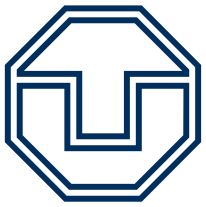

TECHNISCHE
UNIVERSITÄT
DRESDEN

Undergraduate Thesis

A Device Virtualization Framework for L4

Mario Schwalbe

May 28, 2009

Technische Universität Dresden
Department of Computer Science
Institute for System Architecture
Operating Systems Group

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

Aufgabenstellung

Im Rahmen der Arbeit soll eine Infrastruktur für die Gerätevirtualisierung unter L4 auf der x86 Architektur entworfen und am Beispiel eines PCI Busses implementiert werden. Aufsetzend darauf soll ein Gerät eigener Wahl implementiert werden. Als Implementierungsumgebung soll L^{4}LINUX dienen.

Die Implementierung des PCI Busses soll gestatten, daß dieser frei konfigurierbar ist, so daß virtualisierte Geräte beliebig angeordnet werden können. Außerdem soll es möglich sein, daß physische Geräte im Hostsystem durch den virtualisierten PCI Bus an das Gastsystem durchgereicht werden können. Dies soll ebenfalls mit einem Gerät eigener Wahl demonstriert werden.

Erklärung

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

Dresden, den 28. Mai 2009

Mario Schwalbe

I love deadlines. I like the whooshing sound they make as they go by.

(Douglas Adams)

Acknowledgements

First, I would like to thank Prof. Hermann Härtig for the opportunity to work on this project in the operating systems research group at TU-Dresden. My special thanks goes to my supervisor Adam Lackorzynski — who fixes bugs faster than his shadow — for providing me with many insights about the internals of TUD:OS and for his suggestions on how to integrate my work. Last but not least, I would like to thank everybody, who supported my work, in particular Björn Döbel, Björn Thalheim, Eva Wagner, and Stefan Schulz, who found the time to proofread, giving valuable feedback that helped me to improve this thesis for the worse.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Research at the operating systems group at TU-Dresden focuses on micro-kernel-based real-time and secure systems that use the common L4 micro-kernel interface as well as virtualization. The TU-Dresden Operating System (TUD:OS) is based on the FIASCO kernel [3], which is a small, fast, and fully preemptible second-generation micro-kernel written in C++.

With respect to system security, the operating systems group devised the NIZZA system architecture [32]. The core concept inherent to NIZZA is the fine-grained isolation of protection domains. Isolation allows to reduce the trusted computing base (TCB) by using untrusted components via tunneling through trusted wrappers. Hermann Härtig defined tunneling as a technique to use software that by itself does not provide a required property and adding this property in an additional layer [30]. As a well-known example, he highlighted the use of an insecure protocol for secure communication by encrypting packets before handing them over, which is feasible unless denial of service is a concern. Fine-grained isolation, furthermore, allows independent and more efficient evaluation of those components due to their reduced complexity [23].

Based on the NIZZA system architecture, the Secure Inter-Network Architecture (μSINA) for processing classified information used in its initial design two instances of L^4LINUX, a paravirtualized version of Linux running on top of TUD:OS, for network communication [24]. Due to this design decision the system benefits from a robust and stable network stack implementation instead of spending development effort in rewriting without compromising the overall system integrity. These two instances needed access to one network adapter, which was achieved by using different adapters driven by different drivers, only feasible in this fairly restricted setup. If a device virtualization solution had been available, such a system could have been configured more flexible and reliable.

In addition, using a virtual machine to provide unmodified legacy operating systems with an anticipated environment by multiplexing physical devices on top of a virtualization-enabled

micro-kernel, as proposed in [28], has recently been proven to work [14]. Such a solution is beyond the scope of this thesis, but a device virtualization framework may be a foundation for future development in this area.

1.1 Goals

The work, which is presented in this thesis, is L4VMM, a framework to develop virtual devices, in particular PCI devices that can transparently be used next to host devices, with the following characteristics:

Flexibility The framework shall not be limited to specific devices or classes of devices but instead support the whole range of devices currently on the market. With respect to system architecture, it shall be modularized, extendable, and portable to fit actual as well as future needs.

Ease of Use The framework shall obey already deployed TUD:OS interfaces allowing integration without modifications. With respect to extensibility, it shall aid in developing new emulated devices.

Performance Achieving these goals shall not impair the virtualized system's performance without good reason to do so, in particular in comparison to physical hardware.

1.2 Outline

The remainder of this thesis is organized as follows:

Chapter 2 In chapter 2, I will give an introduction to the fundamentals of computer systems, focusing on operating systems. Thereafter, I will discuss general concepts of hardware virtualization paradigms currently used, ranging from the original concept of pure virtualization to recent software- and hardware-based optimization techniques, and introduce different system architectures, recent solutions are comprised.

Chapter 3 Following this, I will give an overview of important TUD:OS components, which constitute the environment of this thesis, and shortly review wide-spread open- and closed-source virtualization solutions.

Chapter 4 In this chapter, I will analyze the requirements needed for efficient device virtualization, explain my approach, and outline how it works.

Chapter 5 In chapter 5, I will explain the final structure, how virtual devices are implemented, and give insights to key aspects of physical devices, such as interrupt delivery and direct memory access. At the end of the chapter, I will highlight the changes to existing L4 components as well as limitations compared to real hardware.

Chapter 6 In this chapter, I will discuss the necessary steps to use the results of this thesis practically: how to incorporate support into applications and how to develop virtual devices.

Chapter 7 I will evaluate the performance of my approach in chapter 7 using microbenchmarks and L^{4}LINUX reading a virtual disk.

Chapter 8 Finally, chapter 8 sums up my thesis and outlines suggestions for future work.

1.3 Terminology

This document is concerned with the emulation of hardware components in user mode under L4. In order to not confuse the reader, the meaning of several terms used in this document shall be defined at first: A **virtual device** is a device, whose functionality is entirely emulated in user mode. It does not need to be physically available. In contrast, a **physical device** is a device that does exist on the machine, to which a driver shall gain access. If not stated otherwise, the term **memory** refers to the physical memory space.

Chapter 2

Fundamentals

This chapter gives an introduction to how computers work and how the components interact with each other from the operating system's point of view. Afterwards, I will introduce general concepts of hardware virtualization.

2.1 Physical Machines

Figure 2.1 shows the core components a modern Intel x86 computer system consists of[1].

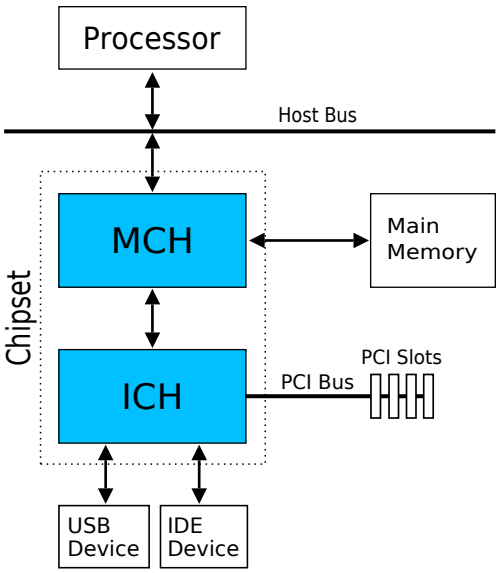

Figure 2.1: Chipset diagram

[1] Recent AMD systems and Intel's next generation incorporate the memory controller into the processor. Nevertheless, to get a basic understanding, I will explain the older system architecture.

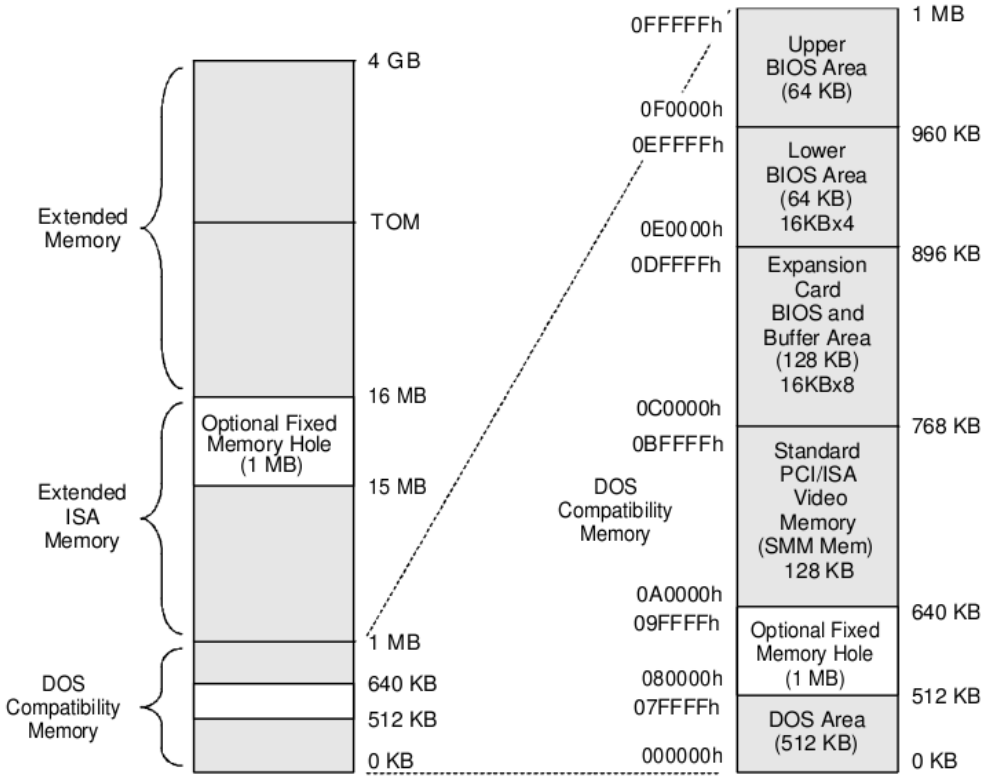

Figure 2.2: The x86 architecture's physical memory map.

The **processor** (**CPU**) executes all instructions, including the operating system itself. The **I/O Controller Hub** (**ICH**)[2] provides extensive I/O support by incorporating a variety of devices and functions, which used to be chips on its own, along with the ability to connect other I/O components to the system. The **Memory Controller Hub** (**MCH**)[3] in between connects the main memory with processor and ICH, allowing the depicted directions of data transfer, which in turn results in the physical memory address space, as shown in figure 2.2 (which was originally published in [38]). The MCH, therefore, forwards the address and data lines, driven by the processor, to either main memory or the ICH. In the latter case, main memory at the same location is not accessible anymore. Furthermore, devices can, via the ICH, directly access main memory. This mechanism is called **direct memory access** (**DMA**).

In addition, the x86 architecture offers a special physical address space, distinct from the normal memory address space, the **I/O space**, which obeys the load/store concept. That is, it

[2] Also called 'Southbridge'.
[3] Also called 'Northbridge'.

must be read and written through special instructions. In contrast to memory space, no other instructions, such as arithmetical operations, are allowed to have I/O space operands.

In general, all devices are driven through **registers**, which can be grouped together in **register blocks**. Read and write access to these registers may have effects beyond data deposition and retrieval. Registers can be located in either physical memory space or, on the x86 architecture, I/O space.

2.1.1 Device Discovery

In a computer system, devices are connected through bus devices. Because recent computer systems' standards usually do not describe the whole system, operating systems need to discover which devices are actually present and which resources they use.

Probing

The simplest solution is to guess that a device is present by trying to access it. If it does not respond, it is assumed to not be available. This assumption is most adequate for legacy devices, whose resources are at fixed locations, defined in the standards. However, modern devices, such as PCI devices, may map their resources to arbitrary locations, allowing greater flexibility and thereby reducing resource conflicts.

Plug and Play

The **Plug and Play** (**PNP**) standard [16] defines a software interface to transfer device descriptions by sequentially reading one single well-known register. These descriptions contain IDs to identify and differentiate devices and their corresponding resources. However, the PNP standard was never widely adopted.

Advanced Configuration and Power Interface

The **Advanced Configuration and Power Interface** (**ACPI**) standard [25] was developed to overcome the previous standard's shortcomings. Besides providing a configuration interface, including power management, it also defines means to enumerate and discover all devices in a computer system. The information is stored in static tables in main memory. The first table is

stored in the high BIOS area, containing the remaining tables' addresses. For device discovery the most relevant table, the **Differentiated System Description Table** (**DSDT**) contains a device tree in **ACPI Machine Language** (**AML**) format.

2.1.2 PCI Devices

Devices attached to the PCI bus are referred to as **PCI devices**. The PCI bus is connected to the system through a host-to-PCI bridge, which resides in the MCH. PCI devices may be integrated in the ICH or, in case of add-in cards, connected to it. These devices can have several types of resources, which shall be described in more detail in the following sections.

PCI Configuration Space

All PCI devices, except host bus bridges, have to implement a **configuration space**[4]. This space is divided into a predefined header region defined by the PCI specification [58], which consists of registers that uniquely identify the device and allow the device to be generically controlled (e.g., completely disabled), and a vendor-specific device dependent region. Devices may implement only necessary and relevant registers, treating other specified registers as reserved. A device's configuration space has to be accessible at all times.

Because system software needs to scan the PCI bus to determine which devices are present, the PCI bridge should return a value of all 1's on read access to nonexistent devices. This value, when querying the device's vendor and device ID, is invalid and signifies that there is no device present. Read access to reserved or unimplemented registers must return 0. Likewise, write access should be discarded. All multi-byte registers follow little endian ordering, irrespective of the processor's preferred byte order.

Unlike memory and I/O space, the PCI configuration space is not directly accessible by the processor. Instead the PCI configuration space is mapped to the processor's address spaces. On the x86 architecture the following mechanisms exist:

- The PCI specification [58] defines a mapping mechanism that has to be implemented within the MCH, which uses the `CONFIG_ADDRESS` 32-bit register (at I/O address `0CF8h` through `0CFBh` in I/O space) and `CONFIG_DATA` register (at I/O address `0CFCh` through `0CFFh` in I/O space). To reference a PCI configuration space register, a 32-bit value is written to `CONFIG_ADDRESS`, as shown in figure 2.3 on the facing page (taken from [46]) that specifies the PCI bus, the device on that bus, the function within

[4] In fact, multifunction devices have to provide a configuration space for each function implemented.

the device, and a specific configuration register of the device function being accessed. CONFIG_DATA then becomes a window into the four bytes of configuration space specified by the contents of CONFIG_ADDRESS. The MCH is responsible for translating and routing the processor's I/O space access to the CONFIG_ADDRESS and CONFIG_DATA registers to internal MCH configuration registers or other components in the system, such as PCI devices.

- The PCI Express specification [57] defines a flat mapping mechanism to memory space for both PCI and PCI Express configuration spaces. The base address is specified in the PCIEXBAR register, which itself resides in the host bridge's configuration space. Therefore, it has to be programmed using the I/O space based protocol described in the preceding item.

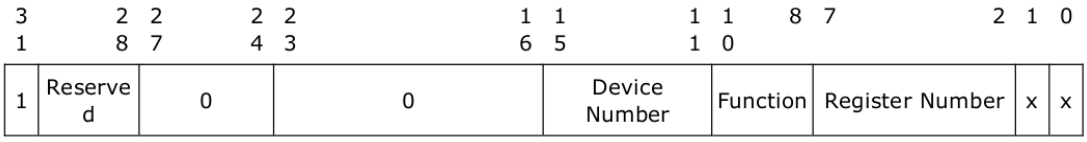

Figure 2.3: Host-to-PCI bridge CONFIG_ADDRESS data word layout.

From the operating system's perspective, the x86 architecture's BIOS provides another possibility: System software may use BIOS service functions, according to the BIOS32 specification [59] to access a device's configuration space.

I/O Regions

One of the most important functions for enabling superior configurability and ease of use is the ability to relocate PCI devices' resources, called **I/O regions**, in the address spaces. At system power-up, the operating system has to be able to determine which devices are present, build a consistent address map, and determine if a device has an expansion ROM. Thus, it has to determine how much memory is in the system and how much address space the I/O controllers in the system require. After determining this information, the operating system can map the I/O controllers into reasonable locations and proceed with system boot. In order to do this mapping in a device independent manner, the address registers for this mapping are placed in the predefined header portion of the PCI configuration space. I/O space or memory space regions can be mapped to arbitrary locations. However, its location has to be naturally-aligned.

Bus Master Direct Memory Access

PCI devices, capable of driving the PCI bus as a master, may transfer data directly to or from main memory (**BM-DMA**), thereby offloading the processor to improve system performance in multitasking environments. BM-DMA transfers do not require the processor to execute instructions other than to program the device. For this reason these transfers result in less cache trashing. In addition the chipset may use burst accesses to transfer larger chunks of data at once. Because devices are not subject to the processor's memory management unit, system software has to ensure to provide physical addresses when programming the BM-DMA operation.

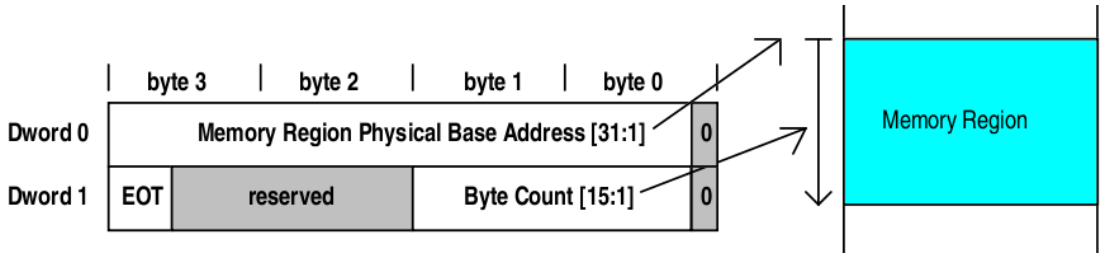

Figure 2.4: Physical Region Descriptor to specify BM-DMA memory for IDE devices.

For instance, the bus master IDE specification [37] mandates a **Physical Region Descriptor Table**. This table comprises consecutive **Physical Region Descriptor**s (**PRD**s) in main memory as shown in figure 2.4. A set EOT bit signifies the last table entry. The table's physical base address has to be programmed through a device specific register.

2.1.3 Interrupt Requests

Interrupts are events that signal that a condition exists somewhere in the system, the processor, or within the currently executing task that requires the attention of the operating system. They typically result in a forced transfer of execution from the currently running task to a special software routine, provided by the operating system's kernel, which is called an interrupt handler. When execution of the handler is complete, the processor resumes execution of the interrupted task, unless another task shall be scheduled. From the interrupted task's point of view, this resumption, being either immediately or later, happens without loss of program continuity [42].

On the x86 architecture, software can generate interrupts by executing the **INT** instruction. However, because system hardware is designed to operate asynchronously without processor

interaction, it uses interrupts to handle events external to the processor, such as requests to service peripheral devices.

Legacy devices usually assert fixed well-known interrupts. PCI devices allow greater flexibility. Therefore, the PCI bus provides four **interrupt lines**, which are routed by a **PCI Interrupt Router** (**PIRQ**) to the system's **Interrupt Requests** (**IRQs**). To obtain the information which IRQ a device actually asserts, system software may probe the following, in order of preference:

1. Information about IRQ routing may be stored in the ACPI Differentiated System Description Table [25].
2. Microsoft proposed for Windows 95 a mechanism to describe the IRQ routing in a table in BIOS ROM at physical memory address `F0000h` [53].
3. The PCI specification defines a register in the PCI configuration space to communicate a device's IRQ. Usually this register has to be programmed by system software, for example on the x86 architecture the BIOS, during initialization. The device itself cannot determine its IRQ [58].

2.2 Virtual Machines

In 1974 Robert P. Goldberg defined **virtualization** as the process of '[...] transform[ing] the single machine interface into the illusion of many. Each of these interfaces (**virtual machine**s) is an efficient replica of the original computer system, complete with all of the processor instructions (i.e., both privileged and non-privileged instructions) and system resources (i.e., memory and I/O devices).' [20]

The concept of virtual machines was invented by IBM in the 1960's as a method of time-sharing their extremely expensive mainframe hardware. These virtual machines detached operating systems from the physical hardware allowing them to run concurrently and isolated from each other on a single **host machine**. In order to achieve the desired complete isolation, the **guest operating system**s did not drive hardware directly anymore. As direct hardware access is desirable from the performance point of view, later architectures such as the zSeries Channel architecture provided devices that could be controlled directly by untrusted guests.

In general, there are two approaches conceivable: emulation (or simulation) and virtualization, which evolved to various forms over years. These concepts shall be introduced in the following sections.

2.2.1 Emulation

The **emulation** approach models the complete hardware, including the processor, entirely in software, which itself runs on its host operating system. The guest operating system's instructions are interpreted by applying their effects to the system model. These emulators, therefore, suffer from poor performance. To solve this problem, it is nowadays quite common to perform a runtime conversion of the target processor's instructions into host instruction sequences that modify the model using a **dynamic translator**. The resulting binary code can be stored in a translation cache allowing reuse of already decoded sequences. The advantage over an interpreter is that the target instructions are fetched and decoded only once. Nevertheless, according to Fabrice Bellard dynamic translation was still measured to be about ten times slower in comparison to native execution [12]. Thus, this approach is most suitable for older systems. On the other hand, the guest system is not limited to the host architecture.

2.2.2 Pure Virtualization

Because the main goal most often is not the emulation of arbitrary architectures, but instead to provide the operating system with an environment similar to the host machine (i.e., the preceding definition in its stricter sense), it is quite obvious to run its instructions directly on the host processor. Doing so has the potential for speedups. However, it has to be ensured that the execution does not interfere with the virtualization by exposing information that should not be visible to the guest.

Fortunately, recent processors distinguish between (at least) two modes of operation: A kernel mode, where all available instructions, including privileged ones to modify crucial system resources, are allowed to be executed, which is usually used by the operating system kernel, and a user mode for applications, where only a reduced set of instructions is allowed. When encountering a privileged instruction executed with insufficient privileges, the processor stops execution and raises an exception that is handled by the operating system kernel. This mechanism can be used by introducing a software layer called a **virtual machine monitor** (**VMM**) that takes complete control of the machine hardware and runs the guest operating system in user mode. Upon fault, the VMM inspects the original instruction causing the exception, checks its validity and, if valid, emulates its effect, including the virtual machine's devices, before continuing with normal execution. Other processor architectures, such as x86, that have more than two privilege levels allow the VMM to deprivilege the guest kernels to an intermediate privilege level [9].

Pure virtualization is a clean concept: It does not require modification of the guest operating system, thereby minimizing engineering effort and permitting virtualization of systems whose source code is not available. However, this approach has two major drawbacks:

- Popek and Goldberg proved that a computer architecture is virtualizable if all virtualization-sensitive instructions are privileged and trap into the VMM when executed from any but the most privileged mode [60]. The x86 architectures' processors were not designed with virtualization in mind, and, thus, its user mode does not raise exceptions for all instructions affecting privileged state. Instead, the state silently does not change [63].

  Recent virtualization solutions, therefore, modify the guest code replacing non-privileged but virtualization-sensitive instructions at load time with calls to the VMM. An improved form of this binary translation, adaptive binary translation, tries to detect and translate virtualization-sensitive but non-privileged memory operations during execution by identifying data locations that are frequently involved in traps [7].

  In addition x86 processor vendors attended to this problem incorporating virtualization support into recent processor generations, namely Intel's **Virtual Machine Extension** (**VMX**) [43] and AMD's **Secure Virtual Machine** (**SVM**) [8] at the cost of more complex hardware. A new processor mode permits virtualization of unmodified operating systems, but performance may decrease due to a lack of optimization [56].

- Pure virtualization imposes a reasonable performance penalty, due to frequent trapping, since a VMM has to authorize all virtualization-sensitive operations within the virtual machine [52, 69]. In particular, switches between protection domains take significant time even on modern processors. As the guest operating system is not aware of being virtualized, it does not avoid these operations.

2.2.3 Paravirtualization

As the guest operating system is not aware of being virtualized, it is very likely to cause a lot of faults. This incurs a considerable overhead, which solely comes from the requirement to model the same interface as provided by real hardware. Thus, it would be beneficial to replace virtualization-sensitive parts of the guest operating system with a high-level software interface to the VMM, avoiding this additional overhead[5] [67]. This is the key idea of **paravirtualization**. As paravirtualization is based on the idea of modifying the guest operating system to make it virtualization-friendly, it is only applicable to systems whose source code is available. This

[5] This interface can also be considered as a special hardware architecture.

approach permits higher performance at the cost of less flexibility: A modified operating system usually does not run on real hardware anymore.

Progressive Paravirtualization

Progressive paravirtualization [18] combines both approaches: It takes advantage of virtualizable processor architectures and selectively adds paravirtualization extensions to a previously unmodified guest operating system. These paravirtualization extensions can, for example, optimize I/O and MMU operation, avoid idle time, or support device hot-plugging, even devices that usually do not change in a real system such as the processor itself.

2.2.4 System Architecture

The preceding investigation only considers the interface between virtual machine monitor and guest operating system with decreasing complexity. With respect to system architecture, there exist two additional approaches: native virtual machine monitors and hosted virtual machine monitors.

Native Virtual Machine Monitor

A **native virtual machine monitor** (or **hypervisor**[6]) obeys the traditional design invented by IBM running directly on the machine whose resources are to be partitioned and multiplexed among several guest operating systems. It is designed to serve operating systems and does not allow to run normal applications directly. Thus, recent solutions on the x86 architecture only perform basic scheduling and memory management, typically delegating management functions, such as creating new virtual machines and driving devices, to a special privileged guest. As a native VMM is a specialized system for its own purposes that provides no further abstractions, it can only be a platform for full operating systems, but does not offer a system with its own applications supporting properties like micro-kernels.

[6] The term hypervisor emphasizes that it has the highest privilege level whereas VMM denotes its functionality giving the guest software the illusion of running on a physical machine.

Hosted Virtual Machine Monitor

A **hosted virtual machine monitor** uses a driver loaded into a host operating system to establish a privileged virtual machine monitor that also runs directly on the hardware. From then on, the processor is executing either the host operating system's or the VMM's instructions, with this driver facilitating the transfer of control between the two protection domains. Such a switch involves saving and restoring all user and system visible state of the processor, and is, thus, more heavyweight than a normal process switch [65].

Like a native VMM, the hosted VMM has full system and hardware privileges. But it behaves cooperatively and allows the host operating system to schedule it. The host operating system can also page out the memory allocated to a particular virtual machine except for a small set of pages that the VMM has pinned on behalf of the virtual machine. This allows the VMM to be treated by the host operating system like a regular application, but occasionally at the expense of performance if the host operating system makes poor resource scheduling choices for the virtual machine.

The primary feature of this design is that it takes advantage of a pre-existing operating system for I/O device support to run on whole classes of hardware without needing special device drivers for each possible device, while still achieving near native performance for processor-intensive workloads. However, the most significant trade-off of a hosted architecture is in potential I/O performance degradation. Because I/O emulation is done in the host operating system, a virtual machine executing an I/O intensive workload can accrue extra processor time switching between their protection domains.

Chapter 3

State Of The Art

3.1 TUD:OS

The TU-Dresden Operating System (TUD:OS) [2] project focuses on research in micro-kernel and virtualization technology applied to systems security and real-time systems. Real-time applications run side by side with non-real-time ones without violating the real-time properties promised [10, 21]. Real-time components for example may be real-time managers for disks, communication and windowing systems [31]. Other components are built on top of these services, including filesystem servers or video players.

In systems security, this project designed the Nizza architecture that supports legacy software together with applications with high security requirements [32] as well as μSINA, a flexible and scalable architecture for secure communication [24].

3.1.1 FIASCO

FIASCO [3, 27] is a micro-kernel implementing the L4 specification [49]. The L4 API is designed to be minimal and provides only simple functions to user-level applications [50]. FIASCO provides threads and address spaces as basic means of separation and fast synchronous inter-process communication (IPC) for well-defined interaction [48]. Currently it supports several architectures (x86, AMD64, and ARM). A port that runs in Linux user mode is also available [64]. Threads usually run in unprivileged user mode, but can have I/O privileges using the x86 architecture's I/O bitmaps. On this architecture, access to I/O registers can be controlled on a per-register basis. FIASCO also provides IRQ support, delivered through IPC. Sharing IRQs is not implemented but a thread may attach to several IRQs.

3.1.2 L4ENV

The L4ENV [1] is an integral part of TUD:OS that aims at providing a uniform interface to the different L4 micro-kernel implementations making programs written for it independent from the underlying micro-kernel. If they do not contain any hardware specific code, these applications run on any platform. The functionality provided by L4ENV is implemented in server tasks and libraries linked to the applications. L4ENV includes a physical memory management server, a memory mapping library, a thread management library, a semaphore and lock library, a naming service, an I/O server, an application loader and execution system, a console and windowing system for output, an input library, and network support. Ongoing projects aim at implementing disk storage and USB support.

3.1.3 OMEGA0

The OMEGA0 server is a server based implementation of the OMEGA0 interface [51], which centralizes all interrupt-logic handling. The OMEGA0 interface hides all hardware dependencies and the micro-kernel's interrupt interface, isolating driver components from future changes to this interface. Interrupts are delivered to drivers through synchronous IPC messages. In a micro-kernel-based system, these drivers run in different address spaces, isolated from each other. To receive an IPC message, a thread inside the driver task needs to wait for a specific interrupt by invoking the appropriate IPC call. To support the x86 architecture's poor interrupt design, where several devices may assert the same interrupt, OMEGA0 allows to share interrupts among different drivers, waking up each of them on interrupt assertions.

3.1.4 L4IO

L4IO [22] is the system-wide I/O synchronization server for TUD:OS. L4IO provides resource management, PCI operations, and interrupt handling. After startup L4IO locates all PCI I/O regions in their address spaces. Drivers may then enumerate all PCI devices by vendor and device id, by PCI class, or by slot. The returned data structure contains the I/O regions' base addresses, their sizes, the device's IRQ, and a handle for further communication. Using that handle drivers may invoke all PCI operations, including PCI configuration space access. The current implementation allows only PCI devices to be queried this way. Other devices are not recognized as devices by L4IO. Instead it relies on drivers requesting resources for their device. A device's resources are handed out to drivers on request if they are not already used by another driver. Thereby L4IO enforces a weak policy that any two drivers may never use

the same resource at a time to avoid hardly detectable errors due to incorrect hardware access. To support the devices' interrupts L4IO incorporates an OMEGA0 server as replacement for a standalone OMEGA0.

3.1.5 L4LINUX

L4LINUX [4, 11, 15, 26] is a paravirtualized Linux [5] kernel that is running in user mode on top of L4ENV allowing execution of unmodified Linux programs besides other native L4 applications. With respect to device access this approach imposes some restrictions.

If L4LINUX is not allowed to access hardware directly, a set of stub drivers, which on one hand provide the necessary in-kernel API, while on the other hand use the relevant L4 servers, implement basic functionality, like a console with input or networking. In fact, L4LINUX may use the underlying system services in the same way as other L4 applications. For example, if a network server drives the network card of the system, L4LINUX has to use this L4 server for network communication, because only one program can drive one particular hardware device at a time.

If direct hardware access is desired, L4LINUX has to gain access to the host-to-PCI bridge resources to enumerate and configure the PCI devices. These resources are initially owned by roottask and only handed out to one task in the L4 system. Thus, this approach interferes with other tasks that need these resources, in particular L4IO. Additionally, it is difficult or in a varying setup even impossible to configure access to dynamically located PCI devices' resources in advance. If raising the x86 architecture's I/O privilege level, special measures must be taken to prevent L4LINUX from accidentally driving the same device as another L4LINUX or L4 driver server. Because the original Linux kernel is supposed to run on hardware directly, L4LINUX tries to initialize and drive all devices it finds. The only workaround, currently existing, is to disable the drivers for devices L4LINUX should exclude, which only allows to enable or disable a fairly rough set of drivers as one driver is typically responsible for several devices produced by the same vendor.

As Adam Lackorzynski stated in his diploma thesis [47], a better approach is to provide virtual hardware, the L4LINUX server can drive, like running on its own machine, where only the desired devices are visible at all. L4VMM is such a solution that incorporates both approaches. Virtual devices can be implemented using the underlying system services that multiplex the host's devices, while direct access to selected physical devices is still possible.

3.2 Other Virtualization Solutions

Over years a lot of open- and closed-source virtualization solutions evolved. I will introduce a few of them in the following sections. Their design principles are explained in Section 2.2 on page 11.

3.2.1 Hosted Virtual Machine Monitors

QEMU

QEMU [12, 13] is a machine emulator that can run on several host operating systems such as Linux, Windows and Mac OS X as well as on several host architectures (x86, PowerPC, ARM, Sparc, Alpha and MIPS). The host and target architectures may be different, as it emulates several processors (e.g., x86, PowerPC, ARM and Sparc) with support for dynamic translation. On x86 hosts QEMU also allows to execute the guest instructions directly. On Linux host operating systems a kernel module even permits execution of privileged instructions. In addition, QEMU incorporates a Linux user mode emulator to run Linux processes compiled for one target processor on another processor without having to emulate a complete virtual machine.

VirtualBox / VirtualPC / VMware

Sun's VirtualBox [35], Microsoft's VirtualPC [55], and VMware's Workstation [68] are hosted virtual machine monitors running next to traditional operating systems, such as Linux, Windows and Mac OS X (except for VirtualPC, which only supports Windows hosts) with host kernel extensions. In contrast to QEMU these do not emulate the target processor and, thus, only support x86 architecture hosts. On the other hand they also support recent processor generation's virtualization extensions.

KVM

Qumranet's [62] KVM [61] is a variant of QEMU running on Linux on the x86 architecture that uses a kernel module already incorporated in the mainline Linux kernel. In contrast to QEMU it does not emulate the target processor, but is instead essentially based on hardware virtualization support to run an unmodified guest.

3.2.2 Native Virtual Machine Monitors

Xen

Xen [9] is an open-source native VMM, originally developed at the University of Cambridge. Xen enables to run paravirtualized guest systems, called domains, using modified kernels of Linux, different BSD derivatives, and OpenSolaris. In addition unmodified guest systems are also supported on hardware with virtualization support. In the latter case the device model is based on QEMU code.

A special privileged domain (dom0), which currently has to be a modified Linux, has full access to the hardware and contains all device drivers. In contrast, hardware access is restricted for normal unprivileged domains. Instead, device support is realized through stub drivers that communicate via asynchronous notification mechanisms and ring buffers of shared memory with the privileged domain. If running unmodified guests, Xen also supports physical device pass-through. Therefore, a patch enables the dom0 Linux to exclude selected devices. Xen's second generation I/O architecture [19] aims at deprivileging dom0 by running each driver in an isolated driver-specific virtual machine.

Xen runs on all x86 architecture processors. There also exists an IA64 port. It can be used on SMP machines to support multiple virtual processors for each domain, where the number of physical and virtual processors are independent from each other, which allows to run SMP operating systems inside domains.

Hyper-Viridian

As Xen's popularity increased, Microsoft decided to incorporate its own virtualization solution into the next Windows Server version. Its hypervisor Hyper-Viridian [54], formerly known as Viridian, bears resemblance to Xen's design. The privileged domain, called root partition, drives the devices and provides virtualization services via Virtualization Service Providers (VSPs) to other unprivileged child partitions, the Virtualization Service Consumers (VSCs). Obviously, the root partition is to run a Microsoft Windows Server operating system. In a future release, these standard server version will be replaced by a specialized minimal-footprint server version.

Chapter 4

Design

Developing a machine emulator is beyond the scope of this document. Instead I strove in creating a framework for virtual devices, which operates as efficiently as possible. The decisions that lead to the final design shall be discussed in this chapter.

4.1 Library vs. Server

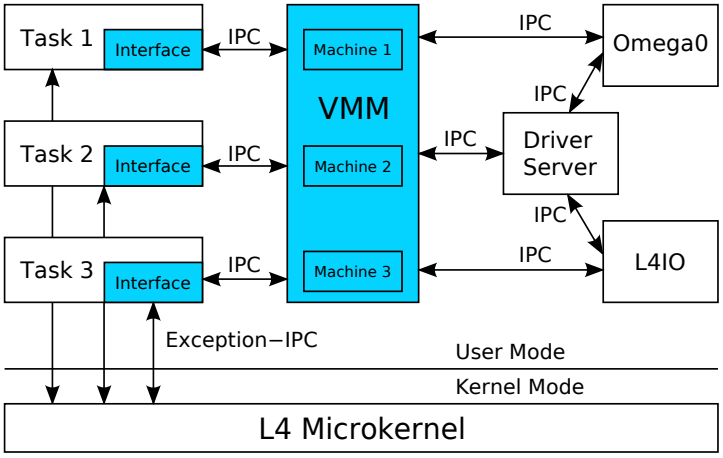

Figure 4.1: A VMM running as a server on behalf of three client tasks.

Obeying the concept described in Section 2.2.2 on page 12, I have to handle exceptions caused by instructions executed with insufficient privileges. The question arises whether to implement the virtual machine monitor as a task of its own, capable of serving multiple clients, or directly in the client's address space, serving only this specific client. Figures 4.1 and 4.2 on the next page illustrate these approaches.

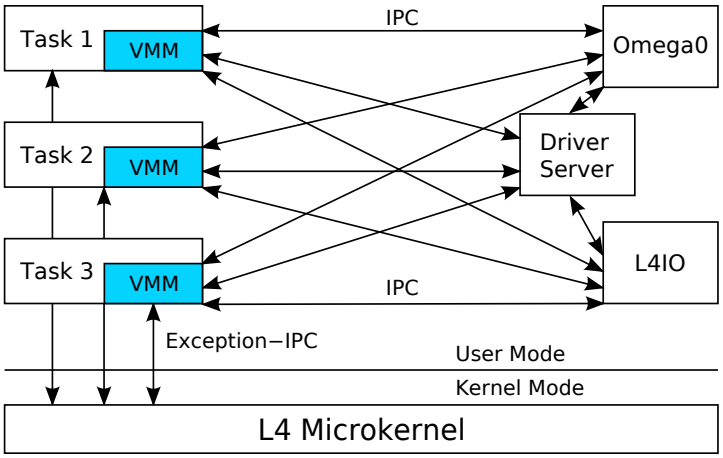

Figure 4.2: Three VMMs running in the client's address spaces.

A separate VMM server offers the possibility to be protected from accidental or intentional modifications by its clients. On the other hand, this protection is not a serious benefit. With the library approach a client might, nevertheless, only modify its own virtual machine, probably causing itself to crash. This solution is indeed more secure, because in a server scenario a client may exploit a server's design flaw to gain access to resources originally requested for another client's virtual machine. Furthermore, a VMM has to interact with its client in some way. Interaction in a server approach would produce considerable IPC usage, whereas with a library faster function calls would suffice. Ultimately, I chose to implement a library. A separate server could be implemented in a future step, using the library as its backend.

4.2 Resources

Following the aim to develop a framework to support virtual devices, I will, in the following sections, investigate how to provide a similar interface to the operating system. This includes discussions about granting access to their resources and handling these accesses efficiently. The hardware related fundamentals can be found in Section 2.1.2 on page 8.

These resources may be provided by the VMM by using the underlying system services in the same way as other L4 applications. For example, if an L4 network server drives the system's network adapter, the VMM (i.e., its virtual network adapter) has to use this L4 server for network communication, because only one program can drive one particular hardware device at a time. In contrast, if a physical device shall be passed through, these resources may be directly mapped to their physical counterpart.

Devices are driven through registers to program commands, the commands' arguments, and to obtain the resulting status code. These registers are made accessible by mapping them to either the physical memory space or, on the x86 architecture, the I/O space.

4.2.1 Memory Space

I/O memory can only be accessed by the operating system after having been mapped into a virtual address space. As shown in figure 4.3, a virtual machine monitor either can map frames decoded by a physical device if it shall be passed through (case a), or use arbitrary frames of main memory to provide a virtual device's region otherwise (case b).

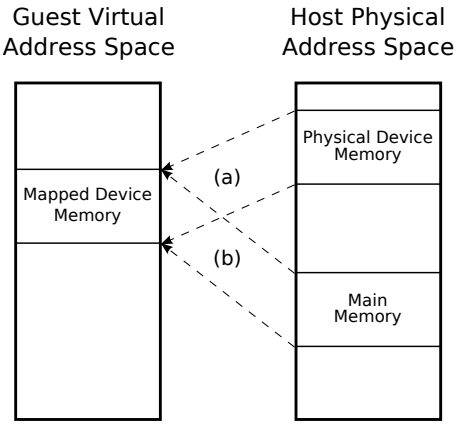

Figure 4.3: Device memory mapped to a virtual address space.

Granting Access

An operating system kernel running in privileged mode can modify its page tables according to its needs. A task running in user mode on top of a micro-kernel is never allowed to do so. Allowing a user mode task to manage its virtual address space in that most transparent way, requires to emulate a virtual memory management unit (MMU), yielding full virtualization. On the other hand this approach would interfere with L4ENV's Region Mapper, who manages a task's virtual address space using micro-kernel primitives, so none of its functionality would be available anymore. Thus, the library would not be suitable for L4ENV tasks, such as L^{4}LINUX.

Handling Access

After access has been granted, a driver can use the acquired resource. However, the VMM has to ensure that this memory behaves identical to an existing device's memory region. In contrast to memory space that can only store data to be read back the way it is, memory mapped registers can behave differently. For instance, a status register may mandate a read-write-clear policy. That is, if a 1 bit is written, the corresponding register bit is cleared. A 0 bit has no effect. Another example can be an interrupt identification register whose bits identifying the interrupt reason are cleared whenever a data register is read.

For this reason, depending on the device's functionality, accessing a hardware register may involve immediate operation. These accesses, therefore, have to be intercepted by the VMM to perform the necessary changes on its device model. To address this problem, there are three different approaches conceivable:

- Either restrict access by mapping memory read-only to trap write accesses (according to the first example) or disallow access by not mapping memory at all to trap both read and write accesses (according to the second example).

  Thus, the processor interrupts execution by asserting a page-fault exception, handled by the kernel, which, in a micro-kernel operating system, hands it over to the pager in user mode. If the VMM provides such a handler, it can emulate the faulting instruction's behaviour by simulating the semantics of the specific register being accessed and incrementing the program counter thereafter. However, this mechanism is quite slow, because it requires the actual register state to be obtained from the kernel and transferred back via exception IPC [17]. On the other hand, it is the most transparent approach, requiring no code modifications at all.

- If it is possible to adapt the client's code, it would be more efficient to replace all accesses with appropriate function calls to the VMM library instead.

  Linux for instance already provides special functions to access I/O memory to ease portability to other architectures. These functions can easily be modified. However, not all drivers in recent Linux versions are developed with portability in mind. There are still important drivers accessing I/O memory directly.

- Do not support trapping of memory regions. Because this only affects virtual devices, it limits the variety of devices that can be implemented.

On the x86 architecture trapping I/O memory access imposes another problem. There are numerous instructions, which support memory operands, increasing the VMM's complexity

while reducing emulation performance. The virtualization extensions incorporated into recent processor generations already address this problem.

In contrast, memory regions containing only data (from the device's point of view) transferred to or from the device, for instance a graphic adapter's frame buffer, do not need to be trapped at all.

4.2.2 I/O Space

The I/O Space on the x86 architecture differs from the aforementioned memory space in two key aspects:

1. There is no task-local virtual I/O space that can be mapped to the physical space with a specific granularity. Instead, I/O space is mapped 1:1[1]. Access can be restricted using the x86 architecture's I/O bitmaps on a per-address basis, causing general-protection exceptions on violation. Distinction between read and write access is not possible.
2. I/O Space obeys the load/store concept as mentioned in Section 2.1 on page 5.

Thus, a VMM always has to intercept access to virtual devices to prevent the client from accidentally modifying the underlying machine. In contrast, physical devices' I/O space resources can be passed through by granting the client access, but relocations have to be prevented. Because there are only a few special instructions available, these can easily be emulated.

4.2.3 Conclusion

Most of the older devices, which are more suitable for emulation because of reduced complexity, compared to recent ones, provide their resources in I/O space, so it is necessary to support them. Because trapping the instruction is the most transparent approach while function calls boost performance, I chose to implement both. These accesses always have to be intercepted. Therefore, I/O space shall be used for efficient register access.

In contrast, memory offers the ability to be accessed without being trapped at all. Thus, memory is to be used for regions containing pure data. To grant access to memory resources, I chose to implement an API identical to L4IO, which already defined a protocol for mapping memory regions, thereby retaining compatibility to existing TUD:OS components. Full virtualization shall be considered a future enhancement. Likewise, trapping memory access on the x86 architecture is a rather complex and slow task that can be avoided, because a VMM only needs

[1] Or to take it differently: cannot be mapped at all.

Device	Usage	Interception	I/O Space	Memory Space
Physical	Data	not required	supported (not trapped)	supported (not trapped)
	Registers	not required	supported (not trapped)	supported (not trapped)
Virtual	Data	not required	supported (trapped)	supported (not trapped)
	Registers	required	supported (trapped)	not supported

Table 4.1: Physical and virtual devices' I/O regions and their usage.

one device of a certain type. For instance, to support networking, the VMM has to provide only one virtual network adapter out of a wide range of devices on the market today. So it is possible to choose one that provides its registers in I/O space instead of memory space. This decision, however, limits the variety of devices that can be implemented, so a future step would be to implement this functionality as well. As this examination only applies to virtual devices, a physical device's resources can be passed through. It is not necessary to intercept accesses. Thus, both spaces can be fully supported. Table 4.1 summarizes these aspects.

On other architectures, such as ARM, without a distinct I/O space, the question does not arise. Memory space has to be used for all regions. Fortunately the ARM architecture obeys the load/store concept, thereby having only a few instructions to load from or store to memory, which can be handled very easily analogous to I/O space on the x86 architecture.

4.2.4 PCI Configuration Space

The x86 architecture offers three different ways to access the PCI configuration space. The BIOS functions are known to be poorly implemented and modern operating systems commonly avoid their utilization. Moreover, if pretending the presence of a BIOS, the operating system would also try to use it for other purposes. Thus, this approach requires to implement a fully-featured BIOS, which incurs a lot of superfluous functions. These functions are only necessary to support legacy operating systems. Modern operating systems can drive hardware, including a PCI bridge, on their own.

The memory mapped mechanism, as described in the PCI Express specification [57] requires to trap memory access, because the configuration space contains registers whose effect has to be emulated. But the configuration space is primarily accessed when the client boots to discover and configure the devices. Performance is not the most important aspect at that time. Since I/O space access also has to be intercepted and the base address in memory space has to be programmed using the I/O space based protocol, the memory mapped approach offers no benefits on the x86 architecture. Using the registers in I/O space, offered by the host-to-PCI

bridge in the MCH, on the other hand, is described in the PCI specification [58] and expected to work on other architectures as well. Thus, the protocol would remain unchanged. Only the access mechanism may vary, allowing to reuse the vast majority of the implementation. Therefore, I chose to implement the I/O space based mechanism.

4.2.5 Interrupt Requests

Providing interrupts requires to communicate the association of devices and IRQs to the client. Because the PCI specification defines a specific register for this purpose, which allows an implementation portable to other architectures, it was obvious to use it. This solution, furthermore, avoids the need to implement an IRQ router, its routing tables and the ACPI system configuration tables. Clearly this approach cannot be applied to legacy devices. Fortunately these devices usually operate with fixed well-known IRQs, so I chose to depend on the operating system's drivers to respond to those assertions.

When devices assert an IRQ the corresponding interrupt handler has to be executed. On the x86 architecture the operating system has to set an interrupt descriptor table, containing the handler's addresses, using a privileged instruction. The FIASCO micro-kernel could trap this instruction and emulate a thread-local interrupt descriptor table. This mechanism was not suitable to virtualize a task. Instead this descriptor table should have been applied to all threads of a task. However, this feature was subject to be removed. Although it is the most transparent approach that requires no adaptions to work with L4VMM, it is not portable. Because OMEGA0 already provided a protocol for TUD:OS to deliver IRQs in a portable fashion, I chose to implement, analogous to the mapping of memory regions, an identical API. Thus, at least applications that are ported to TUD:OS require no modifications.

4.3 Bus Master Direct Memory Access

Emulating direct memory access offers no considerable benefits compared to regions in memory space, because in both cases the processor has to transfer the data. Unfortunately, there are devices, which do not posses on-board memory to support memory regions, and transferring data through programmed I/O has to be trapped, so it was advisable to implement BM-DMA as well.

In some situations virtual DMA is even more efficient. For instance, if a virtual IDE controller uses a physical disk, provided through a TUD:OS block I/O server, as its backend to store and retrieve data, these blocks have to be copied to a device internal buffer inside the VMM, which

is then read sequentially by the client's driver through a `data` register. These accesses have to be trapped to update an internal state in order to obtain the next piece of data upon the next read operation. And hence, the data is copied slowly and twice. This additional overhead due to trapping each access can be avoided by emulating DMA allowing to transfer larger chunks of data at once. If the VMM, furthermore, is able to instruct its backend to directly store the data in the client's target buffer, the required overhead can be minimized to one transfer by the host storage device driven by the VMM's backend.

4.4 Multi Threading

A single device is in general not thread-safe, so I can rely on the client, accessing the library, to ensure protection against concurrent modifications. However, real hardware, in particular the chipset, provides only one set of signal lines to serve exactly one processor's memory or I/O space access[2]. Thus, only one thread can access these spaces at a time. Moreover, an access consists of one single instruction that cannot be interrupted. This yields the effect of every single access being atomic by nature.

Emulating virtual hardware violates this assumption. Trapping these instructions replaces one instruction with several instructions executed by the VMM to update its device model. Thus, the VMM might be interrupted by the micro-kernel's scheduler while hardware emulation is still in progress and the machine model is not yet updated to the final state. Furthermore, there are several API functions that allow to access a particular memory or I/O space address concurrently.

To yield a behaviour identical to real hardware, it would have been necessary to use a lock upon each VMM entry[3]. This lock can shortly become a huge performance bottleneck, so I decided to implement the library fully reentrant by only protecting the core data structures using fine-grained locks. This approach allows concurrent access, hoping they will target different devices. A client, being aware that accessing a device concurrently results in unpredictable behaviour, can still access different devices efficiently. On the other hand, this means a device's programmer has to correctly deal with this problem.

[2] For this theoretical consideration, I will ignore the fact that multi-channel chipsets are also available.
[3] Or at least all functions that directly or indirectly handle resource access.

Chapter 5

Implementation

In this chapter I will describe the implementation of facilities that were needed for this thesis. I will start with an overview on the library structure. Following this I will describe the exception handling mechanism, which is used to emulate access transparently. This section is followed by an introduction to machine configuration, how the host bridge works, and an explanation of key aspects of physical devices, interrupt delivery, and DMA. I conclude this chapter by highlighting changes to existing L4 components as well as L4VMM's limitations compared to real hardware.

5.1 Library Structure

Despite the intention to implement one library the final result comprises several libraries, as shown in figure 5.1 on the following page, to meet the requirements of various TUD:OS components:

Core Framework The core library is the most integral part of L4VMM containing the data structures the machine model consists of, the ability to decode and emulate instructions, and the configuration parser.

Devices The device libraries extend the core framework by providing device implementations either in one or in separate libraries. For this reason a developer may choose to link only the devices that are actually necessary, thereby reducing the application's code base.

L4 API The API library serves as a C adapter to the internal object-oriented API. If a server-based implementation shall be considered in the future, this library has to be replaced to communicate with the server.

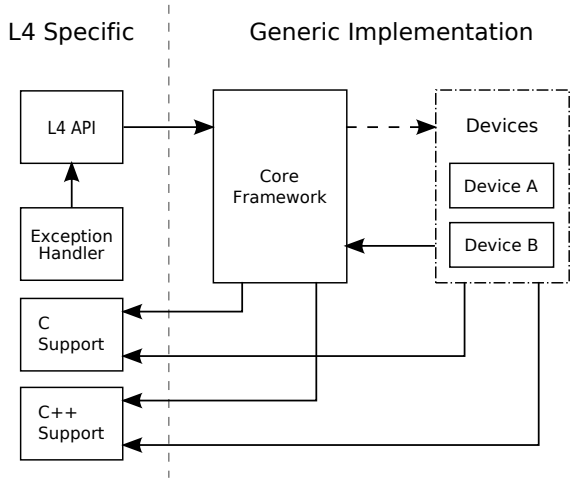

Figure 5.1: L4VMM library structure.

Exception Handler In order to develop applications more conveniently, L4VMM also provides an exception handler incorporated into a separate library, allowing it to be omitted with applications implementing their own exception handling mechanism, such as L^{4}LINUX.

C and C++ Support In addition to the preceding ones, two support libraries provide L^{4}LINUX with the missing C or C++ symbols, respectively. While the latter is complete and may be also used for arbitrary applications, the former is specifically designed for L^{4}LINUX.

5.2 Access Emulation

Since virtualization intends to run operating systems in an isolated environment, these systems are likely to execute privileged instructions to modify the machine state. As explained in Section 4.2 on page 24, these virtualization-sensitive instructions might need to be intercepted by the VMM. Any attempt to execute one of them will result in the micro-kernel delivering a fault IPC to the corresponding handler. This IPC can be a page-fault IPC if trapping memory space access or an I/O page-fault IPC in case of I/O space, respectively.

Therefore, L4VMM provides a pager, which replaces the Region Mapper's pager thread during task startup. This pager determines whether the fault is to be emulated and if it is replies to the kernel requesting an exception IPC ([17]). In addition to the page-fault address and the current instruction pointer, an exception IPC delivers the faulting thread's register state through the UTCB data structure, which is handed over to L4VMM's general purpose emulation function:

```
int l4vmm_handle_exception (l4_utcb_t *utcb);
```

This function checks the opcode at the current instruction pointer to determine which instruction led to the fault. For instance, the instructions **IN** and **OUT** read from or write to an I/O port. L4VMM emulates their behaviour by obtaining the source operands (e.g., the port number) from the UTCB CPU state and calling the attached device's I/O space handler function. This function is one of the following depending on the type of access:

```
virtual l4_uint32_t read_ioport  (l4_port_t port,
                                  access_size access_size) = 0;

virtual int         write_ioport (l4_port_t port,
                                  l4_uint32_t data,
                                  access_size access_size) = 0;
```

A similar interface exists for memory access, although it is not yet used. By providing these functions, a device's programmer may update the device's internal state according to its needs.

Nevertheless, for this to work the library has to maintain a data structure that allows to retrieve an I/O port's handler efficiently. Because on the x86 architecture there are only 65536 I/O ports available, I chose to use an array. If memory consumption is critical or an array would be too large (i.e., if trapping memory access), a multi level lookup table similar to page tables could be used instead. Finally, after the handler returned, L4VMM writes the results back to the UTCB's destination operands and increments the instruction pointer.

If the fault is not to be emulated, L4VMM's pager forwards the page-fault to the Region Mapper. This approach clearly can only work if the application does not install its own pager. If it does, such as L^{4}LINUX, the developer has to ensure that `l4vmm_handle_exception` is called when appropriate.

5.3 Data Structures

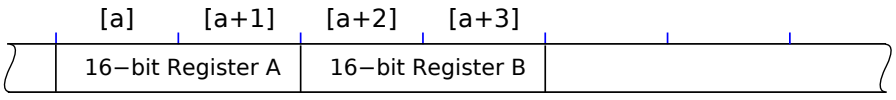

Figure 5.2: Sample register block containing two registers.

In contrast to the explanation, I gave in Section 4.2.1 on page 25, access to emulated device registers may not only be intercepted because of functionality reasons, but also because of implementation reasons. Let us assume a specification mandates a data structure in memory space similar to figure 5.2, with two 16-bit registers being at addresses `[a]` and `[a+2]`, respectively.

Allowing the client to directly access the data structure is only possible if the following conditions are met:

1. Each register is stored at the exact location mandated by the device specification.
2. In case of multi-byte registers, the mandated byte order is obeyed. This order may differ from the host processor's byte order, but is expected by the client accessing a particular register.

Because in practice only few locations require to implement side effects, but the registers are located in I/O space or, if it is located in memory space, access to the whole page has to be intercepted, a VMM can also simply emulate access to most registers by determining its address and reading or writing the data word, resulting in code similar to the following example to access register B:

```
unsigned value = le_to_cpu32(((char *)&sample_regs) + 2);
```

While it is reasonable to emulate client accesses in such a manner, emulation code is more readable if using structured data types:

```
unsigned value = sample_regs.register_B;
```

Unfortunately, C/C++ data structures don't obey the preceding constraints for the following reasons:

1. The compiler may choose to align members of a data structure to a multiple of the machine word size leaving padding bytes in between, because on most architectures word-aligned accesses are faster.
2. The compiler expects the data to be stored in host byte order. For instance, the client may choose to read both registers from the preceding example at once by using a 32-bit access to address `[a]`. Emulating this access may require to swap the byte order of both half-words independently instead of swapping the whole 32-bit word. In particular on the x86 architecture where unaligned accesses are permitted, the VMM has to determine which members are affected, swap their values accordingly, and merge the resulting words together to obtain the value to be returned. Hence, large data structures will require a lot of cases to be considered, considerably reducing emulation performance.

In order to combine both benefits I used C/C++ data structures that obey the hardware specification by using GCC's compiler attribute `__packed__`. In addition, in order to write portable data structures, I implemented an integer template that behaves like a built-in integer type, but always stores its value in a specified byte order decoupled from the host architecture.

```
template <typename WordT, template <typename> class ByteOrder>
struct __attribute__((__packed__)) endian
{
    // ...
};

typedef endian<uint32_t, little_endian> le_uint32_t;
typedef endian<uint32_t, big_endian>    be_uint32_t;
```

5.4 Machine Configuration

To allow the user to configure L4VMM and the machine to be emulated in a flexible manner, I designed a configuration file, which is parsed using flex and bison. In addition, this file also allows to select host devices that shall be passed through.

5.5 Host Bridge

With respect to device discovery and configurability, the host-to-PCI bridge in the MCH is an important part of a computer system. To incorporate both virtual and selected physical devices, the host bridge has to be implemented in software. Therefore, I developed a bridge compatible to Intel's 440FX PCISet [38]. This bridge attaches itself to the PCI I/O space addresses, as described in Section 2.1.2 on page 8, and decodes the PCI configuration space access protocol to forward accesses to the addressed device through designated functions each PCI device has to implement:

```
virtual l4_umword_t read_config_space  (uint8_t offset,
                                        access_size access_size) = 0;

virtual int         write_config_space (uint8_t offset,
                                        l4_umword_t data,
                                        access_size access_size) = 0;
```

5.6 Physical Devices

Passing physical devices and, thus, their resources through to a virtualized environment bears resemblance to the way a pager works. L4VMM waits for the client's I/O space access or I/O memory map requests and maps the corresponding physical region on behalf of its client.

Once the client's drivers are initialized, direct access to their resources has been granted. Thus, L4VMM incurs no additional overhead during further operation.

However, with respect to device configuration, there is one conceptual difference: The client must not be allowed to locate physical resources in the host's physical address spaces. For

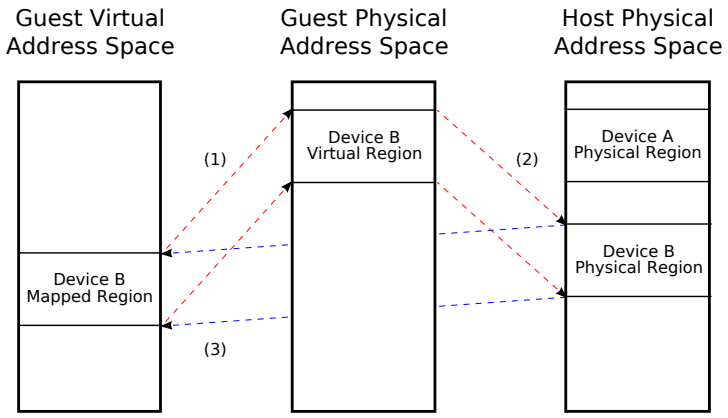

Figure 5.3: Physical device memory mapped to a virtual address space.

instance considering a configuration as shown in figure 5.3, the host system possesses two physical devices A and B, whose resources are located in distinct locations in physical memory space. If only device B is to be passed through, the client might try to locate it at the same position where device A actually is, due to the fact that L4VMM conceals its existence. This algorithm is deterministic assigning each device a reasonable location in memory space starting from a high address. (See section 2.1.2 on page 9). If this relocation would be passed through to L4IO, both devices would not work anymore as the specification mandates that the resulting behaviour is undefined [58]. In addition relocating physical devices' memory regions is a security-sensitive operation. A malicious task might program an arbitrary base address to cover physical memory mapped to another task or even the micro-kernel. As a result the MCH would forward access to a device instead of addressing main memory revealing crucial information or causing the system to crash. For this reason PCI configuration space access has to be intercepted to filter out access to its address registers. To allow these relocations, L4VMM introduces another memory space presented to its client, the guest physical address space, which stores the devices' physical regions' locations. Upon map request (step 1) L4VMM determines the actual physical region in the host's physical address space (step 2) that has to be mapped to the client's guest virtual address space (step 3), yielding the effect that memory regions seem to be relocatable.

Unfortunately this approach cannot be applied to regions in I/O space, because the x86 architecture does not provide a task-local virtual I/O space. To solve this problem, I had to exploit

the PCI specification, which states that a device may hardwire an address register's lower not modifiable bits. The remaining high-order bits may then only be used to program this regions location. If L4VMM denies changes to the whole address register while providing a suitable value obtained from L4IO, the client has to cope with that base address. As a result the region is pinned in I/O space.

I also considered allowing relocations of physical I/O space regions, but discarded this solution due to the following drawbacks:

- Accesses to relocated regions have to be intercepted and forwarded to its actual location by L4VMM. This additional overhead (see Section 7.1 on page 45) is unnecessary.
- If the relocated and the actual regions overlap, parts of the I/O space have to fulfill both requirements: Being trapped if the client accesses and being accessible by L4VMM. As an emulation library runs in the same address space, this behaviour cannot be implemented without unmapping and mapping these ports upon each access. In addition such an implementation might cause race conditions as the library is designed to be reentrant (see Section 4.4 on page 30).

Although switching to a server based approach would solve the latter problem, the former still remains, so I chose to not support these relocations in favor of the resulting performance gain.

5.7 Interrupt Requests

Because I chose to implement an API identical to OMEGA0 to support interrupts, as explained in Section 4.2.5 on page 29, passing physical devices' IRQs is easy to implement. L4VMM introduces an additional layer between OMEGA0 and the client task, calling the corresponding functions if necessary. In contrast, requests to serve a virtual device's IRQ must not be passed to OMEGA0. Instead L4VMM provides its own IRQ state including an interrupt controller's masked and pending behaviour. To block or wake up an attached IRQ thread, L4VMM uses IPC wait or IPC send calls, respectively.

However, one key problem had to be solved: IRQ number assignment. Consider the following example: The host machine consists of several disabled legacy devices, including a serial line, to prevent the BIOS from reserving its resources. Thus, a network adapter could be assigned IRQ 4, which would otherwise have been the serial line's IRQ. If configuring a virtual machine that passes through this network adapter while at the same time emulating a virtual serial line, both devices have to use this IRQ. The network adapter because the BIOS (or L4IO) assigned this IRQ. The serial line because its resources are at fixed well-known locations. Sharing IRQs

between devices is supported on the x86 architecture to overcome restrictions due to its poor design. Nevertheless, L4VMM does not support IRQ sharing[1], because the two mechanisms to wake up a waiting IRQ thread are different from each other. In the preceding example OMEGA0 has to be called to wait for the physical device's IRQ to be asserted. In addition that same thread has to be woken up if the virtual serial line asserts an IRQ. Unfortunately L4VMM cannot introduce an own IPC message, because the OMEGA0 client library's function for security reasons uses a closed-wait to wait for a particular service thread running inside the server. Although it is possible to solve this problem by creating an internal service thread that attaches to OMEGA0 and wakes up the client thread using the same mechanism implemented for virtual devices, this solution needlessly incurs additional scheduling overhead that increases IRQ latency.

As a conclusion a physical and a virtual device must not assert the same IRQ. Therefore, L4VMM assigns a physical device a virtual IRQ that is presented to its client whereas calls to OMEGA0 use the actual IRQ obtained from L4IO. This approach consumes more IRQs (as no sharing is allowed), but imposes no additional problems. A virtual machine might theoretically support an unlimited number of IRQs, although more than 256 are impractical due to hardware interface restrictions.

5.8 Direct Memory Access

In order to show that the infrastructure developed for this thesis suffices to develop virtual devices, I implemented a virtual PCI IDE storage controller conforming to the general interface specifications [36, 37, 66]. In particular I chose to implement the IDE unit incorporated into recent Intel chipsets ([39, 40, 44]). Besides being an essential part in modern virtualization solutions, storage controllers also allow to prove that most of the designed subsystems work correctly, as they rely on I/O or memory space to locate their registers, IRQs to acknowledge commands, and DMA for efficient data transfers.

As mentioned in Section 2.1.2 on page 10, devices are not subject to the processor's memory management unit. Thus, drivers have to provide physical addresses as source or target when programming DMA transfers. In principle, emulating DMA requires to copy blocks of memory from the service used as backend to the client's target buffer, or vice versa. Therefore, these addresses have to be translated back to virtual addresses referring the emulator's virtual address space. This address translation can be implemented in two ways:

[1] The problem of sharing IRQs is set to disappear completely with the introduction of message-based interrupts as part of PCI Express [57].

- The Region Mapper, running in each L4ENV task, provides functions to retrieve the physical addresses of attached regions identified by their virtual address. As L4VMM is linked directly to the target application, it can use these functions. However, because the Region Mapper only supports translating virtual to physical addresses, L4VMM has to search the whole address space, resulting in a considerable performance penalty. To speed up further lookups, L4VMM provides an optional cache as the client is very likely to reuse its buffers. Obviously this cache can only be used if all affected buffers are pinned in physical memory.
- Assuming a client task such as L^{4}LINUX knows its specific requirements more precisely than such a generic implementation, a client supplied lookup mechanism using a call-back function may be more efficient. In fact, in L^{4}LINUX only the Linux server drives devices. Because its memory consists of a single consecutive virtual region, only this region needs to be searched. Moreover, L^{4}LINUX already implements this mechanism for other purposes.

As both approaches have advantages and disadvantages, I chose to implemented both using the former as a fallback mechanism if no lookup function has been supplied during initialization.

5.9 Required Changes to FIASCO and L4 Servers

Although I aimed at not changing existing applications, some minor modifications were still necessary:

- FIASCO-UX did not support I/O flexpages to report I/O page faults, due to the fact that it is running under Linux and does not even has access to I/O ports. Nevertheless, to emulate instructions accessing I/O space, it should provide this information in the page fault IPC's data word, what I had to implement.
- To support physical devices' I/O regions, these regions have to be requested from the responsible server L4IO, whose API already defined the appropriate functions. However, they were not implemented, so I wrote the missing parts.

5.10 Limitations

There are a few limitations when using L4VMM compared to real hardware, users should be aware of:

1. An application must not modify page tables to map or unmap I/O memory. Instead, a library function has to be called. An operating system designed with portability in mind such as Linux (and, thus, L^{4}LINUX) usually provides special functions for this purpose that can easily be adapted.
2. Likewise an application must not load an interrupt descriptor table. Instead, library functions have to be called to attach to or to detach from interrupts.
3. On the x86 architecture, the emulation of instructions accessing memory is not yet implemented. Thus, a virtual device's developer may use memory regions only for data, which does not need to be intercepted.
4. The **CLI** and **STI** instructions to enable and disable interrupt delivery, offered by the x86 architecture, can be trapped and emulated, but there is one conceptual difference. According to the Intel microprocessor documentation [41], the **STI** instruction enables interrupts after the instruction following **STI**, allowing to return from a function with interrupt delivery deferred until after the return. This special behaviour cannot easily be emulated. Although executing the following instruction in single-stepping mode, would allow a semantically correct implementation, I preferred a simpler solution as no recent operating system relies on this delayed effect. Thus, interrupts are enabled and if pending delivered as soon as **STI** has been executed.
5. Besides these instructions, it is possible to change the interrupt enable flag (**IF**), stored in the **EFLAGS** register, using the **POPF**, **POPD**, or **POPQ** instructions. When invoked with insufficient privileges the **IF** bit simply does not change, but the processor does not raise an exception, and, thus, code, which silently relies on this feature instead of using **CLI** and **STI**, will not work as supposed.

If using L^{4}LINUX, the last two limitations are less important, as L^{4}LINUX has already been adapted to not rely on these instructions [33].

Chapter 6

Using The Library

In this chapter I will shortly describe the steps that are necessary to use the library from different points of view.

6.1 Develop Applications

Assuming the application is already ported to L4 and, thus, uses OMEGA0's or L4IO's client API, it is sufficient to include a special header `<l4/vmm/vmm-compat.h>` in the relevant modules, which modifies the original calls to call L4VMM's functions instead. Otherwise it is more difficult, yet not impossible, to identify code, which deals with IRQs and I/O memory remapping that has to be adapted.

Furthermore, to provide the virtual machine's configuration, one has to call `l4vmm_init` during startup with either a file's name, which is opened and parsed automatically, or the (same) configuration data itself.

6.2 Develop Virtual Devices

Developing a virtual device, includes the following steps:

- Derive a new class of the appropriate base class. That is `device_base` for an arbitrary device or `pci_device_base`, which is more suitable for PCI devices.
- Implement and register a factory function, which instantiates the device, when required by the framework.
- Implement the new class's constructor to allocate resources. Moreover, PCI devices have to initialize its configuration space.

The following skeleton code snippet for a ‘Hello World’-PCI device, illustrates these steps:

```
#include "devices/common.hpp"

// A device consists of at least one class.
class hello_device : public pci_device_base<pci_config_space>
{
  public:
    // The constructor.
    hello_device(machine_base &machine);
    // The factory function. Must be a static class member.
    static int create(vector<pci_device *> &devices,
                      machine_base &machine,
                      config_node &device_node);
};

// Register the device's factory function.
REGISTER_PCI_DEVICE(hello, &hello_device::create);

// Implement the factory function.
int hello_device::create(vector<pci_device *> &devices,
                         machine_base &machine,
                         config_node &device_node)
{
    // Create the new device and add it to the returned vector.
    devices.push_back(new hello_device(machine));
    // No error during instantiation. Otherwise error code.
    return 0;
}

// Implement the constructor.
hello_device::hello_device(machine_base &machine)
    : pci_device_base<pci_config_space>(machine, "HELLO PCI")
{
    // Print the hello message.
    log::debug("Hello PCI bus. %s is there.\n", name());
    // Initialize the mandatory registers:
    config_space.vendorID=0x10ec;           // e.g., Realtek
    config_space.deviceID=0x8139;           // e.g., RTL-8139
    config_space.class_code=0x020000;       // e.g., Ethernet
    // Example: Allocate 256 I/O ports at BAR #0.
    alloc_ioregion(pci_ioregion::IOSPACE, 256);
}
```

Listing 6.1: A minimal PCI device

If placed in the correct directory (i.e., `lib/vmm/devices/pci/hello`) the resulting library `libl4vmm-dev-hello.o.a` can be linked to the applications along with the API and core libraries `libl4vmm-api.a` and `libl4vmm-core.a`. In addition, the library `libl4vmm-devices.o.a` always provides all implemented devices, including this one.

For developing, it is advisable to use their debug versions, suffixed with `-dbg`, instead. Finally, after adding a section `hello{}` to the configuration file, the hello message should be displayed.

Chapter 7

Evaluation

7.1 Microbenchmarks

This section examines the performance of resource usage and its overhead using microbenchmarks. For each benchmark I specify the elapsed time in wall clock time and in processor cycles, as measured by the processor's **RDTSC** instruction. If applicable, I additionally specify the ratio compared to a native access.

I chose to measure I/O space accesses, because memory space resources are intercepted only during the configuration of devices. Once the client is initialized, its drivers have direct access to their memory resources. Thus, L4VMM incurs no additional overhead.

The first benchmark (Table 7.1) measures the time the processor has to wait to read a 32-bit value from a physical I/O port, using the **IN** instruction after access to the resource has been successfully granted by roottask. The first column shows the results for the instruction being directly placed inside the measurement loop. These are the results the others are to be compared to. To improve efficiency the library also provides specialized access functions, whose results are shown in the second column. These functions have to determine whether the port is assigned to a virtual or a physical device. In this test scenario a physical device was assigned, so the library uses the aforementioned instruction to fetch the value to be returned. Therefore, these

Machine	Instruction		Function (Physical Access)		
	Cycles	Time	Cycles	Time	Overhead
Athlon XP 1.466 GHz	574	391 ns	574	391 ns	+0 %
Pentium D 3 GHz	2734	914 ns	2822	943 ns	+3.2 %
Core 2 Duo 2.2 GHz	3007	1370 ns	3007	1370 ns	+0 %

Table 7.1: Performance reading a 32-bit value from a physical I/O port.

Machine	Function (Virtual Access)			Decoding & Emulation		
	Cycles	Time	Overhead	Cycles	Time	Overhead
Athlon XP 1.466 GHz	33	23 ns	-94.1 %	57	39 ns	-90.0 %
Pentium D 3 GHz	42	14 ns	-98.5 %	64	21 ns	-97.7 %
Core 2 Duo 2.2 GHz	23	10 ns	-99.3 %	34	15 ns	-98.9 %

Table 7.2: Emulation performance reading a 32-bit value from a virtual I/O port.

Machine	Trapped (Alien)			Trapped (Normal)		
	Cycles	Time	Overhead	Cycles	Time	Overhead
Athlon XP 1.466 GHz	1487	1014 ns	+159.3 %	2306	1573 ns	+302.3 %
Pentium D 3 GHz	5994	2002 ns	+119.0 %	7566	2528 ns	+176.6 %
Core 2 Duo 2.2 GHz	1791	819 ns	-40.2 %	2673	1218 ns	-11.1 %

Table 7.3: Exception handling performance reading a 32-bit value from a virtual I/O port.

functions cannot be faster than the instruction. On the other hand it turned out that the function call incurs almost no overhead in comparison to the physical access latency.

If the resource is not to be passed through, the library has to emulate these accesses. Table 7.2 illustrates the minimal overhead to emulate a particular access. Therefore, I attached a dummy device that simply returns an invalid value. The first column presents the results for the aforementioned specialized access function whereas the second column shows the results obtained for the general purpose instruction emulation function. This function additionally has to decode the instruction to determine the operation to emulate and, thus, takes slightly more time. But both prove to be much faster than a native access.

Since the latter function is designed to be called by pager threads, as described in Section 5.2 on page 32, I further extended the preceding benchmark. The results are shown in table 7.3. The test threads execute native instructions again, which are trapped in this scenario. Compared to the preceding measurement, the exception handling causes considerable overhead: A normal thread causes a page-fault IPC to be sent to its pager, which has to request an exception IPC to obtain the thread's register state. An alien thread in contrast always causes exception IPCs. For this reason alien threads' accesses can be handled more efficiently than those of normal threads. Figure 7.1 on the next page illustrates the measured durations for all types of access.

Finally, the IRQ handling overhead is of interest. For this measurement, a thread requests to consume, mask, and unmask a virtual IRQ at once without waiting for it to be asserted. Thus, Table 7.4 on the facing page shows the accumulated maximum overhead for these operations. The high values compared to instruction emulation in Table 7.2 are due to a kernel entry to obtain the current thread's ID to implement the policy that only the attached thread may request

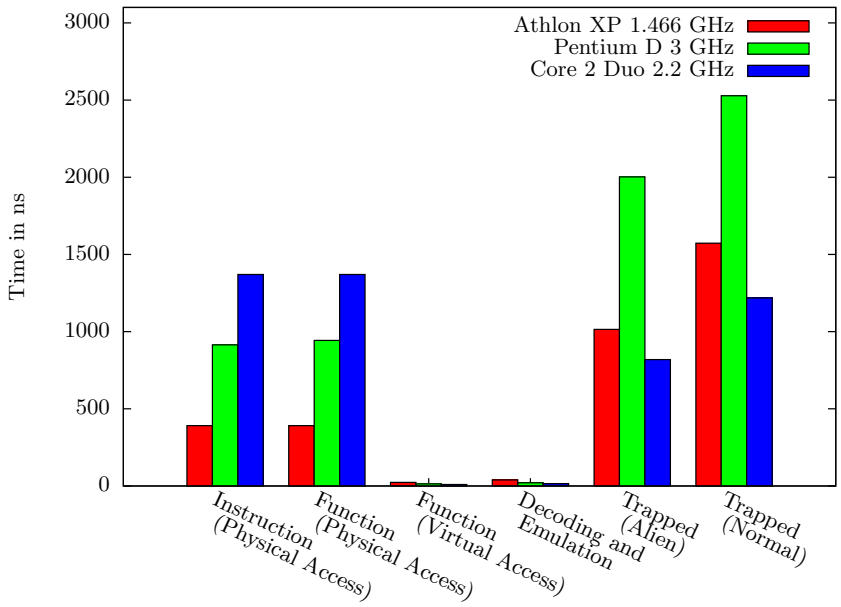

Figure 7.1: Performance reading a 32-bit value from an I/O port.

Machine	IRQ Overhead	
	Cycles	Time
Athlon XP 1.466 GHz	266	182 ns
Pentium D 3 GHz	919	307 ns
Core 2 Duo 2.2 GHz	596	272 ns

Table 7.4: IRQ handling overhead additional to OMEGA0 requests.

actions for a certain IRQ. When waiting for an IRQ this overhead is not important, as the operation is performed before the thread waits. On assertions, the only overhead results from the IPC message to wake up the attached thread from either OMEGA0 to serve a physical IRQ or L4VMM in case of a virtual one.

Lessons Learned

With respect to resource access the paravirtualized approach is clearly more efficient, resulting in approximately 90% performance gain compared to a physical access, although the exact results may vary depending on the emulated device's complexity. Even physical accesses, passed through by the library, do not incur a considerable overhead (which is less than 5%). Thus, function calls to the library are to be preferred whenever possible. On the other hand, native instructions, handled through exceptions, do not require modifications and are more suitable if considering the portability effort.

7.2 IDE Block I/O Performance

Besides the preceding synthetic measurements, I chose to conduct a more realistic task: The virtual IDE controller's block I/O performance. To show, which transfer rates a standard

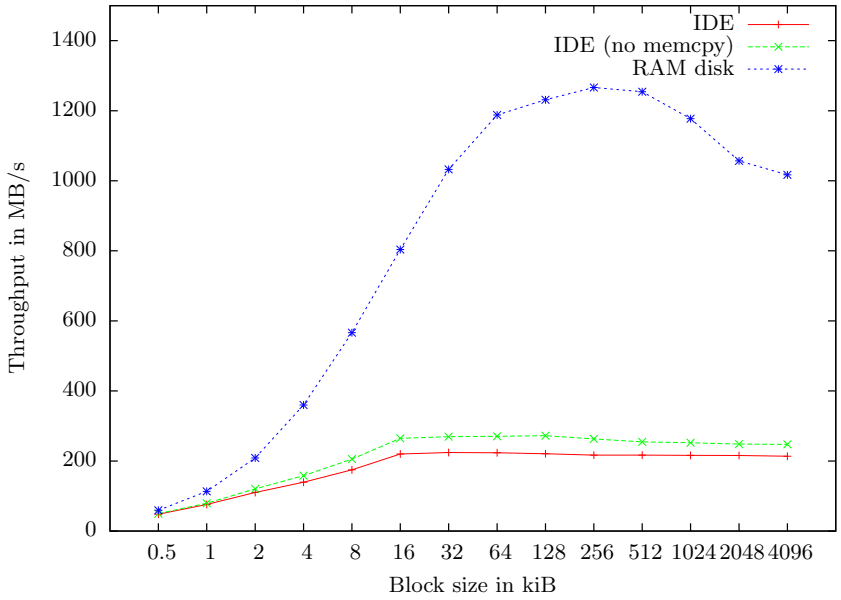

Figure 7.2: Virtual IDE transfer rates compared to a RAM disk.

application could expect, I ran L^{4}LINUX on a Pentium IV 3.2 GHz with 4 GB main memory. An L^{4}LINUX task (dd) sequentially read the whole virtual disk of 128 MB using varying block sizes from 512 byte to 128 kiB. I additionally measured a RAM disk for comparison. During measurement both disk images were already loaded into main memory and L^{4}LINUX's disk caches were disabled. The results are shown in figures 7.2 and 7.3.

The RAM disk (blue curve) reaches up to 1.2 GB/s, which corresponds to the machine's memory transfer bandwidth. In contrast, the virtual IDE controller (red curve) serves 220 MB/s. The increasing transfer rates up to 16 kiB block size are due to the Linux kernel always requesting 32 sectors (16 kiB), even with caches disabled. Larger block sizes show no substantial differences. To investigate the protocol overhead, I repeated the preceding measurement by only emulating the protocol but without copying data to L^{4}LINUX's target buffer (green curve) resulting in a maximum transfer rate of 270 MB/s. Thus, the transfer rates are the result of the complex protocol, which includes several register accesses and two IRQs per block to acknowledge commands. However, in contrast to physical BM-DMA capable storage controllers, the virtual implementation always causes a high processor load.

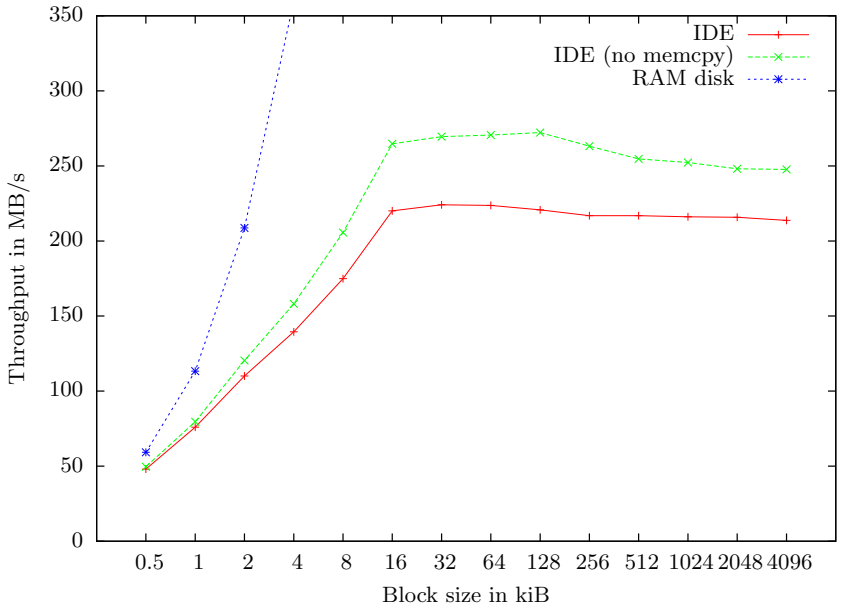

Figure 7.3: Virtual IDE transfer rates more detailed.

Writing is easy. You only need to stare at a piece of blank paper until a drop of blood forms on your forehead.

(Douglas Adams)

Chapter 8

Conclusion and Outlook

In my thesis I integrated means for device emulation and virtualization into TUD:OS. I reused existing TUD:OS solutions such as the OMEGA0 and L4IO interfaces. I developed a framework to implement a wide range of virtual devices, including legacy and PCI devices. Using this framework I wrote a virtual serial line as an example legacy device that supports output through the log server and the CON console service, a mechanism to pass through selected physical resources, and a virtual host-to-PCI bridge to attach PCI devices. With the help of this bridge, I also implemented a virtual IDE controller that supports arbitrary backends, although only a file-based backend is currently available and an adapter to pass through selected physical PCI devices. I, furthermore, showed that the implementation is efficient. Most of the accesses that affect performance are faster than their physical counterpart with up to 90% performance gain obeying the paravirtualized approach. On recent processor generations, even emulated native accesses proved to be up to 10% faster.

In [30] Hermann Härtig addressed the problem of unrestricted use of DMA transfers by untrusted components. A malicious component, although encapsulated in its own address space, may initiate DMA transfers to any physical address, thus in fact breaking the encapsulation of address spaces. He outlined two approaches to tackle that problem. The first one is to disallow DMA on the host machine for untrusted components by relying on hardware virtualization. Thus, an untrusted component only drives a virtual device. Although the DMA operation, necessary to provide the virtual device's functionality, still has to be programmed by a trusted component for efficient data transfers, L4VMM can provide the untrusted component with an anticipated environment.

In addition, in [34] Hermann Härtig et. al. state that a secure I/O architecture for micro-kernel-based operating systems needs to provide means to reliably share unique resources, such as the interrupt controller or PCI buses between separated drivers and the workhorse operating system such as L^{4}LINUX (with the remaining drivers left in the workhorse). L4VMM can be

such a solution, although these devices principally still cannot be shared. Instead L4VMM can provide the workhorse with its own device and IRQ model where only selected physical devices are passed through.

The currently implemented approach, being partially paravirtualized, is fully functional. However, future work shall further extend the machine model concentrating on the following aspects:

Native Operating System Support Access to virtual devices' resources mapped to main memory could be handled transparently, enabling a developer to implement every intended device. For this to work, memory accesses have to be emulated. Complementary as the client might not only be an L4ENV task but an operating system, its memory management currently would conflict with the restrictions imposed by the underlying kernel and, thus, has to be adapted. Providing a virtual memory management unit avoids the necessity for these changes. Analogously the IRQ interface could be improved to be completely transparent, although it would not be architecture independent anymore. Therefore, the FIASCO micro-kernel might need to be extended to allow a VMM server to control and modify the virtualized task. Combined together, these mechanisms would permit to implement full virtualization, allowing to run unmodified guest operating systems under TUD:OS.

Device Implementations Passing through devices is most suitable if performance is the most important criterion, but looses the main advantage of virtualization of multiplexing real devices among several tasks (i.e., a guest operating system and an L4 service). Obviously a virtualization solution would benefit from more virtual devices using the underlying services. Most notably devices that enable basic input and output such as a keyboard controller and a graphics adapter on top of CON or DOPE should be implemented.

Due to the design decision (explained in Section 4.2.1 on page 25) to not support intercepted memory space resources on the x86 architecture, a device currently has to provide its control registers in I/O space. With respect to portability to other architectures, an ideal device would additionally provide them in memory space. This allows faster handling on the x86 architecture, because fewer instructions need to be considered and no address check is necessary as I/O space resources always have to be emulated, while still being flexible. Data shall be transferred using memory space regions allowing the driver to access the resource according to its needs. As the DMA implementation is also based on copying memory, with the disadvantage of a more complex protocol to be set up, it should still be preferred over programmed I/O.

Virtualization of Time Another problem observed during the implementation of the IDE controller is the meaning of time. Once programmed, an operation performed by a physical device is not interruptible anymore, unless terminated by the driver. In contrast, a thread emulating a virtual device's operation might be interrupted if the underlying micro-kernel decides to schedule another thread. As the driver currently sees the total (wall clock) time elapsed, it may time out due to the device not responding within a certain amount of time[1]. However, as the VMM provides the client with its hardware model, it can also influence its time source. Implementing an appropriate solution might be a goal in the future, too.

Virtual ACPI Support A subsequent work to ACPI support in TUD:OS [29], might be its integration into L4VMM. Because virtual machines typically do not consist of as much devices as real hardware, requiring flexible configuration capabilities, and its power interface is not of much use, this aspect can be considered less important. To still save power it is feasible to emulate the privileged **HLT** instruction, which stops instruction execution and places the processor in a halted state, using the `l4_ipc_receive` system call to block the thread until an event occurs, allowing the underlying micro-kernel to enter processor states with less power consumption if appropriate.

In addition, integration into TUD:OS and configurability should be further improved in the following areas:

ACPI Support Lukas Hähnel proposed a new component [29] to manage the physical devices within TUD:OS that incorporates APCI support for device discovery. Having ACPI support would allow this server to fully describe the underlying machine[2]. This information can be used to pass through whole devices automatically. In its current implementation, L4VMM only supports to pass through PCI devices as long as its resources are discoverable through the PCI system. Additional implicitly decoded legacy resources have to be configured manually due to L4IO not providing the necessary information. Likewise legacy devices cannot be configured, unless passing through their resources explicitly.

Storage Support Finally, the virtual IDE controller needs a suitable backend. Loading a large disk image using the GRUB boot loader (for the BMODFS file provider to work) or the TFTP server into system memory takes a considerable amount of time or might even be impossible due to memory consumption. Moreover, both file providers are non-persistent, rendering this approach useless, although write support is implemented in L4VMM.

[1] The IDE drivers in L^{4}LINUX usually wait 30 seconds for the operation to complete before terminating a request. If using the built-in address translation that is based on a full address space search together with excessive logging, this effect can be observed.

[2] Or at least all devices that shall be allowed to be passed through, depending on its policy.

A solution could be to either implement a block service, allowing the IDE controller to store and retrieve blocks, or a complete filesystem service for TUD:OS.

Hardware I/O Virtualization Support If passing through physical devices, the problem of unrestricted DMA usage by untrusted components remains unsolved. Thus, a guest to which access to such a device has been granted still has to belong to the trusted computing base. Proposed enhancements of hardware-assisted I/O device virtualization such as Intel's VT-d [45] and AMD's SR-IOV [8] that selectively constrain I/O device memory access to a specific portion of physical memory by means of multiple DMA protection domains sound promising [6]. When these hardware mechanisms become broadly available, incorporation into TUD:OS would allow to fully encapsulate driver servers in their address spaces and thereby allow to freely assign physical devices to arbitrary untrusted guests.

Bibliography

[1] An Environment for L4 Applications. Available from URL: http://www.tudos.org/l4env/doc/l4env-concept/l4env.pdf. 18

[2] Dresden Real-Time Operating System Project Website, 2008. URL: http://www.tudos.org/drops/. 17

[3] Fiasco Microkernel Website, 2008. URL: http://www.tudos.org/fiasco/. 1, 17

[4] L^{4}Linux Website, 2008. URL: http://www.tudos.org/L4/LinuxOnL4/. 19

[5] Linux Website, 2008. URL: http://www.linux.org/. 19

[6] Darren Abramson, Jeff Jackson, Sridhar Muthrasanallur, Gil Neiger, Greg Regnier, Rajesh Sankaran, Ioannis Schoinas, Rich Uhlig, Balaji Vembu, and John Wiegert. Intel Virtualization Technology for Directed I/O. *Intel Technology Journal*, 10(3), August 2006. Available from URL: http://download.intel.com/technology/itj/2006/v10i3/v10-i3-art02.pdf. 54

[7] Keith Adams and Ole Agesen. A Comparison of Software and Hardware Techniques for x86 Virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 2–13, New York, NY, USA, 2006. ACM Press. Available from URL: http://www.vmware.com/pdf/asplos235_adams.pdf. 13

[8] Advanced Micro Devices. AMD64 Virtualization Codenamed "Pacifica" Technology Secure Virtual Machine Architecture Reference Manual, 2005. Available from URL: http://www.mimuw.edu.pl/~vincent/lecture6/sources/amd-pacifica-specification.pdf. 13, 54

[9] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating System Principles (SOSP)*, pages 164–177, Bolton Landing, NY, USA, October 2003. Available

from URL: http://www.cl.cam.ac.uk/research/srg/netos/papers/2003-xensosp.pdf. 12, 21

[10] Robert Baumgartl, Martin Borriss, Hermann Härtig, Claude-Joachim Hamann, Michael Hohmuth, Lars Reuther, Sebastian Schönberg, and Jean Wolter. Dresden Realtime Operating System. In *Proceedings of the First Workshop on System Design Automation (SDA'98)*, pages 205–212, Dresden, March 1998. 17

[11] Robert Baumgartl, Martin Borriss, Hermann Härtig, Michael Hohmuth, and Jean Wolter. Linux-Portierung auf den Mikrokern L4. *Wissenschaftliche Beiträge zur Informatik*, (1), 1996. 19

[12] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX 2005 Annual Technical Conference, FREENIX Track*, April 2005. Available from URL: http://www.usenix.org/publications/library/proceedings/usenix05/tech/freenix/full_papers/bellard/bellard.pdf. 12, 20

[13] Fabrice Bellard. QEMU Website, 2008. URL: http://fabrice.bellard.free.fr/qemu/. 20

[14] Sebastian Biemuller. Hardware-Supported Virtualization for the L4 Microkernel. Master's thesis, Universitat Karlsruhe, Karlsruhe, Germany, September 2006. Available from URL: http://i30www.ira.uka.de/teaching/thesisdocuments/l4ka/2006/biemueller06l4vcensored.pdf. 2

[15] Martin Borriss, Michael Hohmuth, Jean Wolter, and Hermann Härtig. Portierung von Linux auf den μ-Kern L4. In *Int. wiss. Kolloquium*, Ilmenau, September 1997. Available from URL: http://os.inf.tu-dresden.de/papers_ps/ilmenau.ps. 19

[16] Compaq Computer Corporation, Phoenix Technologies Ltd., and Intel Corporation. Plug And Play BIOS Specification, Version 1.0A, 1994. Available from URL: ftp://download.intel.com/support/motherboards/desktop/sb/pnpbiosspecificationv10a.pdf. 7

[17] Uwe Dannowski, Joshua LeVasseur, Espen Skoglund, and Volkmar Uhlig. L4 experimental kernel reference manual, version x.2. Technical report, University of Karlsruhe, 2004. Available from URL: http://l4hq.org/docs/manuals/. 26, 32

[18] Keir Fraser. Progressive paravirtualization. Available from URL: http://xen.org/files/summit_3/xen-pv-drivers.pdf. 14

[19] Keir Fraser, Steven Hand, Ian Pratt, and Andrew Warfield. Safe Hardware Access with the Xen Virtual Machine Monitor. In *Proceedings of the 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS 2004)*,

Boston, MA, USA, October 2004. Available from URL: http://www.cl.cam.ac.uk/research/srg/netos/papers/2004-oasis-ngio.pdf. 21

[20] Robert Philip Goldberg. Survey of Virtual Machine Research. *IEEE Computer Magazine*, 7(6):34–45, June 1974. Available from URL: http://pages.cs.wisc.edu/~stjones/proj/vm_reading/goldberg-vm-survey.pdf. 11

[21] Claude-Joachim Hamann, Robert Baumgartl, Martin Borriss, Hermann Härtig, and Lars Reuther. Dresden Real Time Operating System — ein Überblick. *Wissenschaftliche Beiträge zur Informatik*, (1):5–14, October 1997. 17

[22] Christian Helmuth. L4Env Generic I/O Reference Manual, 2008. URL: http://www.tudos.org/l4env/doc/html/generic_io/. 18

[23] Christian Helmuth, Alexander Warg, and Norman Feske. Mikro-SINA — Hands-on Experiences with the Nizza Security Architecture. In *Proceedings of the D.A.CH Security 2005*, Darmstadt, Germany, March 2005. Available from URL: http://os.inf.tu-dresden.de/papers_ps/dach2005.pdf. 1

[24] Christian Helmuth, Andreas Westfeld, and Michael Sobirey. μSINA - Eine mikrokernbasierte Systemarchitektur für sichere Systemkomponenten. In *Deutscher IT-Sicherheitskongress des BSI*, volume 8 of *IT-Sicherheit im verteilten Chaos*, pages 439–453. Secumedia-Verlag Ingelsheim, May 2003. Available from URL: http://os.inf.tu-dresden.dehttp//os.inf.tu-dresden.de/~westfeld/publikationen/bsi03.pdf. 1, 17

[25] Hewlett-Packard Corporation, Intel Corporation, Microsoft Corporation, Phoenix Technologies Ltd., and Toshiba Corporation. Advanced Configuration and Power Interface Specification, Revision 3.0b, 2006. Available from URL: http://www.acpi.info/DOWNLOADS/ACPIspec30b.pdf. 7, 11

[26] Michael Hohmuth. Linux-Emulation auf einem Mikrokern. Master's thesis, Technische Universität Dresden, Dresden, Germany, August 1996. Available from URL: http://os.inf.tu-dresden.de/~hohmuth/prj/linux-on-l4/diplom.ps.gz. 19

[27] Michael Hohmuth. The Fiasco Kernel: Requirements Definition. Technical Report TUD–FI–12, Technische Universität Dresden, Dresden, Germany, December 1998. Available from URL: http://os.inf.tu-dresden.de/papers_ps/fiasco-spec.ps.gz. 17

[28] Michael Hohmuth, Michael Peter, Hermann Härtig, and Jonathan S. Shapiro. Reducing TCB size by using untrusted components — small kernels versus virtual-machine

monitors. In *Proceedings of the Eleventh ACM SIGOPS European Workshop*, Leuven, Belgium, September 2004. Available from URL: `http://os.inf.tu-dresden.de/papers_ps/reducingtcb.pdf`. 2

[29] Lukas Hähnel. ACPI for L4Env. Großer beleg (undergraduate thesis), Technische Universität Dresden, Dresden, Germany, June 2007. Available from URL: `http://os.inf.tu-dresden.de/papers_ps/haenel-beleg.pdf`. 53

[30] Hermann Härtig. Security Architectures Revisited. In *Proceedings of the Tenth ACM SIGOPS European Workshop*, Saint-Emilion, France, September 2002. Available from URL: `http://os.inf.tu-dresden.de/papers_ps/secarch.pdf`. 1, 51

[31] Hermann Härtig, Robert Baumgartl, Martin Borriss, Claude-Joachim Hamann, Michael Hohmuth, Frank Mehnert, Lars Reuther, Sebastian Schönberg, and Jean Wolter. DROPS: OS Support for Distributed Multimedia Applications. In *Proceedings of the 8th ACM SIGOPS European Workshop*, Sintra, Portugal, September 1998. Available from URL: `http://os.inf.tu-dresden.de/papers_ps/sintra98.ps`. 17

[32] Hermann Härtig, Michael Hohmuth, Norman Feske, Christian Helmuth, Adam Lackorzynski, Frank Mehnert, and Michael Peter. The Nizza Secure-System Architecture. In *Proceedings of the 1st International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom 2005)*, San Jose, CA, USA, December 2005. Available from URL: `http://os.inf.tu-dresden.de/papers_ps/nizza.pdf`. 1, 17

[33] Hermann Härtig, Michael Hohmuth, and Jean Wolter. Taming Linux. In *5th Annual Australasian Conference on Parallel And Real-Time Systems (PART)*, Adelaide, Australia, September 1998. Available from URL: `http://os.inf.tu-dresden.de/papers_ps/part98.ps`. 40

[34] Hermann Härtig, Jörk Löser, Frank Mehnert, Lars Reuther, Martin Pohlack, and Alexander Warg. An I/O Architecture for Microkernel-Based Operating Systems. Technical Report TUD-FI03-08-Juli-2003, Technische Universität Dresden, Dresden, Germany, July 2003. Available from URL: `http://os.inf.tu-dresden.de/papers_ps/tr-ioarch-2003.pdf`. 51

[35] Innotek GmbH. Innotek GmbH Website, 2008. URL: `http://www.virtualbox.org/`. 20

[36] Intel Corporation. PCI IDE Controller Specification, Revision 1.0, 1994. Available from URL: `http://suif.stanford.edu/~csapuntz/specs/pciide.ps`. 38

[37] Intel Corporation. Programming Interface for Bus Master IDE Controller, Revision 1.0, 1994. Available from URL: http://suif.stanford.edu/~csapuntz/specs/idems100.ps. 10, 38

[38] Intel Corporation. Intel 440FX PCISet 82441FX PCI And Memory Controller (PMC) And 82442FX Data Bus Accelerator (DBX), 1996. Available from URL: http://download.intel.com/design/chipsets/datashts/29054901.pdf. 6, 35

[39] Intel Corporation. 82371FB (PIIX) And 82371SB (PIIX3) PCI ISA IDE XCELERATOR, 1997. Available from URL: http://download.intel.com/design/intarch/datashts/29055002.pdf. 38

[40] Intel Corporation. Intel 82801CA I/O Controller Hub 3-S (ICH3-S), Datasheet, 2002. Available from URL: http://download.intel.com/design/chipsets/e7500/datashts/29073303.pdf. 38

[41] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2B: Instruction Set Reference, N-Z, 2007. Available from URL: http://download.intel.com/design/processor/manuals/253667.pdf. 40

[42] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1, 2007. Available from URL: http://download.intel.com/design/processor/manuals/253668.pdf. 10

[43] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2, 2007. Available from URL: http://download.intel.com/design/processor/manuals/253669.pdf. 13

[44] Intel Corporation. Intel I/O Controller Hub 8 (ICH8) Family, Datasheet, 2007. Available from URL: http://download.intel.com/design/chipsets/datashts/31305603.pdf. 38

[45] Intel Corporation. Intel Virtualization Technology for Directed I/O, Architecture Specification, Revision 1.1, September 2007. Available from URL: http://download.intel.com/technology/computing/vptech/Intel(r)_VT_for_Direct_IO.pdf. 54

[46] Intel Corporation. Mobile Intel 965 Express Chipset Family, Revision 003, 2007. Available from URL: http://download.intel.com/design/mobile/datashts/31627303.pdf. 8

[47] Adam Lackorzynski. L^{4}Linux Porting Optimizations. Master's thesis, Technische Universität Dresden, Dresden, Germany, March 2004. Available from URL: http://os.inf.tu-dresden.de/papers_ps/adam-diplom.pdf. 19

[48] Jochen Liedtke. Improving IPC by Kernel Design. In *Proceedings of the 14th ACM Symposium on Operating System Principles (SOSP-14)*, pages 175–188, Asheville, NC, USA, December 1993. Available from URL: http://l4ka.org/publications/1993/improving-ipc.pdf. 17

[49] Jochen Liedtke. L4 reference manual (486, Pentium, PPro). Arbeitspapiere der GMD No. 1021, GMD — German National Research Center for Information Technology, Sankt Augustin, September 1996. Also Research Report RC 20549, IBM T. J. Watson Research Center, Yorktown Heights, NY, September 1996., September 1996. Available from URL: http://os.inf.tu-dresden.de/L4/l4refx86.ps.gz. 17

[50] Jochen Liedtke. μ-kernels must and can be small. In *5th International Workshop on Object Orientation in Operating Systems (IWOOOS)*, pages 152–155, Seattle, WA, USA, October 1996. Available from URL: http://l4ka.org/publications/1996/ukernels-must-be-small.pdf. 17

[51] Jörk Löser and Michael Hohmuth. Omega0 — a portable interface to interrupt hardware for L4 systems. In *Proceedings of the First Workshop on Common Microkernel System Platforms*, Kiawah Island, SC, USA, December 1999. Available from URL: http://os.inf.tu-dresden.de/~hohmuth/prj/omega0.ps.gz. 18

[52] Aravind Menon, Jose Renato Santos, Yoshio Turner, G. (John) Janakiraman, and Willy Zwaenepoel. Diagnosing Performance Overheads in the Xen Virtual Machine Environment. In *Proceedings of the 1st International Conference on Virtual Execution Environments*, pages 13–23, Chicago, IL, USA, June 2005. ACM Press. Available from URL: http://www.usenix.org/events/vee05/full_papers/p13-menon.pdf. 13

[53] Microsoft Corporation. PCI IRQ Routing Table Specification, 1996. Available from URL: http://www.microsoft.com/whdc/archive/pciirq.mspx. 11

[54] Microsoft Corporation. Server Virtualization Website, 2008. URL: http://www.microsoft.com/windowsserver2008/virtualization/default.mspx. 21

[55] Microsoft Corporation. Virtual PC Website, 2008. URL: http://www.microsoft.com/windows/products/winfamily/virtualpc/default.mspx. 20

[56] Gil Neiger, Amy Santoni, Felix Leung, Dion Rodgers, and Rich Uhlig. Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization. *Intel Technology Journal*, 10(3), August 2006. Available from URL: http://download.intel.com/technology/itj/2006/v10i3/v10-i3-art01.pdf. 13

[57] PCI Special Interest Group. PCI Express Specification, Revision 1.1. 9, 28, 38

[58] PCI Special Interest Group. PCI Local Bus Specification, Revision 3.0, 2004. 8, 11, 29, 36

[59] Phoenix Technologies Ltd. Standard BIOS 32-bit Service Directory Proposal, 1993. Available from URL: http://www.phoenix.com/NR/rdonlyres/ECF22CEC-A1B2-4F38-A7F9-629B49E1DCAB/0/specsbios32sd.pdf. 9

[60] Gerald J. Popek and Robert Philip Goldberg. Formal Requirements for Virtualizable Third Generation Architectures. *Communications of the ACM*, 17(7):412–421, July 1974. Available from URL: http://delivery.acm.org/10.1145/370000/361073/p412-popek.pdf?key1=361073&key2=7884556021&coll=GUIDE&dl=,acm&CFID=15151515&CFTOKEN=6184618. 13

[61] Qumranet, Inc. KVM: Kernel-based Virtualization Driver, 2006. Available from URL: http://www.linuxinsight.com/files/kvm_whitepaper.pdf. 20

[62] Qumranet, Inc. Qumranet, Inc. Website, 2008. URL: http://www.qumranet.com/. 20

[63] John Scott Robin and Cynthia E. Irvine. Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor. In *Proceedings of the 9th USENIX Security Symposium*, pages 129–144, Denver, CO, USA, August 2000. USENIX Association. Available from URL: http://www.cs.nps.navy.mil/people/faculty/irvine/publications/2000/VMM-usenix00-0611.pdf. 13

[64] Udo A. Steinberg. Fiasco μ-Kernel User-Mode Port. Großer beleg (undergraduate thesis), Technische Universität Dresden, Dresden, Germany, December 2002. Available from URL: http://os.inf.tu-dresden.de/papers_ps/steinberg-beleg.pdf. 17

[65] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proceedings of the 2001 USENIX Annual Technical Conference*, Boston, MA, USA, June 2001. USENIX Association. Available from URL: http://pages.cs.wisc.edu/~stjones/proj/vm_reading/vmware_usenix.pdf. 15

[66] T13. AT Attachment with Packet Interface - 7 Volume 1 - Register Delivered Command Set, Logical Register Set (ATA/ATAPI-7 V1), 2004. 38

[67] VMware, Inc. VMI Specification, Paravirtualization API Version 2.5, February 2006. Available from URL: `http://www.vmware.com/pdf/vmi_specs.pdf`. 13

[68] VMware, Inc. VMware, Inc. Website, 2008. URL: `http://www.vmware.com/`. 20

[69] Carl J. Young. Extended Architecture and Hypervisor Performance. In *Proceedings of the Workshop on Virtual Computer Systems*, pages 177–183, New York, NY, USA, 1973. ACM Press. Available from URL: `http://www.cs.ubc.ca/~norm/cs538a/p177-young.pdf`. 13