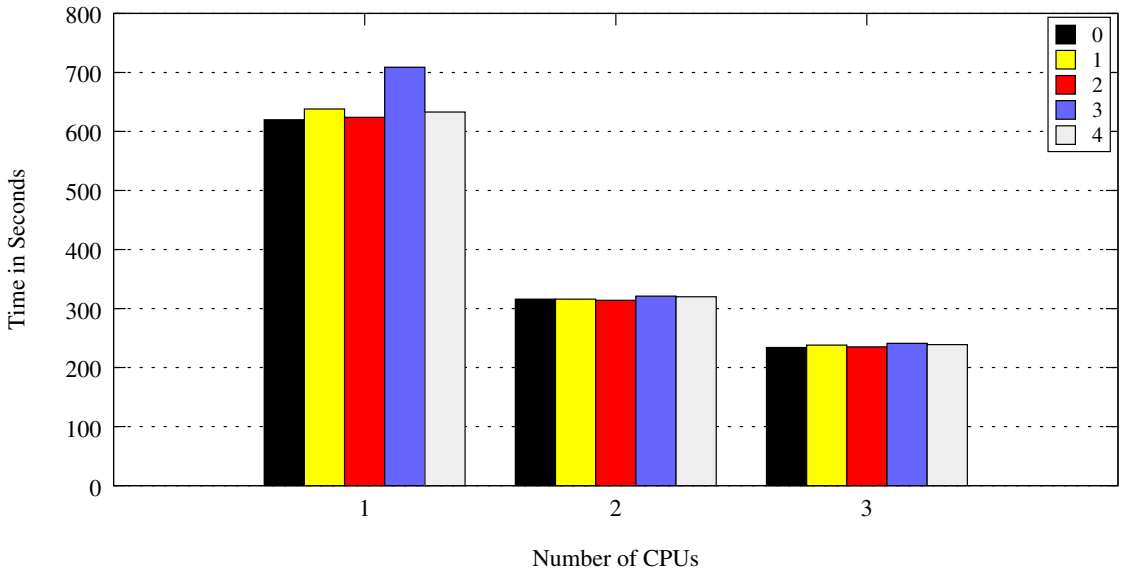

Figure 6.9: Kernel compile with idle VMs.

resorts to hypercalls, to make the VMM analyze the command and execute it on behalf of the driver. To further decrease the TCB, device commands could be handed out to a separate task.

The actual implementation of the algorithms needed to analyze the device commands is out of scope for this work. Instead, I slowed the hypercall down to get an impression of how much performance such an analysis would cost. The slowdown was done with a tight loop. The first experiment slowed all write accesses to the IO memory down by 10000 cycles, which simulates the analysis in the VMM, which would consist of copying all commands out of the VM, analysing them and finally issuing them. Read accesses are used to inquire the state of the device and previous commands. For read operations, little copying is needed, and the analysis should be straightforward. Therefore, I slowed down read accesses by 3000 cycles only.

In the next experiment, I increased both slowdowns by factor 10 to simulate analysis in a external server, which requires IPC, and is therefore slower than analysis in the VMM.

Whereas both experiments use pessimistic numbers, the measured numbers indicate that the overall slowdown is small, and the proposed setup is feasible.

Below are the kernel compile times with the same setup like in 6.1.3 for comparison:

For the kernel-compile benchmark, the additional world switches resulting from the hypercalls that instruct Karma to do IO-memory accesses (read and write) on behalf of the VM, do not impose a significant performance degradation. Even command analysis in a separate task is feasible and the performance degradation is small.

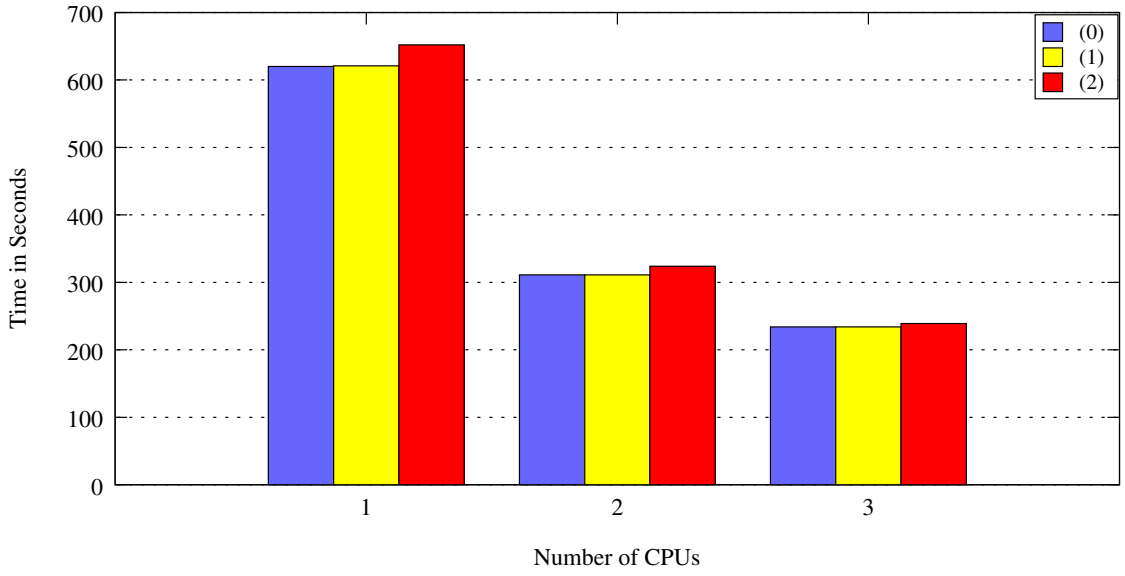

Figure 6.10: Comparison of kernel compile different types of direct hard disk access. (0) is the kernel compile in VM with direct hard disk access, whereas in (1) and (2), Linux has no access to IO memory and instructs Karma via hypercalls. In (1) Karma spends 10000 cycles before writing the command and 3000 cycles before reading values, to simulate analysis of the commands. In (2) Karma waits 100000 cycles before writing and 30000 before read, to simulate command analysis by a separate task.

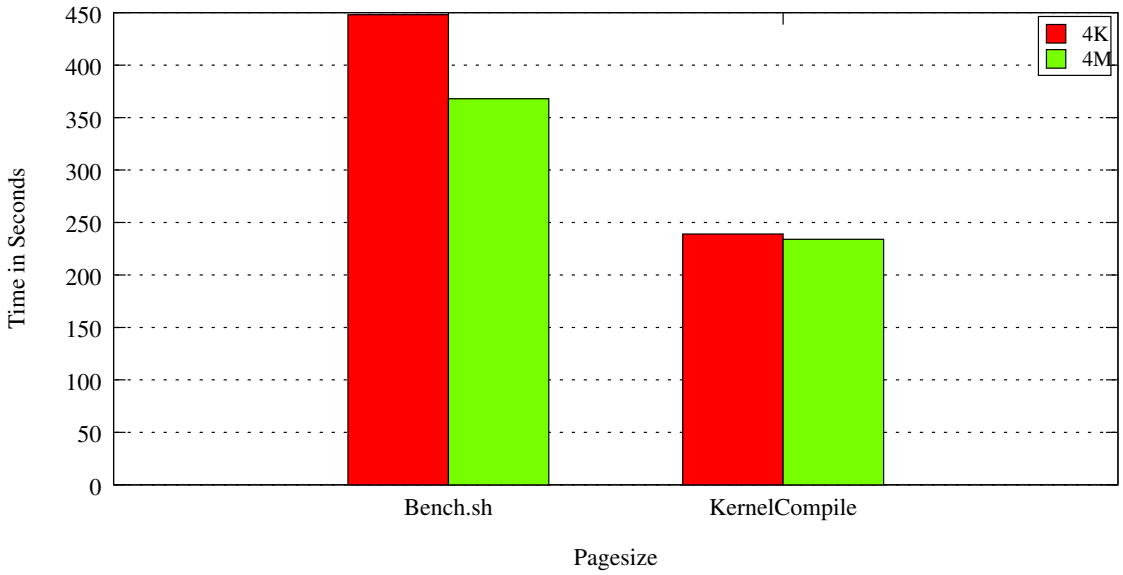

Figure 6.11: Comparison Performance of Bench.sh and a kernel compilation with 4K and with 4M pages (superpages).

6.2 Variation Benchmarks

In this section, I will present three benchmarks. In these benchmarks, I tweaked system parameters and measured their impact on the overall performance. I did this to get an impression where the system can be optimized.

6.2.1 Impact of Different Page Sizes

An interesting aspect is the page granularity of the host address space that is used as guest physical memory. With nested paging, two address translations need to be done: From guest virtual to guest physical and from guest physical to host physical. Both translations are cached in the translation lookaside buffer (TLB). To reduce the TLB pressure, address spaces used for nested paging are augmented with a address space identifier (ASID). The host has ASID zero, whereas VMs can be assigned their own IDs. With ASIDs, VM page translations do not evict TLB entries of the host.

If the host address space used as guest physical memory is built with super pages (4MB), less work is required to translate from guest physical to host physical. To show this effect, I measured both bench.sh and a kernel compile in a VM with 3CPUs and 512MB Ram, both with 4K pages and 4M pages. The results can be found in Figure 6.11.

The impact of a fine grained host address space on the guest's performance is about 18 percent for the bench.sh benchmark, which is rather large. For the kernel-compile benchmark the difference is much smaller (about two percent).

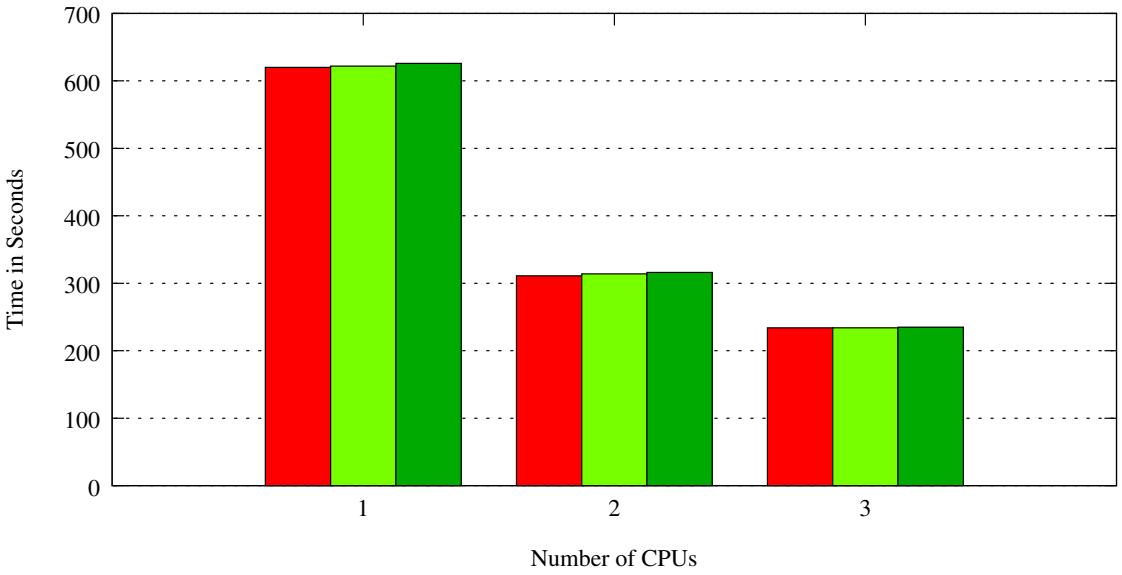

Figure 6.12: Kernel compile benchmarks with a custom slowdown of the world switch path. The red bars denote zero slowdown, light green denotes a slowdown of 5000 CPU cycles and the dark green bar denotes a slowdown of 10000 CPU cycles.

6.2.2 Impact of World Switches

An important aspect in virtualization is how fast a switch between guest and host and vice versa can be executed. To get an impression how the world switch influences the overall system performance, I measured the kernel compile with an artificially degraded world switch path. The VM used 3 CPUs, 512Mb Ram and the framebuffer interface.

The measured numbers are depicted in Figure 6.12, and indicate that the world switch path complexity does not have a significant impact on the performance of complex tasks such as a kernel compile.

6.2.3 Screen Refresh

One of my goals was to support X11 (graphical) applications. Because X11 can make use of framebuffer devices X11 applications are supported with the L4con interface. However, the X11 framebuffer drivers do not update the screen for example after the mouse-pointer moved. Similarly, windows containing video playback are not refreshed. In both L4Linux and Karma we use a thread to periodically refresh the whole screen. This seriously degrades the overall performance. To show the impact on performance, I measured a kernel compile in L4Linux with and without the refresh thread. The results can be seen in Figure 6.13. The refresh thread causes the performance to drop by about 33 percent. These numbers apply to Karma as well, because the mechanism is the same.

	Kernel Compile Time
L4Linux with Refresh Thread	18m 25s
L4Linux without Refresh Thread	12m 11s

Figure 6.13: Kernel Compile in L4Linux with, and without the refresh thread.

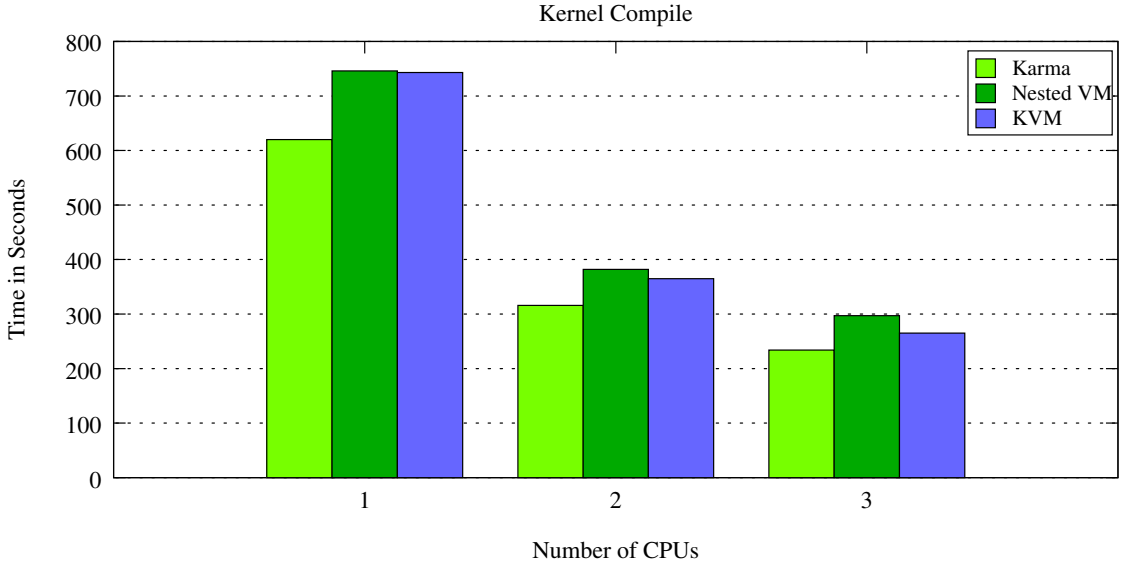

Figure 6.14: Comparison kernel compile in nested and native KVM.

A possible optimization is to keep track of the dirty state of the memory pages that contain the images within the framebuffer. A refresh would then only refresh those screen areas, whose dirty state has changed.

6.3 Staged Virtualization

The performance of the KVM port as described in Section 4.6 can best be compared to native KVM. I did the kernel-compile measurements with the same kernel, compiler and environment configuration as described in Section 6.1.3. Each VM was equipped with 420MB memory. I did not enable the Karma VMM's refresh thread, and I disabled VGA output in both KVM and nested KVM. The measured numbers can be see in Figure 6.14.

The KVM port is in an early state, whereby little optimization has been applied. In this setup, each KVM world switch requires two physical world switches: One to issue the vmrun-hypercall and another to switch into the second-stage VM.

7 Real-time Tasks in Virtual Machines

Electronic control units have changed the way how complex systems are built. They find application in various fields ranging from commodity multimedia players, over industrial installations, to safety critical airplane controls. In these scenarios, the logical correctness of a computation alone is not sufficient; The utility also depends on timely availability of the result.

Complex systems such as cars contain dozens of controllers, which poses a problem with regard to cost, weight, power consumption and reliability. Device manufacturers therefore strive to consolidate applications that used to run on dedicate control units onto fewer compute nodes. However, running on the same node gives rise to the problem of erratic interference. A crashing controller task must not bring down other tasks. Likewise, all tasks on a node must be executed in a timely manner that allows them to meet their deadlines. The OS running the node must therefore enforce strong spatial and temporal isolation. Informally, temporal isolation is ensured if the only reason for a job to miss its deadline is a contract violation –e.g. a worst case execution time (WCET) overrun– on its part. In particular, the misbehaviour of other jobs will never result in deadline misses of conforming jobs.

A microkernel enforces isolation between components. It can therefore isolate RT tasks that are placed in different protection domains. As spatial isolation is not sufficient for RT workload, appropriate measures have to be taken to guarantee timely execution. All components between the hardware and the actual application have to be designed with that goal in mind. Besides the microkernel's scheduler, a VMM is of particular interest in that respect.

The remainder of this chapter is structured as follows: I will first analyse what constitutes a real-time OS (RT-OS) and supplement it with a case study of Xenomai, a typical representative. Afterwards, I will present three designs for RT-VMs.

7.1 Background

Instead of consisting of one continuous computation, RT tasks are made up of jobs, which become ready for computation at different instants of time. The timely execution of jobs can only be guaranteed, if they have well known characteristics, such as a bounded worst-case execution time, or minimal time between releases.

With that information at hand, the OS can check a priori for a given set of RT tasks whether a feasible schedule exists (admission). A task is only accepted for processing if its admission does not preclude a feasible schedule.

During execution, the OS can monitor the behaviour of individual jobs, in particular whether it stays in the limits that were specified during its admission. A job that misses

its deadline can cause subsequent jobs of other tasks to miss their deadlines as well. Therefore, if a contract violation is detected, the OS may take actions such as suspending the wrongdoer and thus allowing all compliant tasks to meet their deadlines.

It is the responsibility of the scheduler to pick the next job to run among all runnable ones. Scheduling decisions are taken at events such as release events or blocking of tasks. After each scheduling decision, a *dispatch* operation makes the chosen job run. In classical systems, both scheduling and dispatching are tightly integrated within the scheduler.

7.2 Case Study: Xenomai

Xenomai is a system that combines a real-time executive and a standard OS. Its architecture is depicted in Figure 7.1. It is built upon Adeos (Adaptive Domain Environment for Operating Systems), a small component that has exclusive control of interrupts. Adeos runs entities, which are called *domains*. Domains are fully preemptable and ordered according to their priority. Adeos presents them with events, for example interrupts, according to their order.

A domain is free to take appropriate measures when presented with an event of interest. In particular, it may decide to keep executing. An RT domain would have a higher priority than non-RT domains. Lower priority domains can consume only the fraction of CPU time that the higher priority domains are idle.

To support complex scenarios in a well-defined manner, Xenomai implements semaphores and mutexes for synchronization within the RT world. The synchronization facilities are aware of scheduling related issues such as priority inversion, and take appropriate measures to avoid them. Additionally, it provides streamlined communication channels between the Xenomai and the Linux domain. This can be used for example to analyse data accumulated by RT tasks in regular Linux applications [Ger].

After the Linux kernel has booted the system, Adeos is brought into position by loading it as a kernel module. With Adeos in place, additional domains can be set up. After Adeos has taken over, Linux can still run non-RT applications. RT tasks are set up with the help of the Linux domain. After they have been loaded, they enter into RT execution mode, whereafter the RT domain schedules them.

RT tasks run in privileged mode with no means of isolation among each other. Even worse, any malicious or faulty RT task may bring down the whole system.

7.3 Single Unmodified RT Guest

The simplest means to achieve RT capable VMs is to give special treatment to one designated VM. An RT OS is used inside it to provide an appropriate environment for its tasks. Scheduling relevant events like timer expirations are first brought to the attention of the RT VM. When the RT VM yields the CPU, other activities are given a chance to run.

Depending on the objective, the VM should not use all the CPU time and allow the host OS and its (best-effort) applications to run as well. The mechanisms required to

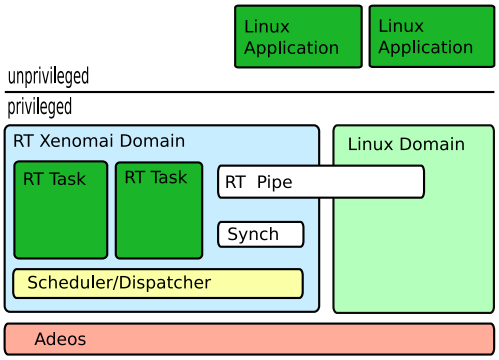

Figure 7.1: Xenomai architecture

enforce that are rather simple: The kernel tracks what portion of time is not claimed by the VM. If that fraction drops below a negotiated value over a longer period of time, the VM is considered non-complying and completely suspended. Therefore a misbehaving RT VM cannot bring the rest of the system to a halt.

If the RT OS signals idle periods, for example by executing the interceptable *hlt* instruction, no modifications are needed at all. However, a major drawback of this design is that such a system can only accommodate one RT VM per CPU. Consequently, it is up to the RT OS to isolate RT tasks from one another.

The compliance on the VM's part mainly hinges on the RT OS. If the RT OS does not guarantee to comply with the contract that coverns the relationship between the RT and best-effort tasks, the operator has to weigh the importance of RT tasks up against the importance of the best-effort tasks: Suspending the RT VM in the case of misbehaviour is not an option for critical RT applications. However, availability is also an important part of security; secure best-effort tasks must not be brought to a halt by the VM.

7.4 Multiple Slightly Modified RT Guests

Whereas running RT tasks in a VM shields the remainder of the system, RT tasks can still interfere with each other for the common case that the RT OS does not provide adequate isolation. VMs suggest themselves as protection domains, which leaves the problem of scheduling more than one of them in a timely manner.

One solution is to run multiple slightly modified guest OSes, which are scheduled round robin with fixed shares of CPU time. The guest OSes are given the illusion of running on a slower CPU.

The guest OS signals timeouts that it needs to implement its internal schedule to the host, which is in command of the system timers. Release timeouts, marking the point in time when an RT task gets ready, are accepted without modification. The guest may use timeouts to provision against WCET overruns. These timeouts are calculated by the guest under the assumption that the job runs on a slower CPU. In reality, it is run at full native speed within the guest's time slice. The difference in computing time needs to

be taken into account by the host, which is done by shortening the provided job length with the slowdown factor. In the same way compute timeouts need to be shortened.

Such a design allows multiple modified RT guests that need only slight modification, and achieves isolation of RT tasks provided they reside in different VMs. Using their original admission checks, guests make sure that their tasks are schedulable under the given conditions (CPU slowdown). The host has to determine a VM time slice length that allows each guest to meet their deadlines. If chosen too long, then a guest might be allocated CPU time at a point in time that is too late for it to meet its deadline. On the other hand, if chosen too short, then every guest will meet its deadline but secondary effects such as switch overhead may become so large that they have to be taken into account, which will also negatively affect the schedulability.

This design is limited to time driven RT job releases. Event driven job releases can occur any time the external event (for example an interrupt) occurs. Even if the host would know which event belongs to which VM and could inform the VM of the event, the latency until the job can actually be executed is bound to the fixed VM time slice length. This latency can be big enough to make the job miss its deadline.

7.5 Modified RT Design

I propose a different scheme whereby scheduling is done by the host scheduler. Compared to the previously sketched approaches, more intrusive modifications to the guest OSes are necessary.

As with the two previous designs, I seek to reuse as much as possible of complex infrastructure that RT OSes provide. Compared to the previous designs, more of the scheduling facilities will be moved from the guest VMs to the microkernel:

In the envisioned design, all scheduling decisions are taken at the host. Just like in an ordinary RT OS, the host is the only entity in the system that has a global view on all RT tasks and can take informed decisions. Support for scheduling decisions that require intensive knowledge about the synchronization facilities and resource access protocols provided by the guest require further research, which is out of scope for this work. The steps needed to make a job actually run are done by the guest's dispatcher. I will now detail the design with a description of its workings.

Before entering them into RT execution mode, the RT guests register each of their RT tasks with the host, who does an admission test encompassing the RT tasks of all guests. For any admitted task, the host establishes an identifier, which is also brought to the guest's notice.

In RT operation mode, the host scheduler picks the most eligible job and generates an upcall[1] to the job's VM. The job's identifier, which accompanies the upcall, allows the guest's dispatcher to find the corresponding job and resume its execution. Job completion would be signaled to the host with a hypercall.

The envisoned design allows multiple RT VMs. Therefore, separate VMs could be used to isolate RT tasks from one another. Because the host scheduler is provided with enough information, and a defined communication channel with the guest's dispatcher is

[1] An upcall may be implemented as a injected interrupt.

in place, event driven RT jobs are supported. Furthermore, the design is much more flexible than the other two designs and allows up to 100 percent CPU utilization.

This design places the following requirements on the VMM; It needs to implement an hypercall for the admission of guest RT tasks, and it also needs to be able to relay scheduling decisions to the guest's dispatcher with a bounded delay.

Conceivable Implementation I now detail how such a scheme could be implemented on top of the current Fiasco microkernel. In the current interface, kernel scheduling contexts are bound to threads. Therefore, upon admission, the VMM creates one thread (RT-thread) with the scheduling parameters given by the guest for each RT task. The microkernel schedules a job by running the associated thread. It is up to this thread to trigger further actions that eventually lead to the job to run in the VM. Once an RT-thread is activated, it writes its identifier into a known location in guest memory. Then it signals the scheduling event to the guest by marking an interrupt for injection and initiating a world switch[2]. The guest interrupt handler recognizes a dispatch event. Instead of activating the scheduler –as it would do in an unmodified system– it directly sets the dispatcher in motion. That, in turn, reads the identifier, finds the corresponding job and makes it run. The RT thread's control loop is depicted in Figure 7.2

A return from the *world_switch()* system call can have two reasons: Either the job is completed, or the job has been preempted. When a VM is scheduled after an preemption, control is not transferred back to it. Instead, its VMM is given control. That is different from the execution of threads, which are first class kernel objects with respect to scheduling and are not controlled by a third party such as a VMM. For them, preemptions are transparent and thus not directly noticeable. Preemptions occur for example when a higher priority job is released and is immediately scheduled. Scheduling decisions are taken at the host, and are visible to the RT thread only with a return from the *world_switch()*. Therefore, after returning from the *world_switch()*, the control loop has to check whether the job is completed[3] or it has been preempted in favour of another job. In the latter case, the execution of the unfinished job has to be resumed. To that end, the job's identifier is made visible to the guest and a dispatch event is injected.

Because there is only one location for the task ID, which is used by all RT threads, a preemption between writing the ID and entering the VM gives rise to a race condition. As a result, the guest might receive the wrong ID and thus dispatch the wrong RT task. Supplying the ID and the world switch must therefore be atomic. Unfortunately, that cannot be ensured by the current design because the RT threads cannot influence their preemption. Thus, a solution has to involve the microkernel. The world switch system call could be enhanced with two additional parameters; The ID and a memory address inside the VM, where the ID should be written. Before executing the world switch, the kernel would check whether the VM has enabled interrupts, which is an indication that the previous job was dispatched, and temporarily disable concurrency (for example by

[2] This mechanism assumes that the previous VM resumption has made all the VM state available, i.e. it does not hold parts of it on its kernel stack where it is hard to recover from. Only with the last state is it possible to resume the VM.

[3] Job completion is signalled with a hypercall, which can be detected by the RT thread by analysizing the VMCB.

```
while(task active)
{
    wait_for_release();
    while(job_not_done())
    {
        write_job_id();
        world_switch();
    }
}
```

Figure 7.2: RT task control loop

using a lock) and write the specified ID into guest memory before executing the world switch.

Figure 7.3 gives an impression of the targeted system architecture.

7.6 Summary

In this chapter, I introduced RT support for virtual machines. I discussed several architectures and presented a sketch of the implementation of a solution for real-time VMs that is flexible and does not compromise on CPU utilization. Its actual implementation is out of scope for this thesis and part of future, more elaborate work.

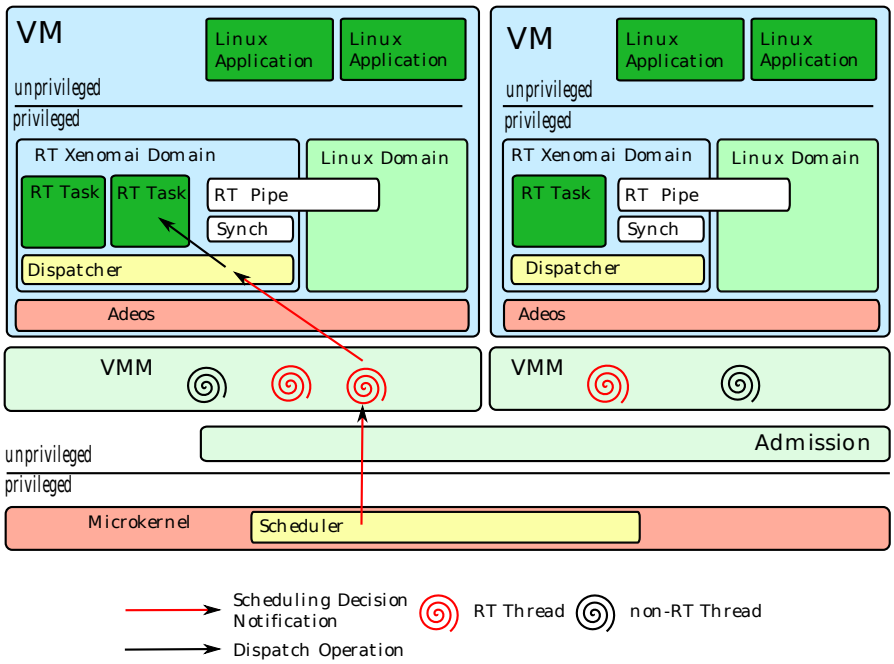

Figure 7.3: Real-time VM setup

8 Outlook and Conclusion

8.1 Outlook

Optimizations Further performance increases can be achieved with the following two optimizations.

- In the current implementation, updates to the IC, such as masking an unmasking of interrupts, are done via hypercalls, each of which causes execution to leave the VM. Instead, the IC stub driver could update the IC state in a VM accessible shared memory data structure, which is queried by the IC back end.
- In this work, the Fiasco world switch system call returns on every host interrupt, regardless of the VM state. As an optimization, a flag in the calling thread's UTCB could indicate the availability of virtual interrupts. On physical interrupts, the world switch system call would return only if this flag is set, which would decrease the amount of surplus VMM involvement.

  In a more advanced scenario, event threads could write the virtual interrupt into the UTCB, and the microkernel would inject them into the VM. Thus, the control loop's duties would be reduced to hypercall and error handling. However, such a solution comes at the cost of increased kernel complexity. For the KVM port, an additional flag needs to be introduced, which disables virtual interrupt injection and instead returns to the control loop. Otherwise, virtual interrupts intended for the first-stage VM would be injected into the second-stage VM.
- The KVM port could be optimized to not return from the second-stage VM into the first-stage VM on every physical interrupt. A return into the first-stage VM is only needed, if virtual interrupts need to be injected. After leaving the second-stage VM, the control loop would check for pending virtual interrupts, and would resume the second-stage VM immediately if none are pending.

Future Work This work opens up a range of possible projects. I will now introduce the most important ones that I have in mind.

Device Manager for Block Devices Currently the hard disk can only be accessed by a single VM, but it cannot be shared directly among VMs. Sharing would be possible for example using a network file system that is exported by the VM with write access and used by all other VMs. A solution for direct disk sharing would be to implement a device manager. It would include the hard disk drivers, and export a block based interface to clients. The multiplexing can be done in several ways, for example by providing each VM with its own virtual hard disk that is

represented by a file on the physical hard disk. Another solution would be to allow write access to a single VM, and read only access to multiple VMs. This setup, whereas simple to implement because the device manager would not need to know about file systems, is rather restrictive to the VMs, but would enable setups where a rich user land is needed, which cannot be loaded using a ramdisk.

Snapshots Snapshot functionality could be helpful for applications that need failure resilience. I could imagine important web servers to run in a VM. The VMM would detect errors, for example caused by attacks, and reset the VM to a known, save state. Snapshot functionality could be supported with Dirk Vogt's L4ReAnimator, as introduced in his diploma thesis [Vog09].

Migration VM migration is a tool that is often used in load balancing scenarios: In a data center, where there are multiple machines running the same microkernel-based system, both Karma and its VM could be migrated from crowded nodes to nodes with little load. With the aforementioned snapshot mechanism in place, migration is a simple as copying the snapshot data to the other node, and subsequently restoring from the snapshot.

Systems Debugger The VMM could be extended with a systems debugger. The a debugger could be built upon the systems debugger created by Julian Stecklina in his diploma thesis [Ste09]. Such a debugger would be a tremendous help for system developers. Together with a snapshot mechanism this could be even more useful, because a VM could be stopped at a certain point in execution, and different scenarios could be played through while debugging.

System Services A Linux system provides a number of services that would be useful to L4 applications as well. A prominent example is the file system, which could be used by L4 applications to store and retrieve data. In L4Linux applications can communicate directly with the L4 environment. Such applications are called *hybrid tasks*.

Contrary to hybrid tasks, Linux applications in the VM cannot directly communicate with the outside. Instead, the VMM would have to act as a server. L4 applications would request service using IPC with the VMM. The VMM would implement a stub driver, which would react on incoming client requests. The request command would be written to a shared memory location, and the guest would be notified with an interrupt. The guest's stub driver would read the command from the shared memory region and forward it to the appropriate service application. This forwarding could be done with a special device file, for example. The application would answer the request by writing to the device file. The stub driver would receive the answer, write it into the shared memory region and inform the VMM, which would then send the reply IPC.

8.2 Conclusion

With this work, I devised a novel way to execute Linux applications on top of the microkernel Fiasco. All Linux applications that do not directly access system devices are supported without modification. I also succeeded in porting KVM, and can therefore also run all OSes that are supported by KVM. For example, the KVM port runs Windows, *BSD and others. With these OSes, all their applications become usable, which significantly enhances the range of available software.

In this work Linux runs encapsulated and does not increase the TCB of L4 applications. The only exception to the encapsulation is hard disk access, which uses DMA. With the VM driving DMA enabled devices, it must be counted to the TCB. I presented two solutions to handle this problem, and remove the VMM from the TCB. The solutions require device commands to be handed out of the VM, and and to be analyzed by an external component. I measured the overhead of these solutions to be small. In the near future all shipped computers will have IO MMUs, which allows DMA accesses to be restricted, and thus resolves the problem entirely.

My solution has a small resource footprint, of about 284KB for an individual VMM instance. However, as with any virtualization solution, additional resources are needed for device functionality and main memory for the guest.

I measured the performance of my solution to be better than L4Linux and KVM in all benchmarks. The performance overhead is small for all but pathological cases. My solution supports VMs with multiple CPUs. In my measurements SMP scaled well.

The complexity of the solution is low, with the VMM comprising about 3900 SLOC and the required kernel patch consisting of only about 2700 SLOC modifications. The kernel patch could be ported to a new kernel version in about one hour, which indicates that the maintenance costs are low. With the tiny VMM implementation, chances are high that the work can be forked and reused by researchers and developers for their own projects.

My solution is only applicable on machines that support hardware assisted virtualization. However, I expect that off the shelf desktop computers will be equipped with processors with hardware virtualization support in the near future. Today, the situation is different in the embedded domain. Hardware virtualization is not available on ARM and thus we have to resort to L4Linux for those machines.

Glossary

ABI	Application Binary Interface	20
ACL	Access Control List, a mechanism for access control, which is employed in standard OSes.	2
ACPI	Advanced Configuration and Power Interface, a standard for a common interface for device and power settings.	52
AHCI	Advanced Host Controller Interface, an interface for Serial Advanced Technology Attachment (SATA) controllers.	61
APIC	Advanced Programmable Interrupt Controller	39
ASID	Address Space Identifier	66
BIOS	Basic Input-Output System	38
DMA	Direct-Memory Access	38
GDT	Segment Descriptor Table	47
GPL	Gnu General Public License	19
Guest	All code running inside a virtual machine	11
Host	Platform hosting a virtual machine	11
Hypervisor	The privileged part of a virtual-machine monitor. If both VMM and hypervisor run in privileged mode, the combination is also called hypervisor.	11
IC	Interrupt Controller	37
IDE	Integrated Drive Electronics, a standard for a hard disk interface.	61
IPC	Inter-Process Communication	6
MMU	Memory Management Unit	12
NIC	Network Interface Card	16
NTP	Network Time Protocol	58
OS	Operating System	1

PCI	Peripheral Component Interconnect, a standard for a bus system, which is deployed in nearly every PC.	52
RT	Real-time	3
SLOC	Source Lines Of Code	6
SMP	Symmetric Multi-Processing, a term often used for machines with multiple CPUs.	39
SVM	Secure Virtual Machine, a processor extension employed in modern AMD processors to aid virtualization.	35
TCB	Trusted Computing Base	5
TLB	Translation Lookaside Buffer	66
UTCB	User Thread Control Block, a part of the thread control block, which is accessible in user land from within in the thread's address space	77
VESA	Video Electronics Standards Association, a standard for a graphics card interface.	54
VGA	Video Graphics Array, a standard for a graphics card interface.	61
VM	Virtual Machine	10
VMCB	Virtual Machine Control Block, a data structure containing both the virtual CPU's state (state save area) and settings to control its execution (control area).	43
VMM	Virtual-Machine Monitor	10

Bibliography

[ABB+86] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A New Kernel Foundation for UNIX Development. pages 93–112, 1986. 7

[AH] A. Au and G. Heiser. L4 user manual. *School of Computer Science*. 7

[AMD07] I. AMD. O Virtualization Technology (IOMMU) Specification. *AMD Corporation*, 2007. 16

[BDF+03] 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. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003. 16, 24

[Bie06] Sebastian Biemueller. Hardware-supported virtualization for the l4 microkernel. Diploma thesis, System Architecture Group, University of Karlsruhe, Germany, September 29 2006. 28

[CN01] P.M. Chen and B.D. Noble. When virtual is better than real. In *Proceedings of the 2001 Workshop on Hot Topics in Operating Systems (HotOS)*, pages 133–138, 2001. 10

[Cor] Microsoft Corporation. Microsoft security bulletin ms09-050. `http://www.microsoft.com/germany/technet/sicherheit/bulletins/ms09-050.mspx`. Online, accessed 03-01-2010. 5

[Dik00] J. Dike. A user-mode port of the Linux kernel. In *Proceedings of the 2000 Linux Showcase and Conference*, volume 2, 2000. 22

[Dik01] J. Dike. User-mode Linux. In *Proceedings of the 5th annual Linux Showcase & Conference-Volume 5*, pages 2–2. USENIX Association Berkeley, CA, USA, 2001. 22, 23

[Ger] P. Gerum. Xenomai-Implementing a RTOS emulation framework on GNU/Linux. Online, accessed 15-12-2009. 70

[GGP06] A. Ganapathi, V. Ganapathi, and D. Patterson. Windows XP kernel crash analysis. In *Proceedings of the 20th conference on Large Installation System Administration table of contents*, pages 12–12. USENIX Association Berkeley, CA, USA, 2006. 5

[Har88] N. Hardy. The confused deputy:(or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 22(4):36–38, 1988. 9

[Här02] H. Härtig. Security architectures revisited. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, page 23. ACM, 2002. 17

[Hei05] Gernot Heiser. Secure embedded systems need microkernels. *USENIX ;login:*, 30(6):9–13, Dec 2005. 6

[HHL+97] H. Härtig, M. Hohmuth, J. Liedtke, S. Schonberg, and J. Wolter. The performance of μ-kernel-based systems. In *16th ACM Symposium on Operating System Principles (SOSP)*, pages 66–77, 1997. 8, 20

[HUL06] G. Heiser, V. Uhlig, and J. LeVasseur. Are virtual-machine monitors microkernels done right? *ACM SIGOPS Operating Systems Review*, 40(1):95–99, 2006. 16

[HWF+] S. Hand, A. Warfield, K. Fraser, E. Kotsovinos, and D. Magenheimer. Are Virtual Machine Monitors Microkernels Done Right? 16

[Int] Intel. Intel Multiprocessor Specification. `http://www.intel.com/design/pentium/datashts/242016.HTM`. Online, accessed 14-10-2009. 48

[Lac04] Adam Lackorzyński. L4Linux Porting Optimizations. Diploma thesis, University of Technology Dresden, 2004. 21

[Lie94] Jochen Liedtke. Improving IPC by kernel design. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 175–188. ACM New York, NY, USA, 1994. 7, 19

[Lie95] Jochen Liedtke. On-microkernel construction. In *15th ACM Symposium on Operating System Principles (SOSP)*. Citeseer, 1995. 7, 19

[Lie96] Jochen Liedtke. Toward real microkernels. *Commun. ACM*, 39(9):70–77, 1996. 7

[Lim] ARM Limited. ARM Security Technology Building a Secure System using TrustZone® Technology. `http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf`. Online, accessed 21-01-2010. 46

[LUY+08] J. LeVasseur, V. Uhlig, Y. Yang, M. Chapman, P. Chubb, B. Leslie, and G. Heiser. Pre-virtualization: soft layering for virtual machines. In *Computer Systems Architecture Conference, 2008. ACSAC 2008. 13th Asia-Pacific*, pages 1–9, 2008. 24

[LvSH05] B. Leslie, C. van Schaik, and G. Heiser. Wombat: A portable user-mode Linux for embedded systems. In *Proceedings of the 6th Linux. Conf. Au, Canberra*, 2005. 22

[MD98] Y.K. Malaiya and J. Denton. Estimating defect density using test coverage. *Rapport Technique CS-98-104, Colorado State University*, 1998. 6

[MI03] M.S. Miller and C. Inc. Capability myths demolished. 2003. 9

[MMH08] D.G. Murray, G. Milos, and S. Hand. Improving Xen security through disaggregation. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS international conference on Virtual Execution Environments*, pages 151–160. ACM New York, NY, USA, 2008. 25

[pag] How retiring segmentation in AMD64 long mode broke VMware. `http://www.pagetable.com/?p=25`. Online, accessed 23-09-2009. 13

[PG74] Gerald J Popek and Robert P Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, 1974. 10, 11

[pis] L4Ka::Pistachio microkernel. `http://l4ka.org/projects/pistachio/`. Online, accessed 22-09-2009. 20

[PSLW09] Michael Peter, Henning Schild, Adam Lackorzynski, and Alexander Warg. Virtual machines jailed: virtualization in systems with small trusted computing bases. In *VDTS '09: Proceedings of the 1st EuroSys Workshop on Virtualization Technology for Dependable Systems*, pages 18–23, New York, NY, USA, 2009. ACM. 6, 7, 28, 33, 34, 39

[RI00] J.S. Robin and C.E. Irvine. Analysis of the Intel Pentium's ability to support a secure virtual machine monitor. In *Proceedings of the 9th conference on USENIX Security Symposium-Volume 9*, pages 10–10. USENIX Association Berkeley, CA, USA, 2000. 12

[Rus] Rusty Russell. Lguest: The Simple x86 Hypervisor. `http://lguest.ozlabs.org/`. Online, accessed 11-01-2010. 26

[Rus08] R. Russell. virtio: towards a de-facto standard for virtual I/O devices. 2008. 37

[ska] skas mode. `http://user-mode-linux.sourceforge.net/old/skas.html`. Online, accessed 23-09-2009. 22, 23

[Ste] Udo Steinberg. NOVA Microhypervisor. `http://os.inf.tu-dresden.de/~us15/nova/`. Online, accessed 15-01-2010. 26

[Ste09] Julian Stecklina. Remote Debugging via Firewire. Diploma thesis, University of Technology Dresden, 2009. 78

[SVL01] J. Sugerman, G. Venkitachalam, and B.H. Lim. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *USENIX Annual Technical Conference*, pages 1–14, 2001. 29

[sys] User-Mode-Linux SYSEMU Patches. `http://sysemu.sourceforge.net/`. Online, accessed 23-09-2009. 23

[Vog09] Dirk Vogt. L4ReAnimator - A restarting framework for L4Re. Diploma thesis, University of Technology Dresden, 2009. 78

[WB07] N.H. Walfield and M. Brinkmann. A critique of the GNU hurd multi-server operating system. 2007. 20