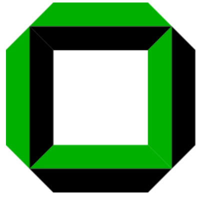

Universität Karlsruhe (TH)
Institut für
Betriebs- und Dialogsysteme
Lehrstuhl Systemarchitektur

Hardware virtualization support for Afterburner/L4

Martin Bäuml

Studienarbeit

Verantwortlicher Betreuer: Prof. Dr. Frank Bellosa
Betreuender Mitarbeiter: Dipl.-Inf. Jan Stöß

4. Mai 2007

Hiermit erkläre ich, die vorliegende Arbeit selbständig verfaßt und keine anderen als die angegebenen Literaturhilfsmittel verwendet zu haben.

I hereby declare that this thesis is a work of my own, and that only cited sources have been used.

Karlsruhe, den 4. Mai 2007

____

Martin Bäuml

Abstract

Full virtualization of the IA32 architecture can be achieved using hardware support. The L4 microkernel has been extended with mechanisms to leverage Intel's VT-x technology. This work proposes a user level virtual machine monitor that complements L4's virtualization extensions and realizes microkernel-based full virtualization of arbitrary operating systems. A prototype implementation within the Afterburner framework demonstrates the approach by successfully booting a current Linux kernel.

Contents

1 Introduction **5**

2 Background and Related Work **6**
- 2.1 Intel VT-x . . . 6
- 2.2 Virtualization Hardware Support for L4 . . . 7
- 2.3 Afterburner Framework . . . 8
- 2.4 Xen . . . 8

3 Design **9**
- 3.1 Virtualization environment . . . 9
- 3.2 Nested Memory Translation . . . 11
- 3.3 Privileged Instructions . . . 14
- 3.4 Interrupts and Exceptions . . . 15
  - 3.4.1 Event Source and Injection . . . 15
- 3.5 Devices . . . 17
- 3.6 Boundary Cases . . . 17

4 Implementation **19**
- 4.1 Integration into Afterburner Framework . . . 19
  - 4.1.1 Resource Monitor . . . 19
  - 4.1.2 Device Models . . . 20
- 4.2 The Monitor Server . . . 20
  - 4.2.1 Virtualization Fault Processing . . . 20
- 4.3 Interrupts . . . 22
- 4.4 Guest Binary Modifications . . . 22

5 Evaluation **23**
- 5.1 Booting . . . 23
- 5.2 Network Performance . . . 25
- 5.3 Webserver Performance . . . 25

Chapter 1

Introduction

With the performance advancements of processors, virtualization has become applicable to personal computers and small servers. On recent processor generations the performance overhead due to virtualization is more than acceptable. This development led to widely-used applications like server consolidation, system isolation and migration.

A virtual machine monitor (VMM) can run on bare hardware, without a full operating system supporting it. Such a VMM, also called *hypervisor*, has full control over the hardware rather than going through abstractions provided by the operating system. Therefore, a hypervisor can optimize for performance and improve reliability. That is, the VMM cannot crash because of bugs e.g. in the Linux kernel, but only depends on a correct implementation of the VMM itself. In other words, the trusted code base is minimized to the VMM.

A hypervisor is much like an microkernel. It is a thin software layer on top of the hardware and provides a clean interface to the next software layer above. In the case of a hypervisor, the interface is a subset of the Instruction Set Architecture, rather than a set of system calls. Both provide abstractions and mechanisms for execution entities, isolation and communication. Based on the thesis that microkernels and hypervisors are similar enough to justify an integration of both, the L4 microkernel [7] was extended [4] to support hardware virtualization extensions like Intel VT-x [11].

The goal of this thesis is to detail the design and functionality of a user level VMM on top of the L4 microkernel leveraging L4's hardware virtualization support. It will be integrated in the already existing Afterburner framework, a set of servers and device models targeted for pre-virtualization [6] on top of L4. The resulting VMM will be able to run an unmodified Linux guest.

This thesis is organized as follows: The next chapter gives a short introduction into Intel's virtualization hardware extensions, the extensions made to L4 and some related work. The third chapter presents the design and functionality of the user level VMM. The last two chapters give details about the implementation of the VMM within the Afterburner framework and present some performance results.

Chapter 2

Background and Related Work

In this chapter I will first give a brief introduction to the Intel VT-x extensions which allow full virtualization of the IA-32 architecture. Section 2.2 is dedicated to the extensions made to the L4 microkernel which provide abstractions and protocols for hardware virtualization support. The next section is a short overview over the Afterburner framework in which the implementation of this thesis will be integrated. In the last section I present Xen, a popular open source hypervisor, which supports Intel VT-x extensions in its most recent release 3.0.

2.1 Intel VT-x

The IA-32 architecture has never been efficiently virtualizable according to the formal requirements introduced by Popek et al. [8]. One of those requirements demands that all virtualization sensitive instructions are a subset of privileged instructions. This is not the case for example for the instructions `MOV from GDT` or `POPF`. Although it is possible to build VMMs upon IA-32 using sophisticated virtualization techniques like para-virtualization (used by Xen [9]) or binary-translation (used by VMWare [13]), it is payed by either performance and/or high engineering costs. They also suffer from deficiencies like ring compression, which means that the guest OS runs at another privilege level than it was designed for (e.g. ring 3 instead of ring 0). The main design goal for the Intel VT-x extensions was to eliminate the need for sophisticated virtualization techniques like para-virtualization or binary translation and make efficient virtualization of the IA-32 architecture possible [11].

Intel VT-x introduces two new execution modes, VMX root mode and VMX non-root mode. VMX root mode is comparable to IA-32 without VT-x. A virtual machine monitor running in VMX root mode can configure the CPU to fault on every privileged instruction of code running in VMX non-root mode. Every fault causes a transition from non-root mode to root mode. This transition is called *VM Exit*. The VMM can determine the reason of the exit by the value of the *basic exit reason* register. It can also access and modify all guest state (registers, flags etc.). The VMM can therefore emulate the privileged instruction, update the guest state and resume guest execution by reentering

non-root mode.

Both root mode and non-root mode contain all four privilege levels (i.e., rings 0, 1, 2, 3). Therefore deficiencies like ring compression can be overcome because the guest OS can run at the privilege level it was designed for. Then the guest can for example efficiently use the low latency system call instructions `SYSENTER` and `SYSEXIT`, which would otherwise cause expensive traps into the VMM.

Intel VT-x provides also support for managing guest and host state between transitions and mechanisms for efficient event injection.

2.2 Virtualization Hardware Support for L4

The L4 microkernel is a second generation microkernel. It provides abstractions for address spaces, threads and IPC. Biemüller suggests in [4] a small set of extensions to L4 to allow user level threads to leverage hardware virtualization support. According to the minimality principal of microkernel design only those parts were integrated into the kernel that could not be realized by a user level server, or prevent the system from being usable, if implemented outside the kernel. The extensions undertake four fundamental tasks that need to be addressed in a Intel VT-x based hypervisor:

Provision of execution entities and isolation containers L4 already provides threads and address spaces as primary abstractions for execution and isolation. The virtual machine model therefore maps each guest OS to one address space, and each virtual CPU to a thread within this address space. These abstractions are extended minimally that the kernel knows whether an address space caters for a virtual machine. Thus the kernel can resume each thread in this address space by a VM Resume instead of returning to user level.

Management of VMCS structures Intel VT-x introduces Virtual Machine Control Structures (VMCS). A VMCS is a data structure where guest and host state are kept when the CPU is in VMX root mode or VMX non-root mode, respectively. The kernel is responsible for allocating and managing the VMCS structures transparently for user level VMMs. A VMM can access relevant portions of the guest state through the virtualization protocol (see below).

Dispatching/Virtualization Protocol For most VM Exits L4 dispatches the handling of the exit to a user level server. The kernel communicates with the user level VMM using the *Virtualization Protocol* which is based on IPC. Most VM Exits require intervention of the user level VMM. Therefore the kernel sends a virtualization fault message, similar to a page fault message, to the VMM on behalf of the guest. The message contains the reason for the exit and some additional guest state. The exact contents of the message can be configured by the virtualization fault handler on a per-exit-reason basis. The VMM also uses IPC to reply with state modifications and eventually to resume the guest thread.

Shadow pagetables The guest OS does not have access to real physical memory. Instead it runs on virtual memory provided by its VMM (which is

always also its pager) in an L4 address space, which the guest sees as physical memory. The kernel performs the additional translation step between real physical memory, L4 virtual memory and guest virtual memory. When the guest is running, the kernel installs a modified version of the guest's page table, a *shadow pagetable* or *virtual TLB (vTLB)*, which contains mappings from real physical memory to guest virtual memory. Shadow pagetables are explained in more detail in section 3.2.

2.3 Afterburner Framework

The Afterburner framework [5] provides a set of tools and servers to support virtualization on L4. The original main target was to support a technique called *pre-virtualization* or *afterburning*. Pre-virtualization uses a semi-automated approach to prepare a guest binary for virtualization. The approach is somewhat similar to para-virtualization, but works at assembler level, allows the prepared binary to run on both bare hardware and a virtual machine monitor and reduces the engineering effort of preparing a binary by about one order of magnitude [6].

Although the framework contains an increasing number of L4 servers and virtual device models to support the construction of a VMM, it lacks the possibility to run a unmodified guest OS. Modification is not always possible, because there are operating systems whose source code and therefore assembler code is not available (e.g. Microsoft Windows).

2.4 Xen

Xen [9] is a popular open source hypervisor. In early revisions Xen only supported para-virtualized guests. With assistance of Intel VT-x, Xen 3.0 now also provides virtualization for unmodified OSs. Xen consists of a privileged hypervisor (running in ring 0, and in VMX root mode while using Intel VT-x extensions) and a user level component. The user level component (called *Dom0*) is a para-virtualized Linux that has passthrough access to the machines hardware. It uses Linux device drivers for hardware access and provides virtual device models for disks, network cards etc. to other guests. A modified version of the full system emulators QEMU and Bochs provide emulation of the whole PC platform. One process of QEMU or Bochs runs in Dom0 for each guest and emulates guest IO accesses. The guest domains communicate with Dom0 via shared memory which is set up as ring buffers. Dom0 also runs VM management and configuration utilities.

The main differences between Xen's approach and ours are that Xen's hypercall interface is designed for virtual machines only and not generic enough to build arbitrary light-weight systems on top of the hypervisor and that Xen keeps far more logic (for example the emulation of heavily used virtual devices like programmable interrupt controllers) in the privileged part of the VMM.

Chapter 3

Design

A generic virtual machine monitor can be divided into three parts: the *dispatcher*, the *allocator* and the *interpreter* [8]. The dispatcher is the main entry point for exits from the virtual machine. The allocator allocates and manages resources. It also ensures that different virtual machines do not access the same resource in an uncontrolled way. The interpreter emulates unsafe instructions and commits resulting state changes back to the virtual machine. In our case, the kernel already implements the dispatcher by sending virtualization IPC messages on behalf of the guest. Therefore, the user level monitor needs to implement allocator and interpreter.

The user level monitor complements the extensions for hardware virtualization support made to the L4 microkernel. It controls and multiplexes accesses to physical resources using abstractions and mechanisms provided by the kernel. In particular, it is the major IPC endpoint for virtualization messages sent by the microkernel. It also serves as the virtual machine's pager and provides mappings for the guest physical memory.

In this chapter I will propose a design for the user level VMM. Section 3.1 will give an overview over the VMM server and its address space layout. In section 3.2 I will explain in detail how the kernel and the VMM perform nested memory translation. In section 3.3 I will analyse which instructions need to be handled by the VMM, and the last three sections deal with interrupts, exceptions, devices and boundary cases.

3.1 Virtualization environment

The virtualization software stack consists of a set of L4 servers. In cooperation with the kernel they provide the necessary services to host virtual machines. Figure 3.1 shows the basic architecture of the software stack.

Resource monitor The resource monitor is the root server of the monitor instances (*VMM1* and *VMM2* in figure 3.1). It manages all physical resources available to the virtual machines. During system bootup it creates a monitor server for each virtual machine and reserves for it the requested amount of memory, if available. On later request by the monitor server, the resource monitor provides mappings for physical device access in such

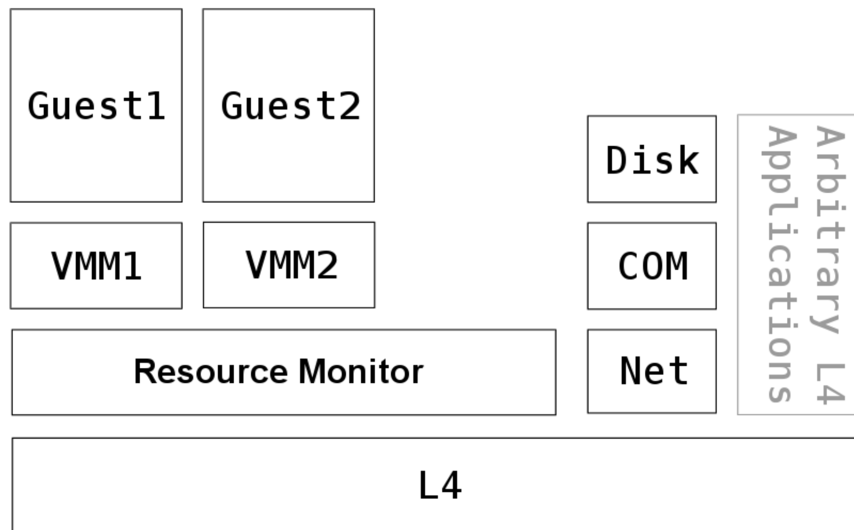

Figure 3.1: Overview over the virtualization environment architecture.

way that just one guest at the same time has access to a specific physical device.

Monitor server The monitor constructs and manages the virtual machine environment. It creates an L4 hardware virtualized address space as isolation container for one guest. In this address space it initializes one thread per virtual CPU (VCPU). The monitor allocates enough memory for the mapping of guest physical memory and loads necessary binaries for the bootup process to the guest physical memory. The monitor also serves as the virtual machine's pager and provides mappings on page faults on the guest physical memory.

Each monitor instance caters for exactly one virtual machine. The guest's physical address space is identity-mapped into the monitor's address space starting at address 0x0. The size of the guest physical memory can be configured at load time of the monitor. The monitor's code is outside the guest physical memory region, so that the guest can not interfere with the monitor. If the guest is granted passthrough access to memory mapped IO, the monitor needs to map the IO pages to a safe device memory area inside its own address space, before it can map it on to the guest. See figure 3.2 for an overview of the address space layout.

Although each monitor instance only holds one virtual machine, it is still possible to run multiple virtual machines in parallel by running multiple monitor instances. This way each virtual machine is securely isolated using L4's address space protection.

Device Driver A set of device driver servers provides access to the machines hardware.

This architecture allows to build arbitrary applications next to the virtualization stack. They can either be completely independant from the virtualiza-

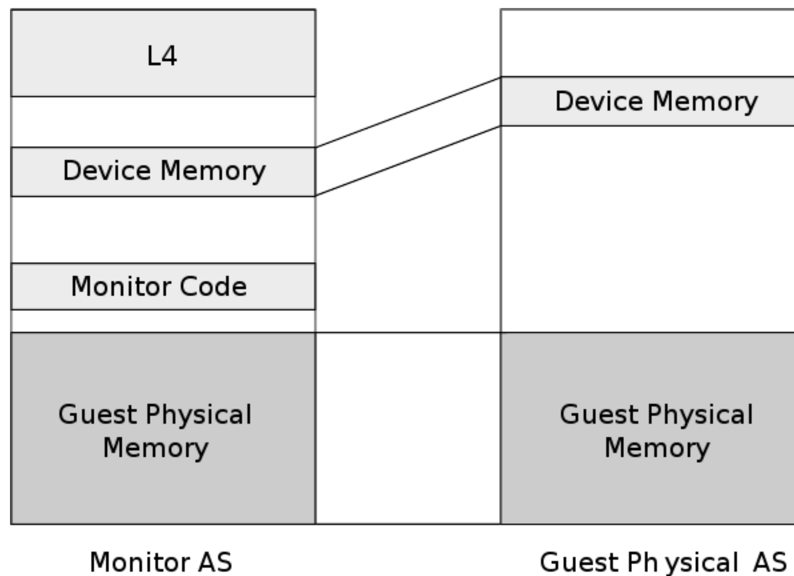

Figure 3.2: The monitor's address space contains a one-to-one mapping of the guest physical address space. Device memory might be mapped to a safe address region in the monitor.

tion environment or for example use services provided by some legacy software running inside a virtual machine.

3.2 Nested Memory Translation

In this section I will explain how the L4 kernel virtualizes guest physical memory in collaboration with the user level monitor.

In a virtual machine environment, the guest OS cannot have direct access to physical memory, because the hardware does not support fine grained protection of physical memory. Instead, the guest physical memory is abstracted by an L4 address space, for which hardware guarantees protection. The monitor uses L4's mapping mechanisms to populate the guest's address space with guest physical memory. Because the guest expects to be on a real machine, it will implement virtual address spaces itself on top of the guest physical memory. On the other hand the kernel cannot allow the guest to manipulate the kernel's page table for security reasons. Unfortunately, current hardware does not (yet) support such a nested virtual memory hierarchy. So the illusion of a nested memory translation has to be provided by the kernel. Therefore, the kernel uses the Intel VT-x extensions to make the guest trap on all page table related events, that is, *pagefaults*, `MOV to CR3` and `INVLPG`, as well as the rare case of flipping the `PG` bit in `CR0` (turning paged mode on or off). On each such event, the kernel modifies the *virtual TLB (vTLB)*, also called *shadow page table*, the real page table installed by the kernel and seen by the MMU. It contains the mappings to perform the correct and safe guest-virtual to host-physical address translation.

In the following I will cover each of the above page table related events

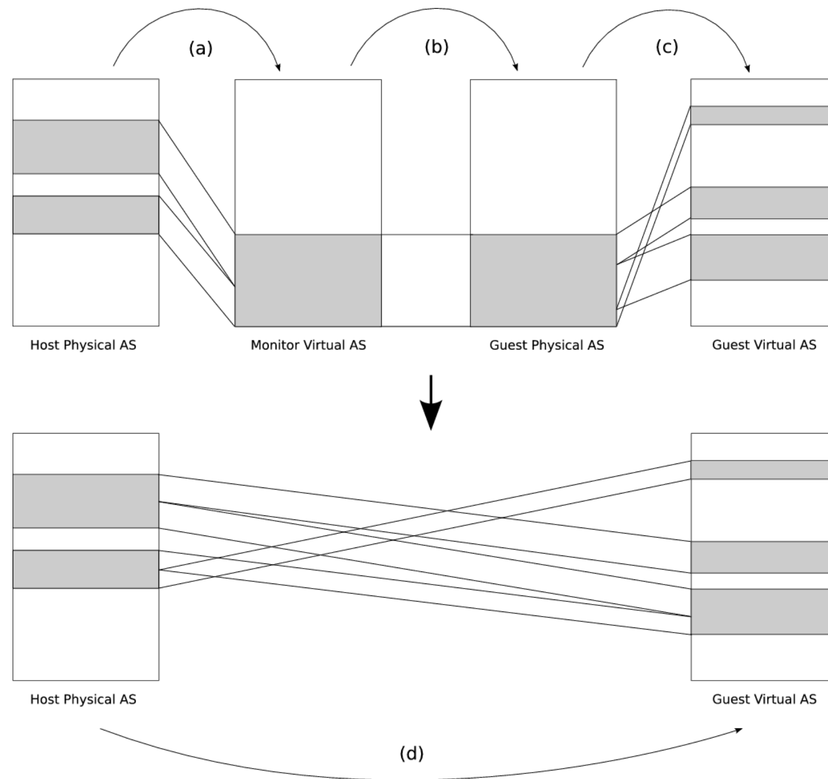

Figure 3.3: Nested memory translation using shadow page tables. **(a)** Virtual memory translation generated by L4's page table for the monitor's address space. **(b)** Mapping from monitor's address space to the guest's L4 address space. **(c)** Virtual memory translation as defined in the guest's *current* page table. **(d)** The vTLB combines translation steps a, b and c into one L4 page table, generating the illusion of nested memory translation.

and instructions in more detail. See also figure 3.3 for an overview over nested memory translation using shadow page tables.

Pagefault on Guest Virtual Memory A guest pagefault is by far the most frequent event to trigger a vTLB update. It is raised when the guest tries to access a virtual address which has no mapping in the vTLB or only insufficient access rights. Although this is true for all guest page faults, because the vTLB is *the* actual page table which is used by the MMU for address translation, we can distinguish several causes, why the vTLB does not contain a valid mapping. The kernel determines the exact reason by parsing the guest's page table, and potentially the vTLB, too. We can distinguish the following cases:

1. The guest page table does not contain a valid mapping, or only a mapping with insufficient access rights for the faulting address. This must be handled by the guest. Therefore the kernel injects the pagefault back into the guest, which can update its page table accordingly.

When the guest repeats the faulting instruction, another pagefault will be raised, because the vTLB still does not contain a valid mapping. Only in the uncommon case that the guest ignores the pagefault (e.g. because a user process accessed an invalid memory area), no second pagefault will be raised.

2. The guest's mapping points to a guest physical page, which is not mapped into the guest's address space. Such a pagefault does not exist on a real machine. Therefore the kernel and the monitor handle it transparently for the guest. L4 translates the pagefault into a page fault IPC to the monitor, which in return provides a mapping for the faulting guest physical address. The monitor can use flexpages of any size for the mapping, although larger mappings should be favored for efficiency reasons. The monitor can refrain from replying with a mapping for the emulation of memory mapped IO. For details see section 3.5.

3. The guest physical page has insufficient access rights for the guest access (e.g. a write on a read only page). This case occurs when the monitor implements a copy on write mechanism for guest physical memory [12]. Then the monitor makes a copy of the page before it grants the guest write access to the page.

4. If non of the above reasons apply, the relevant vTLB entry does not yet reflect the guest's and the monitors's mappings. Therfore, the mapping in the vTLB needs to be updated for the faulting virtual address. The kernel determines the associated guest physical page from the guest's page table and, using L4's mapping database, the host physical page frame which backs the guest physical page. It now updates the vTLB with a mapping from the guest virtual address to the host physical address, therefore providing the illusion of a nested memory translation. The access rights for the page are derived from the guest's page table. This also includes the kernel bit, which is important for the guest kernel's protection from its user processes. In two cases the kernel unsets the write bit, although it is set in the guest page table. Firstly, if the monitor implements a copy on write mechanism and only provides a read only mapping. Secondly, the vTLB also emulates the dirty bit in the guest page table. The MMU sets the dirty bit correctly in the vTLB, but the guest expects it to be set in its own page table on the first write access to the page. So if the first access to the page is a read, the kernel maps the page read only, so that the first write access to it raises another pagefault. On this pagefault, the kernel can finally update the dirty bit in the guest's page table. This is not necessary if the first access is a write access. The kernel then sets the dirty bit immediately.

MOV to CR3 A `MOV` to the control register `CR3` loads a new page table and implicitly invalidates all previous cached mappings. Because the guest must not have access to the real page table, the kernel traps on this instruction. The kernel reloads `CR3` itself with a new vTLB. If it implements a caching strategy, it might already have a cached, prepopulated vTLB for this guest page table. Caching promises to increase performance because each pre-

populated vTLB entry might save a pagefault. On the other hand, an elaborate mechanism is needed to detect changes to the guest page table, so that the vTLB does not reflect an old, now incorrect mapping.

To hide that it installed a different page table than the guest expects, the kernel ensures that the guest does not read the real `CR3` register. Intel VT-x provides a shadow `CR3` register, which content is returned on every `CR3` read by the guest.

INVLPG Changes to a page table are not guaranteed to take immediate effect because the hardware TLB caches mappings from the current page table. The `INVLPG` instruction is used to remove a cached mapping from the TLB, so that on the next access to the page the translation is re-read from the current page table. With the vTLB in place, the kernel not only invalidates the TLB by re-executing `INVLPG` on behalf of the guest but also removes the corresponding mapping from the vTLB. Otherwise the MMU would find the the old mapping in the vTLB, which would not reflect a possible new mapping in the guest page table. The kernel does not immediately translate the new guest mapping, because it might still change until the next page access.

Nested memory translation is implemented in the kernel mainly because of performance reasons. If implemented in the monitor, each guest pagefault would require the monitor to map the corresponding guest physical page from its own address space to the guest's address space at the requested virtual address. This additional mapping operation is considered too expensive [4]. Also, L4 does not support mapping of kernel pages (the kernel bit set), which would be needed to ensure protection for the guest OS. Thirdly, upcoming hardware promises to support nested paging in hardware, rendering software solutions obsolete but for older processors. On the other hand, the monitor needs to implement guest page table parsing and pagefault injection anyway to emulate IO string instructions such as `REPZ INSW`.

3.3 Privileged Instructions

The monitor handles privileged instructions in one of three possible ways. I will use the IO read instruction `INB` as an example in each case. `INB` reads a byte from an IO port to the `EAX` register.

Emulation The hardware traps on the instruction and the kernel transfer control to the monitor by sending a virtual fault message on behalf of the guest. The monitor uses the virtual fault message to determine the faulted instruction. It updates the state of the guest by emulating the instruction in a safe way. Example: The monitor notifies the virtual device for this port about the `INB` instruction. The virtual device returns the current value of the virtual IO port according to the device's current state. The monitor updates the guest's `EAX` register with the return value before it resumes guest execution. No real device is accessed during the emulation.

Execution on Behalf Analogously to emulation, the hardware traps on the privileged instruction and the monitor is notified through a virtual fault

message. The monitor makes sure that the parameters for the instruction are safe or adjusts them as needed. Then it executes the privileged instruction on behalf of the guest. The return value, if any, is verified to be safe before the monitor updates the guest with the new state. Example: The monitor executes the INB instruction itself, but might choose a different port number. That way, the monitor can grant passthrough access to a real device, but keeps complete control and can for example redirect `COM0` access to `COM1` transparently for the guest.

Passthrough The guest is allowed to execute the instruction without trapping. In this case the monitor has no longer control about the execution of the instruction nor does it get notified. To reenable complete control, the monitor can reactivate the hardware trap. Passthrough execution must only be allowed if isolation is not harmed. Example: Intel VT-x lets the monitor specify fine grained passthrough access rights to IO ports. If a device is assigned to a single virtual machine, the monitor can grant full access to the devices IO ports for performance reasons. A guest INB instruction (and every other IO instruction) on these ports executes without fault which saves the cost of a VM Exit and reentry. Access to all other IO ports still raises a fault.

3.4 Interrupts and Exceptions

The guest expects interrupts and exceptions to be delivered similar to native execution. As each guest has a different set of interrupt handlers, it implements its own interrupt descriptor table in its guest address space. The VMM divides interrupts and exceptions into two classes: those which are critical to host/VMM operation, and those which are not. All non-critical events will be directly delivered to the guest (i.e., do not raise a VM Exit), whereas all critical events are configured to raise a VM Exit. In the following there is a short overview which event belongs to which class:

Critical Events As already discussed in 3.2, the pagefault exception is used to virtualize guest physical memory and is therefore a critical event. External interrupts are critical, too, because the VMM cannot allow the guest to handle external interrupts directly for security, isolation and multiplexing reasons. They are handled by either L4 device drivers or by a generic interrupt server in the VMM. Although the debug registers are not critical to host operation, they can be used to set breakpoints in the guest for debugging the VMM and/or the guest. In this case, also the debug exception is configured to exit the guest.

Non-Critical Events All software-generated interrupts and all exceptions beside pagefaults can be handled directly by the guest and do not exit to the VMM.

3.4.1 Event Source and Injection

Although all critical events are handled solely by the kernel, L4 device drivers and the VMM, they can still be the source for virtual events for the guest. When

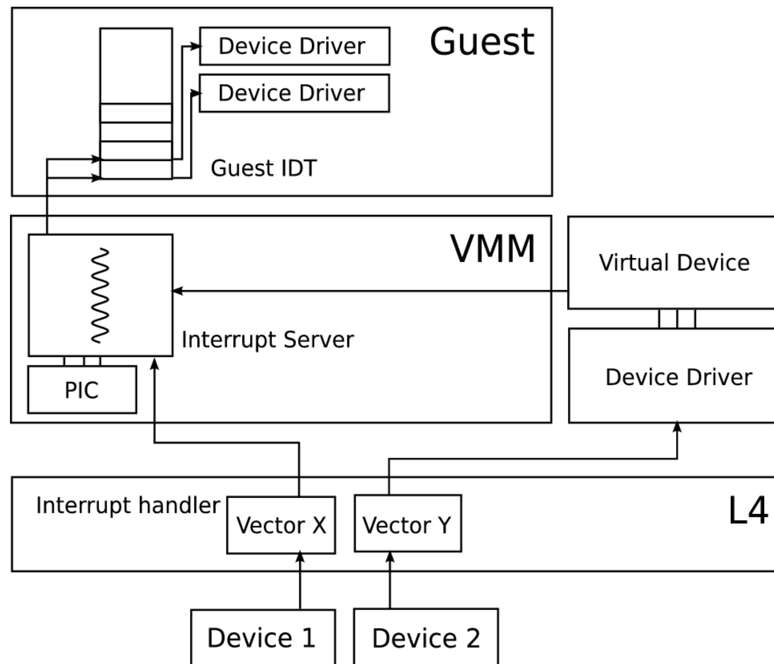

Figure 3.4: Interrupt injection.

the VMM has a pending event for the guest, it uses Intel VT-x's event injection mechanism to deliver it to the guest. We distinguish between interrupts and exceptions:

Interrupts Figure 3.4 illustrates the source and injection model of guest interrupts. The VMM runs a dedicated interrupt server which is responsible for receiving and injecting virtual interrupts. Interrupts are either generated by real devices, or by virtual device models. Real device interrupts are either received by a device driver (like vector Y), or by the interrupt server if the guest has passthrough access to the device (like vector X). For the latter, the VMM associates itself as the interrupt handler thread for L4 interrupt IPCs. It only grants this kind of interrupt passthrough when a device is exclusively assigned to one guest.

The VMM emulates two nested i8259 interrupt controllers which are standard programmable interrupt controllers (PICs). When the interrupt thread receives either a real interrupt or a virtual interrupt from a device model, it updates the PICs' state with the new interrupt pending. The interrupt server now notifies the guest about the event. If the guest is currently descheduled and waiting for interrupts, e.g. as a result of executing the `HLT` instruction, the VMM rewakes the guest by performing the interrupt injection. If the guest is on the other hand currently in an executing state (e.g. the last VM Exit only occured because of expiration of the time slice), it could have interrupts disabled. Therefore, the VMM uses Intel VT-x's *Interrupt Window Exit* mechanism to force a VM Exit on the next occasion when interrupts are allowed to be deliv-

ered (that is, the IF flag is set), without the need to poll the guest state. The VMM can then inject the pending interrupt on the next Interrupt Window fault. L4 exposes the Interrupt Window request feature by an extension to the `ExchangeRegisters` system call. The permission, that `ExchangeRegisters` can be called by any thread in the pagers address space, is granted in an experimental extension to the L4 API.

Exceptions A pagefault exception is the only critical exception which the VMM delivers to the guest. The common case is that the vTLB reinjects pagefaults into the guest whenever it determines that they are really pagefaults on guest virtual memory. Also the emulation of instructions by the VMM that access guest virtual memory is a source for virtual pagefaults. As the guest cannot mask out the reception of exceptions, the vTLB and the VMM immediately inject the pagefault into the guest with the next reentry.

Interrupt and exception handling in the guest procedes normally, i.e. the guest interrupt handlers are triggered without further intervention of the VMM.

3.5 Devices

A virtual machine only makes real sense if the guest has access to devices which it can use to communicate to the outside world. Therefore, the VMM should cater for at least a network card and a harddisk, either through emulation or by assigning a real device to the guest. The guest communicates with the devices using IO ports and/or memory mapped IO. Intel VT-x provides fine-grained access control to IO ports via bitmaps, so that an IO operation on a restricted port (one that has not been assigned to the guest) raises a VM Exit. The VMM then forwards the request to an appropriate device model which emulates the instruction. L4 supports this fine-grained access control through IO-Flexpages [10].

For emulation of memory mapped IO, the VMM uses pagefaults to determine access to device memory regions. Instead of mapping the page, it triggers device emulation. Even for guest assigned devices memory mapped IO might need to be trapped upon. This is due to DMA, which operates on physical addresses. Unless the VMM would intervene, the guest kernel would configure the DMA controller with physical addresses and therefore circumvent the MMU. But the guest physical addresses in almost all cases will not match its real physical addresses, so that 1. the DMA controller overwrites uninvolved memory regions and 2. the data never arrives at the guest. The VMM overcomes this by catching and modifying the DMA controller configuration, and possibly even emulate DMA as a whole.

3.6 Boundary Cases

Although we can expect that the guest runs most of the time in paged protected mode, the VMM provides support for real mode and unpaged protected mode as well. Neither is supported natively by Intel VT-x in VMX non-root mode. VMX non-root mode can only run in paged protected mode so that real mode and

unpaged protected mode are emulated by the monitor and/or kernel. Virtual-8086 mode is similar enough to real mode for using it to execute real mode code. The vTLB emulates unpaged protected mode by identity-mapping guest physical memory to guest virtual memory. The guest's shadow `CR0` register reflects the current execution mode, even if the guest is really running in paged protected mode.

During the execution of real mode code the monitor catches BIOS interrupts from the guest and emulates them properly.

Chapter 4

Implementation

In this chapter I will present some details about the implementation of the user level VMM and its integration into the Afterburner framework.

4.1 Integration into Afterburner Framework

The VMM was implemented as part of the Afterburner framework. The Afterburner framework already comes with a large code base from which some components could be reused for our implementation. Especially the resource monitor and some device models could be integrated with only small changes.

4.1.1 Resource Monitor

The resource monitor is an L4 root task. It manages all system resources and makes them available to virtual machines on request. It is designed to load an in-place VMM (the *wedge*) into the same address space as the guest binary. The configuration of the virtual machine environment can be done with command line arguments via the GRUB bootloader. By choosing these arguments carefully, we can make the resource monitor load our VMM correctly. We want to achieve:

- ELF-load the VMM (see 4.2) to a new address space.
- Make the guest binary accessible within that address space.
- Allocate a continuous chunk of memory (the amount is configurable) for the guest physical address space.
- Access additional supporting binaries such as ramdisk or a floppy/harddisk image.

The resource monitor loads multiple modules (including ramdisk and disk images) into one address space, if the first module's command line contains the paramenter `vmstart`. All modules until the next `vmstart` are placed into one address space, and the first one (which should be our VMM) is correctly ELF-loaded if it is an ELF file. The amount of available memory can be configured with `vmsize=`. This memory will be available starting at address 0x0. We want

to make sure, that the guest cannot access the VMM code later. Therefore, we make the resource monitor load it with an offset (`wedgeinstall=`) beyond the maximum guest physical address. A sample GRUB entry would therefore look like:

```
title=pistachio-vt afterburner-vt bootfloppy
 kernel (nd)/tftpboot/baeuml/vt/kickstart
 module (nd)/tftpboot/baeuml/vt/pistachio
 module (nd)/tftpboot/baeuml/vt/sigma0
 module (nd)/tftpboot/baeuml/vt/l4ka-resourcemon
 module (nd)/tftpboot/baeuml/vt/afterburn-wedge-l4ka-passthru \
                        vmstart vmsize=540M wedgeinstall=512M
 module (nd)/tftpboot/baeuml/floppy.img
```

4.1.2 Device Models

I reused the device models of a i8259a programmable interrupt controller, a i8253 serial port, a mc146818 real time clock and a i8253 programmable interval timer. The device models have a simple interface: they accept read and write access to their specific IO port ranges. Thus their integration consists merely of instantiating each device model and forwarding access to these IO ports to the corresponding model.

4.2 The Monitor Server

The monitor server is the VMM module that is loaded into a new address space by the resource monitor. After it is started as a new thread, it parses module information which the resource monitor relays to the monitor via a shared page. Then the monitor creates a new hardware virtualized address space and one thread as virtual CPU inside it. A multi-processor virtual machine environment is not supported so far. If the first module is a Linux kernel, it is loaded correctly as a Linux kernel: The monitor fills the kernel boot header according to the Linux x86 boot protocol, copies command line options and sets up a ramdisk (if present and loaded as another module by the resource monitor), before the thread is started at Linux's entry address. If the first module is a disk image, only its boot sector is loaded to memory and the VCPU is started in real mode. Finally, the monitor thread starts the interrupt server, which handles all incoming real and virtual interrupts. After all initalization is done, the monitor sends the startup virtualization message to the VCPU and enters a server loop, where all incoming virtualization fault messages are processed.

4.2.1 Virtualization Fault Processing

Each incoming virtualization fault message contains a *basic exit reason*. Based on its value, the nature of the fault can be determined and the proper fault handler can be called. For example:

HLT The basic exit reason indicates that the guest tried to execute the `HLT` instruction. The Linux kernel does this normally in the idle loop to shut down the processor until the next interrupt event. Therefore, the monitor

can safely deschedule the guest (by just not replying the virtualization fault) until the interrupt thread receives the next interrupt. On reception of the next interrupt the monitor sends a virtualization reply to the guest to make it runnable again. This reply also contains an element to trigger immediate interrupt delivery.

The implementation also deals with a small but subtle issue: Linux reenables interrupts by executing `STI` just before executing `HLT` in its idle loop. Therefore, interrupts are blocked by the `STI` instruction[1] until after executing the `HLT` instruction[2]. Since the guest traps on the privileged instruction `HLT` before executing it, the monitor has to disable this *blocking by STI* explicitly to make interrupt injection possible in this case ([2, Section 22.3.1.5])

IO access Virtualization of IO instructions has to be implemented carefully, because IO instructions can involve device access (real or virtual), guest virtual memory access, and can even trigger multiple port accesses until an exit condition is met. A single `INB` into a general purpose register is implemented quite straight forward (see section 3.3 for an overview over the implementation models). More care has to be taken on a (repeated) string IO instruction (e.g. `INS`) which implicitly operates on guest virtual memory[3]. The destination operand is a memory location defined by `ES:EDI`. A `REP` prefix can precede the string IO instruction to repeat it until `ECX` reaches 0 (`ECX` is decremented implicitly each iteration). In case of a conditional prefix (e.g. `REPNZ`: repeat while not zero) the instruction breaks out of the loop when the `ZF` flag meets a condition or `ECX` reaches 0, whichever comes first. On each repetition, EDI is implicitly incremented or decremented, depending on the `DF` flag. Now the VMM has to check on such an instruction if

- the exit condition is met
- `ES` contains a valid segment descriptor at all
- `ES:EDI` is a valid guest address.

The segment check can be done once in the beginning, while the current guest page table has to be checked more often since `EDI` changes implicitly. To avoid unneccessary effort it suffices to parse the guest page table on the first access and each time a page boundary is crossed. If the VMM discovers an invalid mapping in the guest page table, it injects a pagefault into the guest. Although not implemented in the VMM, the IA-32 architecture would allow delivery of interrupts during execution of such an instruction. See [1] for details on the behaviour of IA-32 string instructions.

[1] The `STI` instruction delays recognition of interrupts until the *next* instruction is executed [1, Section STI].

[2] Afterburner's pre-virtualization step removes this subtle semantic by replacing a `STI` by `STI NOP NOP...`, rather than `...NOP NOP STI`. I stumbled upon this when I found that a pre-virtualized kernel behaved differently than a non-modified kernel.

[3] I found that Afterburner's pre-virtualization step does not replace `INS` at all. Must have been overseen.

4.3 Interrupts

The interrupt server is the endpoint for interrupt messages by virtual and real devices. It keeps track of pending interrupts in device models of two nested i8259a programmable interrupt controllers. Timer interrupts are generated by using L4's timeout mechanism for the `IPC` system call. Each time the `IPC` system call times out, a timer event is raised. Under heavy load the interrupt server IPC call might never time out, because before each timeout another IPC is received by the server. Therefore, the interrupt server also raises a timer event if the last timer event happend a certain amount of time ago.

When the interrupt thread is notified of an event, it triggers the injection into the guest. If the guest is currently in a running state, the interrupt thread uses `ExchangeRegisters` to request an Interrupt Windows Exit the next time the guest is able to receive interrupts. If the guest is currently halted and descheduled (after executing `HLT`), it sends a notification message to the monitor thread. The monitor thread then resumes the guest by injecting the interrupt properly (see also section 4.2.1).

4.4 Guest Binary Modifications

Although the design and the technology provide grounds for virtualization of unmodified guests, I took a shortcut for the network card's DMA controller. Instead of filtering IO access to the network card to configure the DMA controller with correct physical addresses, I modified Linux's `virt_to_phys()` and `phys_to_virt()`. Those functions are responsible for translating a virtual address to the corresponding physical address and are mostly used in DMA related operations. My modifications add a static offset to physical memory addresses so that Linux itself already programmed the DMA controller correctly. When guest DMA access is fully managed by the VMM this shortcut can safely be removed from the guest.

Chapter 5

Evaluation

In this chapter I will first give an overview over how far the Linux bootup process is supported by the VMM. In the second half of this chapter, I will evaluate a Linux instance running on the VMM against a Linux instance running on bare hardware. Both have a comparable setup: The evaluation was performed on the same machine, with a VT-enabled Intel CPU with 3.6 GHz, 2MB Cache, 2GB RAM and a Gigabit network card. The VMM runs a single instance of the guest, and the guest has direct device access to the network card. Both guest and bare Linux run from a ramdisk. The Linux kernel version used is 2.6.9.

5.1 Booting

The VMM successfully boots a Linux 2.6.9 kernel and runs a small Debian installation from a ramdisk. The kernel completes all necessary boot steps beginning from CPU detection and virtual memory activation over device probing to the login prompt. Although serial input is not implemented, the user can login over the network if the ramdisk contains a SSH server. Once logged in, the user can start arbitrary applications, e.g. a webserver.

I made the subjective observation that the bootup procedure is slower on the VMM than on bare hardware. I suspect the main reasons for this to be first the inefficient vTLB implementation in the kernel and second slow device probing of passthrough devices. For the second case I was not able to find the source: Probing devices is taking at least a factor of 100 longer than on bare hardware[1].

See table 5.1 for an overview over Linux's boot steps and whether they can be completed successfully, if booted on the VMM.

[1] The strange thing is that this problem does not occur when using an afterburnt (but unpatched) Linux kernel (i.e., with some `NOPs` after each privileged instruction) instead of an unmodified Linux kernel. I suspect that these delays come from kernel-internal busy wait loops, during which the kernel switches to the idle thread several times. The main difference here between the unmodified and the afterburnt kernel is that the `HLT` instruction is not covered by a blocking `STI`, due to a bug in the afterburning procedure (see footnote 2 on page 21). Unfortunately this is not the source for the problem, which I tested by patching the afterburnt binary.

Boot step	State
BIOS RAM map	Y
Virtual memory	Y
vCPU detection	Y
Console output	Y
HLT check	Y
WP bit check	Y
MWAIT for idle	F
Fast system calls	F
PCI passthrough access	Y
Serial port	Y
IO APIC	Y
Timer calibration	Y
Real time clock	Y
Mouse input	N
Parallel port	N
Floppy disk	N
Network card passthrough (using Intel e1000 driver)	Y
Harddisk passthrough access	F
USB support	N
Static NIC configuration	Y
Ramdisk support	Y
INIT fork	Y
Swap disk activation	S
Login prompt	Y
Console input	N
Ping	Y
SSH access	Y
Apache server	Y
Reboot	N

Table 5.1: Overview over Linux 2.6.9 boot steps and state in the implementation. Abbreviations are: (Y) implemented, step completes successfully; (N) not implemented; (F) bootup fails if not deactivated in the VMM or via Linux kernel options; (S) skipped by Linux.

5.2 Network Performance

I used the `netperf` benchmark to evaluate I/O and network performance. The evaluation machine acted as netperf server. The client was a Dual Opteron with 3.2GHz and a Gigabit network card. I calculated CPU utilization as quotient of unhalted clock cycles and total clock cycles which I measured using hardware performance counters. Table 5.2 show the result of the benchmark run. We can see that the virtualized guest achieves almost the same throughput as the Linux on bare hardware, but pays with a higher CPU utilization. This is the expected result: Because the guest has direct access to the network card and uses DMA to copy data, there is almost no CPU overhead for the data transfer. The higher CPU utilization is mostly a result from additional code in the VMM, which is executed for example whenever the guest accesses IO ports (e.g. the PIC) or on vTLB updates on guest pagefaults. Because the CPU overhead is small enough, the throughput is not significantly affected.

	Bare Hardware	Afterburner/VT
Throughput [MBit/s]	854.36	852.62
CPU Utilization [%]	20.4	55.5

Table 5.2: Network throughput achieved by the `netperf` benchmark.

5.3 Webserver Performance

The performance of a webserver in a virtual machine is of twofold interest. First, a webserver is a common used application to be consolidated into a virtual machine. And second, a webserver depends heavily on the kernel for file access, sockets/network and multithreading/-tasking. We can therefore expect that if a webserver performs acceptable, any other common application should perform acceptable as well.

For measuring webserver performance I used the program `ab` [3], a benchmarking tool for the apache webserver. `ab` performs a given number of requests on a URL, a display a report about timings in the end.

The benchmarked webserver was an apache2 webserver. It ran locally from a ramdisk and provided three files of sizes 3MB, 16KB and 1MB. `ab` loaded the three files 1000 times each. To eliminate major errors in measurements, each experiment was reiterated three times. Table 5.3 shows `ab`'s execution times for Linux running on bare hardware and Linux running on Pistachio-VT/Afterburner, and the overhead introduced by virtualization over bare hardware.

While the overhead for File 1 and File 3 seem acceptable, File 2 breaks rank. I suspect that the reason for this is the small size of File 2. This might result in a context switch without fully filling the transfer buffers. Therefore more context switches in relation to transferred file size are necessary. Because of the brute force vTLB implementation in the kernel (the vTLB is flushed completely on guest context switches) this results in worse overall performance. Still, even a performance penalty of around 50% for unoptimized, non-production-level kernel and user level prototypes is acceptable, considering that the vTLB as

a major performance bottle neck can very likely be replaced by a hardware solution in the near future.

	Bare Hardware [s]	Afterburner/VT [s]	Overhead[%]
File 1	2.938316	3.425481	0.165797
(3MB)	2.937339	3.400574	0.157706
	2.938438	3.405301	0.158881
File 2	0.154082	0.242747	0.575440
(16KB)	0.154372	0.236964	0.535019
	0.154089	0.234748	0.523457
File 3	1.133297	1.313877	0.159340
(1MB)	1.125355	1.314016	0.167646
	1.136073	1.458734	0.284014

Table 5.3: Execution times of the apache benchmark tool `ab`. The column *Overhead* specifies the performance penalty of the virtualized version against bare hardware.

Bibliography

[1] Intel architecture software developer's manual: Volume 2: Instruction set reference, March 2006.

[2] Intel architecture software developer's manual: Volume 3: System programming guide, March 2006.

[3] ab Apache HTTP server benchmarking tool. http://httpd.apache.org/docs/2.0/programs/ab.html.

[4] Sebastian Biemüller. Hardware-supported virtualization for the l4 microkernel, September 2006.

[5] Afterburner framework. http://l4ka.org/projects/virtualization/afterburn/.

[6] Joshua LeVasseur, Volkmar Uhlig, Matthew Chapman, Peter Chubb, Ben Leslie, and Gernot Heiser. Pre-virtualization: Slashing the cost of virtualization. Technical Report 2005-30, Fakultät für Informatik, Universität Karlsruhe (TH), November 2005.

[7] Jochen Liedtke, Uwe Dannowski, Kevin Elphinstone, Gerd Liefländer, Espen Skoglund, Volkmar Uhlig, Christian Ceelen, Andreas Haeberlen, and Marcus Völp. The l4ka vision, April 2001.

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

[9] I. Pratt, K. Fraser, S. Hand, C. Limpach, A. Warfield, D. Magenheimer, J. Nakajima, and A. Malick. Xen 3.0 and the Art of Virtualization. *Proc. of the 2005 Ottawa Linux Symposium*.

[10] Jan Stöß. I/o-flexpages on the x86-architecture, May 31 2002.

[11] Rich Uhlig, Gil Neiger, and Dion Rodgers. Intel virtualization technology. 2005.

[12] C.A. Waldspurger. Memory resource management in VMware ESX server. *ACM SIGOPS Operating Systems Review*, 36(si):181, 2002.

[13] VMWare Workstation. http://www.vmware.com.