Großer Beleg

Efficient Virtualization on ARM Platforms

Steffen Liebergeld

06. Mai 2009

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

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

Erklärung

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

Dresden, den 6. Mai 2009

Steffen Liebergeld

Acknowledgements

I would like to thank Professor Hermann Härtig for allowing me to work in the operating systems group. I am especially grateful for the help of Michael Peter, who contributed a lot to this work with his competence, his helpful explanations and patience. I would also like to thank Henning Schild, who wrote the original Kqemu-l4 for x86. Further thanks go to Adam Lackorzyński for fixing a bug in Fiasco and implementing support for module loading in L4Linux on ARM, and Alexander Warg for proof-reading this document. I would also like to thank the OS group and the students of the student lab for the interesting discussions and the relaxed atmosphere.

Many thanks go to my parents who enabled me to study, and Stefanie for her great patience and support.

Contents

List of Figures

1 Introduction

In the recent past embedded devices have seen a rapid gain in popularity, a trend that seems to carry on into the foreseeable future. Being no longer limited by resource constraints, many devices that started as special purpose appliances have evolved towards general purpose gear. Mobile phones, for example, are no longer used as telephones exclusively, but have become devices that can be used to take photographs, play music and browse the Internet. They are even used to store all sorts of personal data such as passwords and contact lists. The revelation of that data may cause personal distress or result in financial loss. Also third parties such as content providers call for functionality to support DRM. Therefore embedded devices have become a security concerned environment. While these strict security requirements call for a very static setup, people still want to be able to install new software and enhance the functionality of the device.

The market for embedded systems is special in that it does not have commodity products but OEMs build both custom hardware and tailored software. This has resulted in a wide variety of operating systems, such as Windows CE, Symbian OS, Google Android and Palm OS, each having its own API and user interface. The growing complexity of embedded operating systems makes it hard to implement new features. However, OEMs stick to existing operating systems because of rigid time-to-market requirements and substantial development costs. Development of software, such as web browsers or custom enterprise software takes huge investments and usually requires years. Because of the market pressure OEMs have to push new features to the market at a high pace, leaving little time for development. That is why the reuse of legacy applications is very important for embedded platforms. This challenge has been tackled in various manners, such as abstract virtual machines. An example is Java, that is widely used to establish a common environment for cross platform development. However such an abstract virtual machine needs a lot of infrastructure such as a compiler, a byte code interpreter and custom libraries. Furthermore applications have to be developed explicitly for the Java VM, excluding a large body of non-Java legacy applications.

System *virtualization* can be used to develop systems that support legacy software, fulfill strict security requirements and support real-time execution. After having led a shadowy existence for decades, virtualization has eventually gained momentum in the general computing landscape. Early inroads into server and desktop environments are likely to be followed by a fair adoption in the high-end embedded market.

With virtualization OEMs can establish a common interface for legacy applications that is more efficient than a Java VM. Virtualization enables the execution of unaltered legacy software and is therefore a general approach that has a huge scope. Virtual appliances, virtual machine images without external dependencies, can ease software distribution for mobile platforms. The adaption of applications that usually has to be

done for every new device can be avoided by using the interfaces of a virtual machine. Virtualization can thus be used as a transition path to foster device adoption.

Because operating systems often do not provide needed functionality, virtual machine monitors (VMMs) are commonly implemented in an ad-hoc fashion by means of kernel modules. The unrestricted nature of kernel modules results in the evasion of operating system (OS) implemented security mechanisms, effectively leaving unlimited system access to the VMM. If a VMM is implemented by using only the interfaces of the host operating system, virtual machines (VMs) may be treated like any other protection domain. As such, any security policy that can be implemented by the host OS is readily applicable to information flowing in and out of the VM. Such an unobtrusive VM can be scheduled as one entity allowing the host operating system to enforce any scheduling strategy such as real-time scheduling.

If an instruction set architecture contains sensitive instructions that do not trap when executed in user mode, a trap-and-emulate virtualization scheme can not be implemented, and therefore the architecture is not virtualizable. On non-virtualizable architectures such as ARM, guest privileged code cannot be executed natively without loss of equivalence and has to be emulated. Equivalence is not affected for non-privileged code which accounts for the majority of executed instructions in many scenarios. With this work I present a virtualization solution for the ARM platform that executes unprivileged guest code directly on the host CPU and thereby achieves a significant acceleration compared to emulation.

My solution leverages the interfaces of an L4 microkernel and unlike other VMMs requires only small modifications to the kernel that do not compromise the security properties of the system.

The document is structured in the following way: Section 2 provides an overview of the used software and hardware environment. It is followed by the introduction of the design of my solution in section 3. Section 4 contains details about the actual implementation, which is evaluated in section 5. Section 6 is comprised of the conclusion and an outlook.

2 Background

2.1 Microkernel

In the 1980s several problems of existing monolithic operating systems surfaced. The size and complexity of the monolithic kernels had increased significantly. More code meant that the operating system was much more prone to error. Even worse, there was no fault isolation between different parts of the kernel. As such, a single failing component could lead to the failure of the whole system. In order to support new hardware and application requirements, new operating systems needed to be explored which proved difficult with existing monolithic kernels [Lie96a].

This is where the idea of a microkernel was invented: Only a small policy-free portion of code is executed in privileged mode. Operating system functionality such as file systems and device drivers is implemented in servers which run in user space. These servers are isolated meaning that errors may only affect the server and do not necessarily threaten the system stability. Microkernel-based systems are more amenable to customization and may even run multiple coexisting personalities. The microkernel design also enforces a more modular system structure. A smaller kernel should in general be much more maintainable and less prone to error [Lie96b].

So microkernels would deliver new levels of flexibility and power, while still allowing to keep Unix compatibility when inventing new operating systems [Lie96b]. The microkernels of the first generation were stripped down Unix kernels and as such they retained functionality such as the memory management in the kernel. This turned out to be problematic when kernel functionality adversely interfered with similar user-land facilities.

An important example of these first generation microkernels is **Mach**, which was developed at the Carnegie Mellon University. It provided four basic abstractions: **tasks** representing address spaces, **threads** which represent activities in tasks, **ports** are communication channels and **messages** which are typed collections of arbitrarily sized data or typed capabilities. Early tests using the in-kernel Unix layer showed good performance [ABB+86].

However Mach could not deliver on all the promises of microkernels. It implemented asynchronous inter-process communication (IPC) with in-kernel buffers which burdened the IPC path with numerous complicated corner cases. A conceptual weakness of all first-generation microkernels is the in-kernel policy such as memory management policy, because it hinders the implementation of new policies in the system [Lie96b].

After analyzing the drawbacks of early microkernels Jochen Liedke proposed to design microkernels from scratch. According to him microkernels should be as small as possible following the rule ”everything that can be implemented in user space without

compromising the security of the system should not be implemented in the kernel". To keep flexibility all policy should also be implemented in user space.

The *L4 specification* defines a set of abstractions, which are:

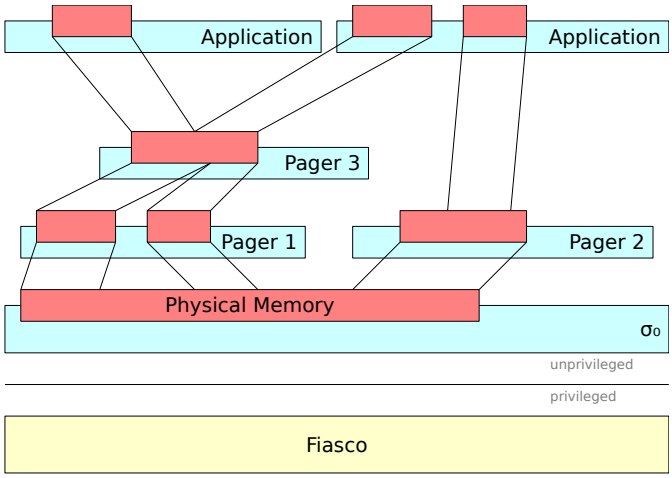
- **Address spaces** are used to enforce isolation. They are the units of protection in the system (protection domains).
- **Threads** are the abstraction for activity inside an address space. They are also the endpoints of IPC.
- **IPC** is optimized to be fast. It is synchronous which dispenses with buffering in the kernel. Agreement by both the sender and the receiver is needed in order to execute the IPC. All message marshaling and unmarshaling is done in user space.

Interrupts and exceptions are mapped to IPC allowing device drivers and exception handlers to be implemented in user space.

Address spaces are constructed in an hierarchical manner and managed in user space. In order to do so, L4 provides a special form of IPC called *flex-page IPC*. By using flex-page IPC, threads can delegate access rights to a page to another task. Both sender and receiver have to agree before a mapping can be established. Revocation of access rights is done without consent of the receiver. Every thread has an associated pager. On a page fault the kernel synthesizes a page-fault IPC on behalf of the faulting thread including the faulting address and information on the kind of page fault. The pager can then use this information to map the appropriate pages into the faulting address space. A special task is σ_0 which has access to physical memory and serves as the root pager of the system. The memory system is depicted in Figure 2.1.

Several flavours of L4 exist such as V2, X.0, X.2 and L4.sec. There are several implementations such as Fiasco, Pistachio, and OKL4 [sgTD].

Figure 2.1: Hierarchical memory management



2.1.1 Fiasco

Fiasco is an implementation of the L4 API. It is written in C++ for the x86 platform. Development was initiated by Michael Hohmuth at the chair for operating systems at the University of Technology in Dresden.

Fiasco supports real-time execution. It is fully preemptable, providing small interrupt latencies which is a key requirement for real-time applications. It employs helping locks to implement wait-free synchronization.

Fiasco implements *alien threads*. On every exception, the kernel synthesizes an exception IPC on behalf of the alien thread and sends it to its pager.

Fiasco is licensed under the terms of the GPL and runs on x86, x86_64, Linux user space, and on the ARM processor. It implements several versions of the L4 specification, such as V2, X.0[Hoh]. Fiasco is used for research. For example, it is used as a basis for experimental implementations of L4 X.2 and L4.sec.

2.1.2 L4Linux

L4Linux is a port of the Linux kernel to the L4 microkernel [Lac]. It runs completely in user space and allows the execution of unmodified Linux applications [BHWH97].

The L4Linux server is implemented with multiple threads. There are threads to handle interrupts from devices and a timer thread, which handles timer interrupts. The main L4Linux thread runs the Linux code.

In L4Linux every Linux process is implemented by an L4 task. The user address-space layout is the same for Linux and L4Linux processes.

Conventional Linux is in full control of the memory management hardware of the computer. Hence memory operations, such as allocation and mapping are implemented by direct page-table manipulations. In an L4 system the microkernel is in command of the page tables. Consequently all memory operations in L4Linux are implemented with L4 primitives.

The implementation of L4Linux employs an extended cross-task *thread_ex_regs* syscall to force running processes back into the L4Linux kernel.

Currently version 2.6 of the Linux kernel is used as a basis for L4Linux [Lac04].

2.1.3 DROPS

The DROPS system is built upon the Fiasco microkernel (see 2.1.1). As a microkernel, Fiasco strives to provide mechanisms only with policies implemented in user-level servers. This allows different security schemes to be run side-by-side. DROPS includes L4Linux and the L4Env environment. Applications developed for DROPS may have a very small trusted computing base.

2.2 ARM Architecture

The ARM architecture was originally developed by Acorn Computers in 1985. It allows efficient processor implementations with a comparably low power usage. Today it is popular for embedded applications such as smart phones and mp3-players. ARM Incorporated sells the chip design to System on Chip (SOC) manufacturers.

2.2.1 ARM Instruction Set Architecture

The ARM instruction set architecture (ISA) is a reduced instruction set following the load-store principle. The ARM processor has a set of 16 user registers, two of which serve as program counter and branch register, respectively. Program status such as flags and processor mode are stored in a special program status register (PSR).

The ARM ISA went through various stages of evolution. In version 4 the Thumb instruction set was added. Thumb instructions are slightly less powerful but have a 40% lower instruction footprint than original ARM instructions. It is therefore used in memory constrained devices [ARM00].

Later direct support for Java bytecode was added. This technique is called Jazelle and allows the native execution of Java byte code in hardware [arm].

2.2.2 Memory Management

Depending on the intended use of the machine, system builders can choose to employ either a simple *memory protection unit* (MPU) or a powerful *memory management unit* (MMU). Both MPU and MMU enforce memory protection and are programmable only in privileged mode and can therefore provide isolation.

The MPU defines eight regions within the physical address space and allows assigning access permissions and cacheability attributes to individual regions. It does not do any address translation.

The MMU on the ARM processor is implemented using two stage page tables. Supported page sizes are 1Kb, 4Kb, 64Kb and 1Mb. Translation look aside buffers (TLBs) for data and instructions cache recently translated addresses and are often implemented in a fully associative manner.

Until version 5 caches were virtually tagged, which mandated a cache flush on address space switch, making address space switches expensive. So in order to reduce the costs of address space switching a mechanism called the fast context switch extension (FCSE) was introduced. It allows multiple processes to have their first 32Mb use identical address ranges as long as their physical addresses differ. The virtual addresses within the first 32Mb of the address space are augmented with the process identifier (PID) register. TLB flushes on address space switch can often be avoided with this information. FCSE might be used to implement a system with small address spaces that has low overhead in address space switches.

2.2.3 Exceptions

The ARM processor supports exceptions to signal certain events. To handle an incoming exception the processor switches to the according processor mode. User code runs in the unprivileged user mode. All other modes are privileged.

The following is a list of exceptions, their cause and the resulting processor mode.

- **Reset:** Restarts the processor from a known state
- **Data Abort:** Is raised when the memory management unit encounters invalid access permissions on load or store. This abort is also issued on any other address translation error (*abort-mode*).
- **Fast interrupt exception:** Issued when the processor receives an interrupt from a designated fast interrupt source (*fiq-mode*).
- **Normal interrupt exception:** Issued upon encountering an interrupt (*irq-mode*).
- **Prefetch abort:** Raised by the MMU when encountering invalid access permissions during instruction fetch (*abort-mode*).
- **Software interrupt:** Raised upon a special instruction (*system-mode*).
- **Undefined instruction:** Raised upon encountering an operation neither the processor nor any coprocessor can handle (*undef-mode*).

The processor modes have banked registers which are used in order to save the state of the program before the mode switch. The saved program status register (SPSR) stores the content of the program status register at the time the exception was raised.

2.3 Virtualization

An interface is a set of rules which describe what actions need to be taken to achieve a certain result. *Virtualization* is a method whereby the interface is retained for the interface user whereas the original interface provider is replaced with a new one. The new provider may (safely) share one resource among multiple users, provide the service by means of other resources or implement additional services such as logging or screening. While virtualization as characterized above covers numerous situations I will restrict the use of the term to virtualizing machines at the system level. A *virtual machine* is an efficient, isolated duplicate of a real machine [PG74].

Reasons for using virtual machines include:

- **Consolidation and green computing:** Multiple servers are hosted on one physical machine without compromising isolation. It is possible to quickly deploy VMs with different software configurations in order to debug test programs. Modern systems are often underutilized and it makes sense to run multiple VMs on a system in order to achieve a better utilization. This also leads to an improved energy efficiency.

- **Load balancing and fault tolerance:** VMs can be migrated from a system with high load to a system with less load. Furthermore it is possible to migrate VMs away from failing systems.
- **Isolation:** Security critical parts of a system may be split into multiple VMs in order to achieve better isolation. In such a system attackers have to overcome not only the isolation boundaries of the OS, but also those of the VMM.
- **Better inspection:** The behavior of virtual machines can be analyzed and the operator can be alerted whenever a virtual machine shows unusual behavior. The introspection techniques may also be used for operating-system development as it also enables extensive and deep debugging.
- **Flexibility and configurability:** Virtual machines can be more easily created, altered and replaced than real hardware.
- **Legacy system support:** Legacy systems may be run inside of a VM, which may feign hardware that is no longer physically available. Existing systems can thus run on hardware that they were not intended for as the actual low level device drivers reside in the VMM.

A virtual machine has to adhere to the following requirements [PG74].

- *Equivalence:* All instructions must have the same results as they have on the real machine.
- *Control:* Only the resources explicitly allocated to the virtual machine are accessible to the guest.
- *Efficiency:* A statistically dominant subset of the virtual processor's instructions has to be executed directly on the real processor. This requirement explicitly rules out emulators.

The term **guest** will be used throughout the rest of this document with the following meaning: a guest is constituted by all the software which is executed inside a virtual machine. This includes the operating system and all applications.

2.3.1 Virtual-Machine Monitors

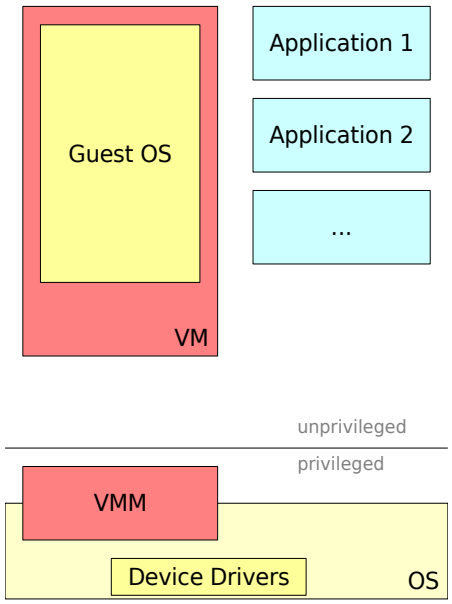
Virtual-machine monitors (VMMs) implement *virtual machines* (VMs) [PG74]. VMMs are either hosted on top of a host operating system or hypervisors.

In a hosted environment, the VMM resides in the host OS in a cooperative manner. It is still the OSs duty to implement basic services such as scheduling and protection domain management as well as driving the system devices (see Figure 2.2).

Contrary, the VMM as hypervisor exercises full system control. For practical reasons, it may delegate some duties such as device interaction to a less privileged component, though.

Examples for hosted VMMs are VMware Workstation [vmwb] and VirtualBox [vir]. Examples for hypervisors are Hyper-V [hyp] and VMware ESX [vmwa].

Figure 2.2: Hosted Virtual-Machine Monitor



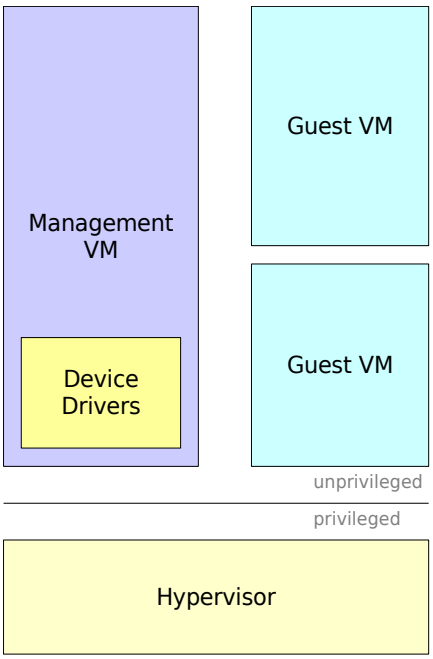
The VMM has to virtualize the following resources, giving each VM the illusion of exclusive access:

- Processor
- Main memory
- IO devices, such as mass storage and network devices

2.3.2 Paravirtualization

On some platforms, such as x86, processor virtualization is hard to implement efficiently because there are sensitive instructions that do not trap in user mode. One solution is to present a software environment to the virtual machine which is slightly different from the underlying hardware. Such implementations may be less complex and more efficient than VMMs implementing full virtualization. Guest operating systems need to be modified to be used in a paravirtualized environment. An example for paravirtualization is Xen [xen].

Figure 2.3: Hypervisor (simplified)



2.3.3 Emulation

Another way to run guests on top of a host is *emulation*. In emulation every instruction of the guest is interpreted and executed by the emulator. Achieving equivalence and isolation, emulators are not considered to run virtual machines because they do not adhere to the *efficiency* criterion. Each guest instruction causes multiple host instructions to be executed.

For emulation no support neither from the host operating system nor from the host hardware is needed. A guest may even have a different ISA than the host.

Emulators are typically by factors slower than the underlying hardware. Causes for the slowdown are manifold and inherent: Today's computing hardware typically employs means of concurrent computation like pipelines or simultaneous address translation, which an emulator has to reproduce in software. Hardware facilities such as translation look aside buffers (TLBs) are not available leading to even more instructions. Another problem is instruction inflation, which means that in order to run few guest instructions a lot more host instructions have to be executed. An emulator has to fetch the guest instruction, decode and finally execute it. Fetching an instruction requires extra effort in any jump but also regular (non-branching) operation, inducing loop overhead that can account for a fair portion of the overall overhead. When the guest ISA does not match the host one with respect to condition flag handling then achieving the required behavior requires additional effort. Another problem is the register pressure: The guest ISA may use more registers than the host, and the emulator has to keep additional

administrative data. This means that the host register set often does not suffice and additional operations are needed in order to load and store register contents to memory. Instruction inflation and register pressure also lead to a much bigger cache footprint. All these problems cannot be overcome and great effort has to be put in optimizations.

Examples for system emulators are Bochs [boc], Qemu [qem] and JPC [jpc].

2.4 Virtualization on the ARM Platform

The term *sensitive instruction* refers to instructions that either behave different in different processor modes or that are only allowed in privileged processor modes.

A number of sensitive instructions in the ARM ISA have been identified [BRG+06]:

- Instructions that manipulate the the program status register (MRS, MSR).
- Instructions that have different effects depending on the processor mode (STM, LDM).
- Coprocessor instructions that directly touch the MMU and page-tables (LDC, STC, MRC, MCR, CDP).

Because not all of these instructions trap when they are run in user mode, trap-and-emulate cannot be implemented. Therefore the ARM ISA is not virtualizable.

2.4.1 TrustZone

The ARM TrustZone is an ISA extension aimed at creating an environment that enforces a clear division between a secure and a non-secure side. On the secure side a small trusted operating system is employed, which has access to all the security critical infrastructure such as cryptographic keys and critical hardware. A slightly modified commodity operating system such as Linux can be run on the non-secure side. Operations which are not security critical such as drawing the user interface or the implementation of network protocols can be done on the non-secure side by untrusted software without compromising the overall system security. Only a well defined channel, controlled by a monitor, allows both the secure and the insecure side to exchange information.

The ARM TrustZone solution is a technique tailored for a rather static setup of a small secure system on the one hand side and a modified commodity operating system on the other side. It does not allow for efficient execution of multiple guest operating systems [tru].

2.4.2 Qemu

Qemu is a machine emulator. It uses dynamic translation to achieve good performance [Bel].

Qemu supports two execution modes:

- **User-mode emulation** allows the execution of user space applications on a different architecture. In this mode syscalls are forwarded to the host operating system.

- **Full system emulation** on the other hand emulates complete machines and can be used to execute unmodified guest operating systems.

Qemu has support for different host architectures such as x86, x86_64, PowerPC, Sparc32 and ARM. Work is underway to support even more hosts. Supported emulation target CPU architectures include x86, x86_64, PowerPC, Sparc, Mips and ARM.

The ARM emulation target supports different baseboards such as the Integrator, with supported CPUs ARM926E, ARM1026E, ARM946E, ARM1136 or Cortex-A8 and the RealView Board with the CPUs ARM926E, ARM1136 or Cortex-A8. It also supports a range of peripherals for these boards, including ethernet devices and LCD controllers.

To speed up code execution on virtual machines where target architecture matches the host architecture a special kernel module has been developed. As of now it supports x86 and x86_64 hosts and guests. It currently runs on Linux 2.4, 2.6 and Windows hosts. The kernel module enables native execution of guest user space code. It also allows guest privileged code to be run but provides only a paravirtualized environment. That means that not only user-mode but also kernel-mode code of the guest can be executed natively.

During his diploma thesis, Henning Schild implemented a Kqemu compatible kernel module for L4Linux. It is called **Kqemu-l4** and allows user code to be run natively. He was able to leverage the interfaces of the L4 system [Sch08].

3 Design

3.1 Goals

With my work I want to create a secure and fast virtualization solution for the ARM platform.

My solution has to fulfill the following criteria:

1. *Control:* VMs should only be able to access resources that have been explicitly allocated to them. Different VMs have to be isolated from each other in order to be secure.

2. *Efficiency:* My solution has to have as little overhead as possible. As many instructions as possible should be executed directly on the host processor without VMM intervention.

3. *Equivalence:* The VM should behave like the real system, supporting unchanged guest operating systems. I want to be able to run existing OS like Windows Mobile and Symbian OS inside of VMs.

4. *Security:* The trusted computing base of applications running side-by-side with VMs should not be increased.

5. *Small changes to the environment:* Every change has to be ported to new platforms and revisions of the base system. So in order to be able to maintain my software it is important that the underlying system is very close to the original.

3.2 Virtualization

3.2.1 CPU and Memory Virtualization

CPU virtualization is complicated by the behavior of some sensitive instructions that do not trap when executed in user mode (see 2.4). The ARM ISA is not virtualizable because of these instructions. This means that a VMM on ARM can only execute guest user-mode code natively without loss of equivalence and requires guest privileged code to be run in emulation.

When executing guest user-space code natively, the VMM has to be able to regain control under all circumstances. A VM may only access resources that were explicitly assigned to it. Any exception, be it voluntary (e.g. a syscall) or involuntary (e.g. a page fault) is reported to the VMM immediately.

A guest system has asynchronous state changes such as device and timer interrupts that have to be supported. On a microkernel based system this could be implemented

with IPC. A device emulator would have to synthesize a message on behalf of the device. The VMM needs to have a thread that listens for the IPC and intercepts the execution of the guest if needed.

In modern operating systems memory is abstracted away from physical addresses to virtual addresses with the use of page tables and a memory management unit. The memory management unit translates from virtual to physical address on every memory access. Direct manipulation of page tables is a privileged operation and is therefore done by the operating system kernel. In virtualization the guest is provided with the illusion of physical memory. The guest manages its own guest page table, and the mapping of guest virtual to guest physical address is done by a virtual memory management unit that operates using the guest page table. Guest physical memory resides in host physical memory.

When guest code is to be executed natively, all its memory accesses are subject to the host page translation. That is why the VMM traps on every page fault of the guest. The page fault occurs on the guest virtual address. The VMM walks the guest page table to find the corresponding guest physical address. If a mapping is found, then it is a shadow page fault and can be resolved without guest interaction: The VMM acts as the resource allocator for its VMs, and thus knows where the guest physical memory is located and uses the guest physical page address to compute the appropriate host page address and then establishes a mapping for the natively executing code. Such a mapping is then added to a shadow page table.

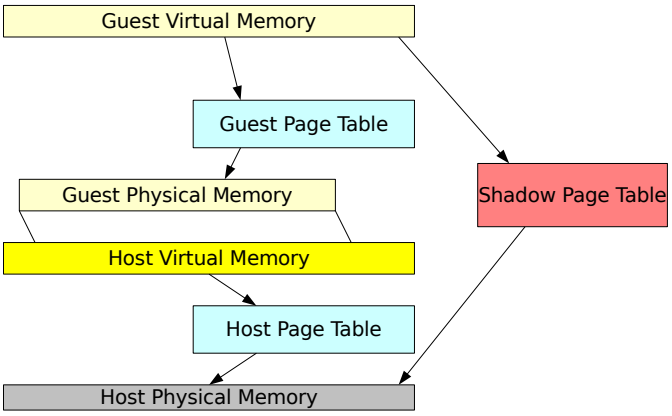
If the VMM cannot find a guest page mapping during the guest page table walk, the page fault is logical in the sense that the guest does not have a valid mapping for that page. The page fault can therefore only be resolved by the guest operating system.

In summary, we have the host MMU and one software MMU per guest. We have to handle three kinds of addresses: Guest virtual addresses are those used by the guest to designate a memory object in a guest address space. With the help of the guest page table guest virtual addresses are translated into guest physical addresses. If the guest ran on a physical machine these addresses would be used to access a physical memory object. The addresses that are used to designate memory objects in a system accommodating virtualized guests are host virtual addresses. The VMM lazily constructs a shadow page table that translates guest virtual addresses to host physical ones (see Figure 3.1 for an illustration).

3.2.2 Device Virtualization

Device virtualization can be split in tightly coupled devices such as timers and individual loosely coupled devices such as network cards. Device emulation either directly interacts with devices or is implemented with abstractions provided by a driver stack. For example, the hard disk seen by a guest can either be implemented by a file in a host environment which can either go to the local disk or involve a network file system. In either case the implementation has to reside in an user-level server, as mandated by the microkernel principle. Platform devices such as timers and IRQ controllers are often exposed only in an abstracted form. For example, the L4 interface does not include the subtleties

Figure 3.1: Guest Memory Management



of physical timers but provides IPC timeouts instead. For most applications, this abstraction is helpful.

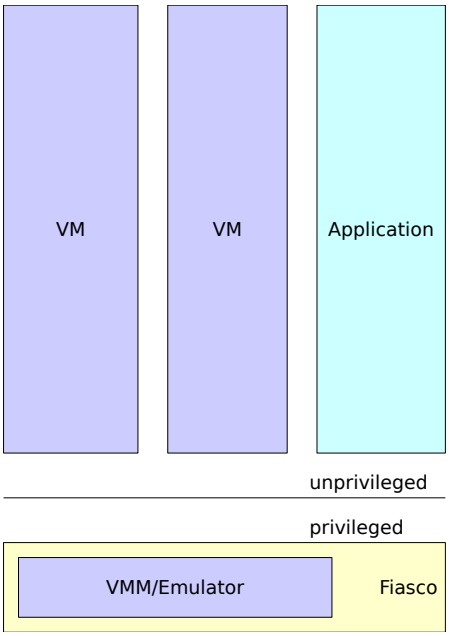
Devices have a memory like interface with read-write access. Accesses to these memory regions can have command characteristics (each individual one counts, no coalescing). Device memory has to be shared between the device emulator and the VM. To keep equivalence, guest drivers and applications in the VM have to have write access to device memory and the device has to be notified of any write. Because it is not possible to trap on memory access, device memory cannot be provided to the natively executing guest code. Instead any access results in a page fault and the code in question is emulated.

3.3 First Class VM

One way to host VMs is to introduce a new flavour of protection domain that sports the CPU interface (see Figure 3.2). VMs are treated as a new type of first class object and scheduled side-by-side with regular tasks. The VMM includes a CPU emulator that emulates all guest privileged code and provides a software MMU implementation. The guest page table and the shadow page table are managed directly in Fiasco. This approach enables very efficient page fault handling as no additional address-space switches and user kernel transitions are needed. However, a new kernel interface has to be designed that allows device emulators to get to know the addresses of the mapped device memory. Exceptions of natively executed guest code result in a user-kernel transition and the appropriate guest exception handler can be emulated immediately. Guest physical memory has to be put in a special task that exists only as a container for guest memory. Guest timers might be implemented using the Fiasco internal timer infrastructure.

Extra effort has to be done to support VM inspection and logging. To support user-level debuggers and loggers, exception IPC has to be extended to contain the whole

Figure 3.2: VMM as a First Class Object



ARM CPU state. Any event inside the VMM has to be reflected immediately to the user-space debugger or logger.

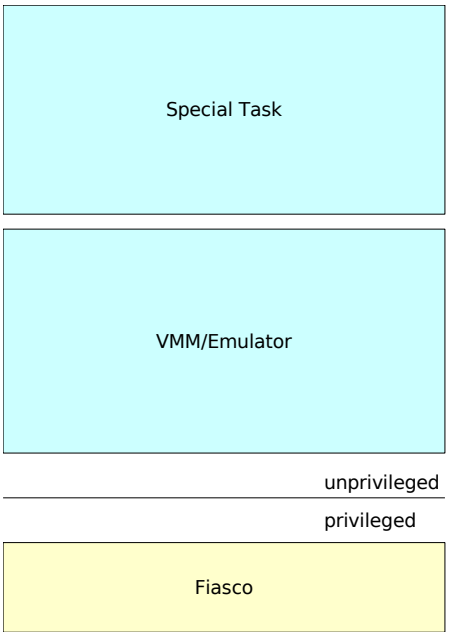
The implementation of a CPU emulator for the ARM ISA is a significant effort because a lot of different modes, coprocessors, and the Thumb instruction set have to be supported. With the inclusion of the CPU emulator, the management of virtual page tables, a new interface for device communication, and extended exception IPC, the complexity of Fiasco would be enhanced enormously, and the trusted computing base of all applications would be increased.

In such a solution user-kernel transitions of the guest that are mapped one on one to a user-kernel transition of the host providing efficient guest user-kernel interaction.

3.4 Emulation at User-level

The kernel growth can be avoided if the emulation of kernel code is handled at user level (see Figure 3.3). The execution of unprivileged guest code can be supported with rather small changes to existing Fiasco mechanisms. Native execution can be provided by employing a special task that has to have a clean address space, and all of its page-faults and exceptions need to be reported to the VMM. The VMM is implemented as a user-space server, and implements the CPU emulator and tightly coupled devices. A memory region inside the VMM address space can be used as guest physical memory. Page tables should be managed in the CPU emulator as well as the shadow page table. The VMMs threads and the special tasks thread would be scheduled like ordinary L4

Figure 3.3: Emulation at User-level



threads. Communication with the special tasks thread and the VMM thread is done using exception IPC. Support for inspection and logging can be conveniently implemented in the VMM server.

Timers can be implemented using IPC with timeouts. Loosely coupled devices may be implemented by other emulators. Their interrupts can be mapped to IPC and device memory can be shared between the CPU and device emulators. Another option is to implement emulation for loosely coupled devices directly in the VMM server thus reducing overall system complexity and the number of IPCs.

Implementing the aforementioned special task for native execution is of little effort: In contrast to an ordinary task, the special task has to have a clean address space in the sense that no pages in the kernel memory area should be user-accessible as well as the user area that must not have pages like the user thread control block (UTCB) or the kernel info page (KIP). Alien threads already have properties required by the special task: They are not allowed to do syscalls and all of their exceptions are reported to their pagers.

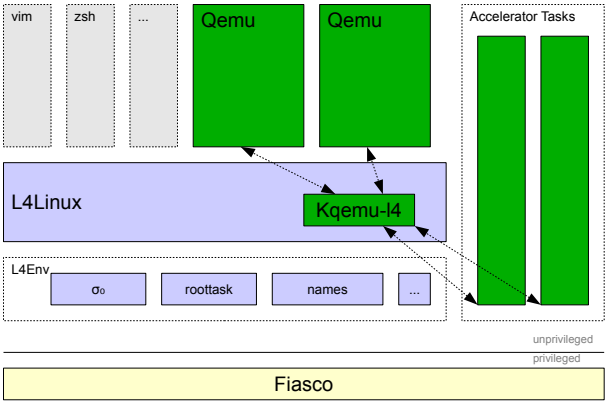
In contrast to an in-kernel approach, these modifications do not increase the trusted computing base of the system. However this solution causes more address-space switches and user-kernel transitions and thus has a higher overhead compared to an in-kernel solution.

3.5 Architecture

The approach of implementing a VMM in user-space has much more appeal than an in-kernel solution. Contrary to the in-kernel solution, it does not increase the trusted computing base of the system and the complexity of the microkernel is not significantly altered. A user-level solution has the advantage that a legacy emulator can be used and thus the effort needed for implementation is greatly reduced. The loss of performance due to the increased number of kernel-user transitions is outweighed by less kernel modifications.

That is why I chose to implement a VMM without large modifications to the microkernel. Fiasco supports alien threads, threads that are not allowed to do syscalls, but whose syscalls and exceptions are reported to its pager. I chose to use a special task with one slightly modified alien thread to be used for native execution of guest code. As a CPU emulator I chose Qemu, which already runs on Fiasco with the help of L4Linux. Qemu also implements emulation for both tightly and loosely coupled devices.

Figure 3.4: Architecture



So my solution consists of three parts as is depicted in Figure 3.4. I am using a modified instance of Qemu, a L4Linux kernel module (Kqemu-l4) and an accelerator task. The following paragraphs will describe all parts and their interactions.

3.5.1 Qemu

Qemu consists of the following subsystems [Bel05]:

- Processor emulator
- Emulated devices
- Generic devices (e.g. block devices and busses used to connect emulated devices)
- Machine descriptions (used to instantiate the emulated devices)
- Debugger and user interface

The processor emulator uses *dynamic translation* to emulate efficiently. A compile time tool is used to create a dynamic code generator, which concatenates micro operations to target instructions at run time [Bel05].

Basic blocks of guest code are translated into host instructions in one run, forming so called **translation blocks** (TB). Qemu maintains a cache for the generated TBs. After execution of a TB, Qemu tries to find the next TB. On success, Qemu branches directly to its start. If the next TB is not yet translated, the dynamic translator is invoked. To decrease the loop overhead, Qemu also tries to patch existing TBs with the correct branch address so that the execution can jump directly to the next TB without leaving the TB cache. This technique is called *direct block chaining*.

Checks for interrupts are done only at TB boundaries. That means faster execution because it avoids constant polling [Bel05].

Internally, Qemu has data structures storing the state of the guest CPU and devices. It uses a continuous area inside of its address space as guest physical memory. On Linux and FreeBSD this is done by creating a sufficiently large file and using mmap to map it into the address space [Bel].

Memory management of guests is done with a software MMU. Address translation is done on every memory access. Recent address translations are cached. When MMU mappings change, the chaining of TBs has to be reset because memory references may have changed [Bel05].

In the development version, Qemu supports execution on ARM Linux hosts. Being a Linux user-space application, it can be used without modification on L4Linux.

Qemu for x86 and x86_64 hosts has an interface to Kqemu. Henning Schild was able to use this as the basis for his work on Kqemu-l4 [Sch08]. He did not need to adapt Qemu in any way. At the beginning of my work, no such interface was available for ARM hosts.

3.5.2 Accelerator Task

Kqemu-l4 uses an L4 task with one *alien thread* per virtual machine to execute guest code natively. An alien thread is a L4 thread with the property that it is not allowed to execute syscalls and its exceptions are reported to its pager.

The Linux kernel module registers as the pager of the accelerator thread and is therefore invoked on any page fault and exception occurring in the thread.

3.5.3 Kernel Module

The kernel module manages the native execution. It implements the interface to Qemu and handles exceptions and page faults of the accelerator task. Because my solution is similar to Kqemu-l4, I decided to keep the naming.

3.5.3.1 General Operation

Qemu communicates with the kernel module Kqemu-l4 using a special device, usually */dev/kqemu*. Once Qemu opens */dev/kqemu*, it is registered as a client of Kqemu-l4. In turn, it is unregistered when it closes the file. Communication is done using IO controls (ioctls) (see 3.8).

When the KQEMU_EXEC ioctl is issued, Kqemu-l4 tries to execute code natively in the accelerator task. It returns to Qemu with one of the following return codes, indicating why it returns to Qemu and sending along the current CPU state:

- **KQEMU_RET_EXCEPTION:** The guest code triggered an exception, which has to be handled in Qemu.
- **KQEMU_RET_SOFTMMU:** The guest code has to be run in Qemu emulation.
- **KQEMU_RET_INTR:** Execution in the accelerator task was interrupted by the operating system.
- **KQEMU_RET_ABORT:** A fatal error has occurred.

Control is passed to the accelerator task by sending an exception reply IPC whereby the register state can be specified. Exceptions triggered in the accelerator task are encoded into exception IPC and sent to Kqemu-l4 by the kernel. Page faults are sent as exception IPC, containing exact information about where the page fault occurred, and what kind of page fault it is. In either case, CPU state of the accelerator task is saved in the **UTCB** and thus made accessible to Kqemu-l4.

Whenever Qemu instructs Kqemu-l4 to execute code natively it exchanges a list of pages which have to be flushed from the accelerator task. Such a flush has to take place for example if the accelerator task is going to execute guest code in a guest address space, other than the previously used one. After Kqemu-l4 flushed pages, it remaps pages that have been marked non dirty with read-only access rights.

When returning to Qemu, Kqemu-l4 informs Qemu of pages that have changed during native execution. Additionally, it marks pages that have been mapped with write access as dirty. Qemu may use that information for example to keep its cache of translated guest code consistent with the actual state.

3.5.3.2 Page Faults

Kqemu-l4 is the pager of the alien thread of the accelerator task. On page fault Fiasco synthesizes an exception IPC on behalf of the faulting thread and sends it to Kqemu-l4.

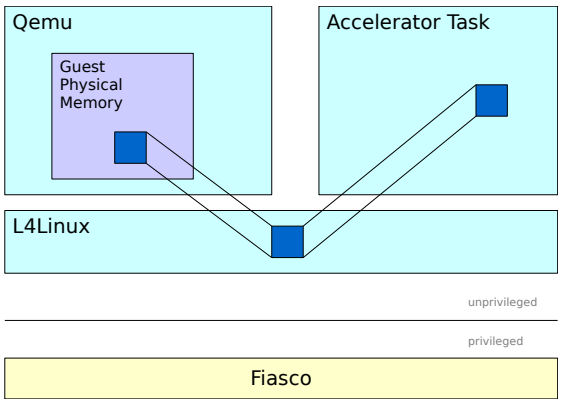
Page faults in the accelerator task are either **logical** or **shadow page faults**. If the page fault is logical, then it is a page fault of the guest, and only the page-fault handler of the guest OS can resolve it. In that case control will be given to Qemu to emulate the guest page fault handler to resolve the fault. The other case is when the page is accessible according to the guest, but is not currently mapped into the accelerator task. This situation is called a shadow page fault. Kqemu-l4 does the address translation in order to get to know the physical address of the page. It then maps the page from the address space of Qemu into the address space of the accelerator task. After resolving the fault, it hands control back to the accelerator task. A shadow page fault is transparent for the guest.

A shadow page fault cannot be resolved under any of the following conditions:

- The page has a size of 1Kb: because of a restriction in Fiasco, only pages with sizes which are multiples of 4Kb are supported
- The physical page has been marked by Qemu as IO memory
- The page would be mapped into the part of the address space which is reserved for the Fiasco microkernel

In either case, Kqemu-l4 hands control back to Qemu for software emulation.

Figure 3.5: Page Mappings in Kqemu-l4



As depicted in Figure 3.5, a page mapped into the accelerator task is mapped in the address space of Qemu and L4Linux as well. The mapping in L4Linux exists because Qemu is an L4Linux task, and L4Linux is its pager.

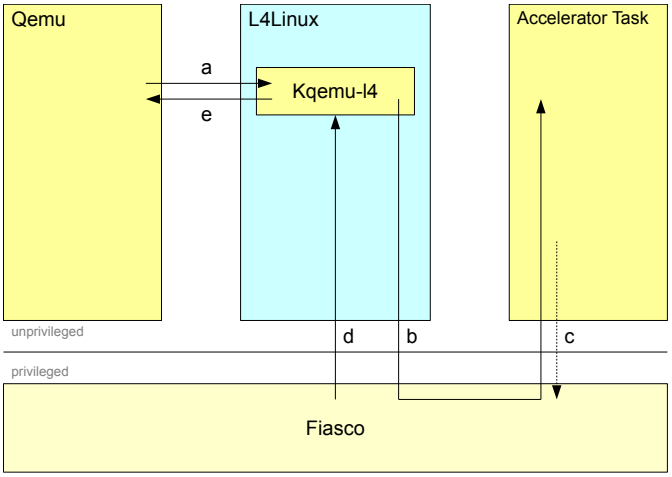
3.5.3.3 Exceptions

Exceptions triggered by the accelerator task are sent as exception IPC to Kqemu-l4. Kqemu-l4 decodes the type of exception and returns to Qemu with the KQEMU_RET_EXCEPTION return code. It also notifies Qemu on the type of exception.

After the accelerators time quantum is exhausted, the timer thread in L4Linux posts an exception, forcing the accelerator task to return to the L4Linux kernel. Subsequently Kqemu-l4 checks whether signals are pending. If so, it returns to Qemu with KQEMU_RET_INTR, indicating that the native execution has been interrupted by the host OS. L4Linux then sends the pending signals.

3.5.4 Control Flow

Figure 3.6: Control Flow



A graphical representation of the basic control flow inside the system is given in Figure 3.6:

a Qemu switches control to Kqemu-l4 to run code natively.

b Kqemu-l4 switches control to the accelerator task.

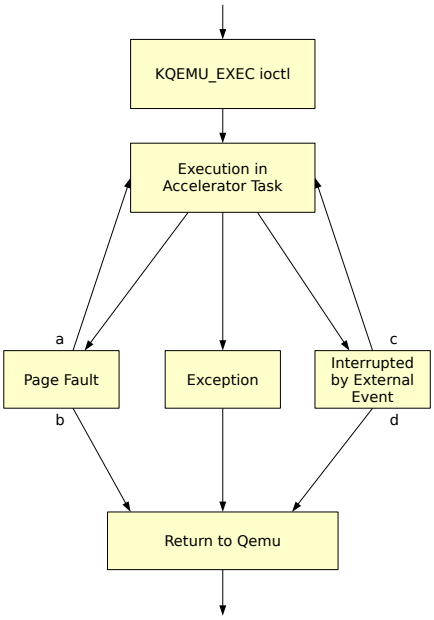
c The accelerator runs code and eventually causes an exception or a page fault. Control is automatically given to Fiasco.

d Fiasco synthesizes an exception IPC and sends it to Kqemu-l4.

e Kqemu-l4 analyzes the message and acts accordingly, either giving control back to Qemu or to the accelerator task

In Figure 3.7, you can see how the native execution is handled. On receiving a KEMU_EXEC ioctl, Kqemu-l4 cleans out pages as indicated by Qemu and switches to the accelerator task. In the accelerator task, guest code runs natively and eventually encounters one of the following conditions: If an exception is triggered by the guest code, Kqemu-l4 receives an exception IPC and switches back to Qemu. If the accelerator task is interrupted by L4Linux, Kqemu-l4 checks whether signals are pending. If signals are pending (c), it hands control over to Qemu else (d), it switches back to the accelerator task. If a page fault occurred, Kqemu-l4 receives an exception IPC from Fiasco that indicates a page fault. If the page fault is a logical one (b), control will be given back to Qemu. If it is not (a), Kqemu-l4 transparently resolves the fault and switches back to the accelerator task.

Figure 3.7: Decision Diagram of Kqemu-l4



a Shadow page fault

b Guest page fault

c OS event without effect for the guest

d Event in guest model

3.6 Limitations

The design is not well suited for workloads with a low user code executed per state transition ratio. Every context switch of the guest is translated into four context switches and two address space switches. Furthermore a lot of bookkeeping has to be done in Kqemu-l4 that further deteriorates performance.

This might seriously decrease performance especially for microkernel based guests. Consider the following simplified sequence:

- The guest microkernel has booted, its root pager is in place, and it creates the first user task.
- Control is given to the new task. Qemu switches to Kqemu-l4 in order to run the code natively.
- Kqemu-l4 hands control over to the accelerator task.
- The address space of the accelerator task is empty, thus the first memory access will cause a page fault.
- Fiasco hands control over to Kqemu-l4.
- Kqemu-l4 does the address translation, which fails because the page is not yet mapped by the guest (logical page fault).
- Kqemu-l4 switches control back to Qemu, which will run the guests page fault handler.

Figure 3.8: Kqemu-l4 IO Controls

- **KQEMU_INIT:** Initialize data structures inside the kernel module.
- **KQEMU_EXEC:** Switch from Qemu emulation to native execution.
- **KQEMU_GET_VERSION:** Get the version number of Kqemu. Kqemu is only used when the reported version number is the one expected.
- **KQEMU_MODIFY_RAM_PAGES:** Qemu may use this instruction to report modified pages to the kernel module. It is not currently used.
- **KQEMU_SET_PHYS_MEM:** Used to inform Kqemu of the memory layout of the guest.

- The guests microkernel will activate the pager of the task, which is also a user-space program. Therefore Qemu switches to Kqemu-l4 for native execution.
- The guest pager runs a very short time and resolves the page fault using IPC, thus switching to the guest kernel. Kqemu-l4 therefore switches back to Qemu.
- After the page fault is resolved, Qemu switches back to Kqemu-l4, which activates the accelerator task.
- Execution in the accelerator task will immediately cause a page fault because the page is mapped by the guest but not yet mapped into the accelerator task (shadow page fault). Control will be given to Kqemu-l4.
- Kqemu-l4 resolves the shadow page fault and switches back to the accelerator task.
- The whole sequence is is iterated until enough memory is mapped into the accelerator task that guest code can actually execute.

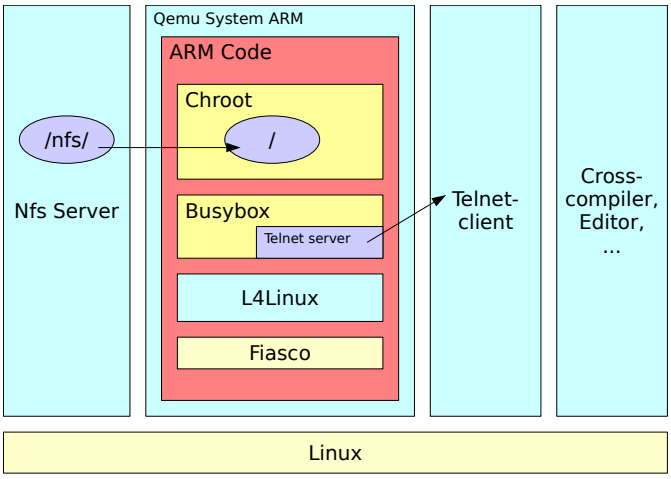
Because the upper GB of memory is occupied by Fiasco, it is not possible for the guest to use any of that memory. Also pages smaller than 4Kb are not supported because Fiasco currently does not support them.

In my work I do not support hardware floating point computations.

4 Implementation

4.1 The Development Environment

Figure 4.1: Development Environment



Since no physical ARM hardware was available as development environment, I used a custom setup with two instances of Qemu. It is important to describe the setup in detail because the understanding of it is crucial to understanding how I did the performance measurements, and what I did to debug my code.

At the base level, I am using an instance of Qemu on my x86 development machine in order to emulate an ARM environment (see figure 4.1). I configured Qemu to emulate an Integrator Board with an ARM926 processor and to provide 256Mb memory. At the time of this writing typical embedded devices do not have large amounts of memory which makes 256Mb reasonable. The ARM Integrator Board is a typical setup for an embedded device. It provides an ethernet device, keyboard and mouse interfaces and an LCD controller.

4.1.1 The ARM Base System

Inside of the emulated ARM environment my setup consists of the following components:

I am running my base system from a custom boot image. It consists of the Fiasco, roottask, L4Linux and several L4Env binaries. Additionally, it includes a small ram disk.

The ram disk is only 3Mb in size and holds a very small Linux base system that is based on Busybox [bus].

Lacking a driver for the LCD controller of the board, I used the serial port for system output. For input, two solutions are possible: direct interaction over the serial port or interaction over the network. Because input on the serial port clashes with the output and only one L4Linux program can make use of it at a time, I opted for the networked solution. Busybox provides a telnet server, which I am using to connect as many telnet clients as needed.

For the Linux base system to use networking, I leveraged a tun/tap device, which establishes a virtual network device having its own IP address that is visible to remote computers.

To solve the problem of how to insert my modified Qemu binaries into the Linux base system, I used a NFS based setup. I created a NFS share on my development PC, which is mounted inside the Linux base system. Installing new binaries is done by copying the file to the NFS share. Additionally, I installed a small Debian Etch base system inside the NFS share. This has the benefit of providing all the commodity Linux tools such as gdb and strace, which are not included in Busybox. By using chroot into the mounted NFS share, I can use a powerful system to run Qemu.

The instance of Qemu that runs on the x86 host and translates ARM code to x86 code will be called **outer Qemu**. The outer Qemu establishes a virtual ARM machine, and all my development will be done inside the outer Qemu.

The other instance of Qemu, which runs inside the virtual environment established by the outer Qemu, will be called **inner Qemu**. The inner Qemu is an ARM binary. It is an L4Linux application, and runs entirely in the virtual ARM environment which is emulated by the outer Qemu.

4.2 Making Qemu Work

When I started my work the inner Qemu did not work on L4Linux. The same binary, which ran fine on pure ARM Linux, exited immediately with a segmentation fault. Careful analysis revealed that the segmentation fault was not actually caused by a fault, but sent by L4Linux because of an error in the signal handler.

It turned out, that Qemu uses a periodic signal, SIGALRM, to emulate timer interrupts and also executes floating point operations. By writing a small program, which did exactly the same, registering a signal handler for SIGALRM and executing floating point operations in an endless loop, I was able to reproduce the problem.

Further investigation showed, that during signal handling the L4Linux main thread was in SVC mode. The signal handler detected this erroneous condition and raised a SIGSEGV to the process which should receive the SIGALRM.

SVC mode is a privileged mode and as such should never be used to run L4Linux. The cause of the erroneous mode could then be traced back to an error in the Fiasco exception handler.

4.3 ARM Implementation of Kqemu-l4

In order to use Kqemu-l4 on ARM, there were three components that needed modification. The interface from Qemu to Kqemu had to be adapted, because it used to be x86 specific. Of course the kernel module Kqemu-l4 had to be ported. Finally the changes Henning Schild implemented for Fiasco had to be ported as well and extended for the special address space layout of the ARM version of Fiasco.

4.3.1 Implementation of the Kqemu Interface for ARM

The interface to Kqemu, which is implemented in Qemu, contains both x86 specific and architecture independent code. All the communication algorithms, for example the code that issues the ioctls, did not have to be changed. However, a lot of the data structures that are used to communicate with the kernel module needed adaption. An example is the state of the ARM processor and the accompanying cp15 coprocessor.

An important function that had to be adapted is *kqemu_is_ok*. Besides checking if Kqemu is enabled, it verifies that the guest is executing code in user mode.

4.3.2 Adapting Kqemu-l4 for the ARM Platform

In Kqemu-l4 most of the interface to Qemu and all of the initialization code did not have to be changed. All the code dealing with memory management such as managing mapped pages and setting pages as dirtied were also left as-is.

All code that deals with segments was removed because the ARM platform does not have segments. Especially the complex testing for segment conflicts is not needed on ARM.

The handling of exceptions and page faults had to be reimplemented for ARM. Both page faults and exception information is to be found in the UTCB of the L4Linux task. Because the UTCB is different on ARM, the functions that copy the guest CPU state to the UTCB and from UTCB back to the guest CPU state had to be rewritten. Of course, the ARM platform has different exceptions than x86, so the algorithms that check for the type of exception and react accordingly had to be designed and implemented as well.

4.3.3 Changes to Fiasco

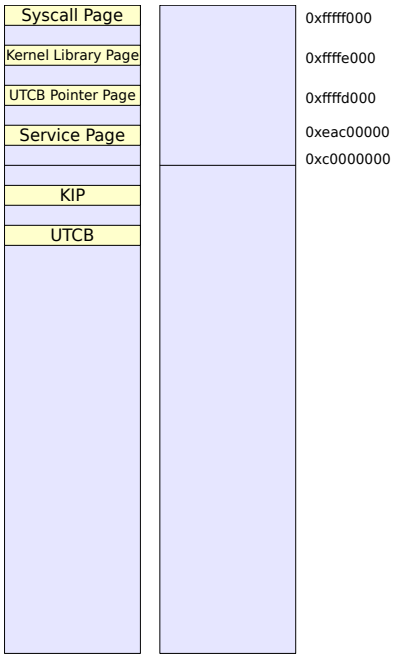
In order to correctly implement memory management for the accelerator task, I had to make sure that all memory accesses to locations that were not explicitly mapped by Kqemu-l4 result in a page fault.

Usually the address spaces of a L4 Fiasco system have pages that are provided by the kernel, and are readable by user space applications, namely the user thread control block (UTCB), the kernel info page (KIP) and a service page.

On ARM additional user accessible pages are mapped into the upper Gigabyte of every address space. A so called syscall page is used as an entry point into the kernel. To do a syscall, user space applications save values into registers and execute a jump to a location in the syscall page and execute a software interrupt (SWI) instruction that resides there. Another user accessible page is called the kernel library page. It holds

code that is used to implement atomic operations known to the kernel. If a preemption occurs while executing this code, the kernel rolls the execution back. The third user accessible page is the UTCB pointer page.

Figure 4.2: Address Spaces, with and without Address Space Artifacts



I marked the address space of the accelerator task with a new *clean* flag. Address spaces with that flag neither contain UTCBs nor the KIP. I altered the behavior of address space switches in Fiasco by adding a new check for the *clean* flag. Whenever Fiasco switches to an address space with the *clean* flag set, the service, UTCB pointer, kernel library and syscall pages are remapped to be accessible to the kernel exclusively. When switching back from the special address space, the original mapping of these pages is restored.

With these modifications all accesses to the user accessible pages in the upper Gigabyte of the virtual address space result in a page fault. Guest pages cannot be mapped there which precludes native execution and requires Qemu assistance.

5 Evaluation

5.1 Performance

5.1.1 Measuring Guest Performance in Qemu

The lack of real hardware requires my measurements to be run in Qemu. To get convincing measurements in an emulator one has to take into account that it is impractical to get precise absolute time stamps. For two reasons: Timers in the emulator rely on the scheduling of the host operating system and therefore depend on the system load of the host, and some effects are not visible because the emulation does not properly capture hardware characteristics.

In my setup I am using two nested instances of Qemu, which means that the time inaccuracies of both instances multiply. Therefore I cannot trust any time values taken in the guest operating system.

The guest instruction count is a much more reliable and deterministic figure. However it can only give an approximation of the real run-time behavior because it does not take into account the different execution times of the instructions. My measurements can therefore only show relative improvements.

The instruction counter in Qemu is implemented in the following way: during translation, Qemu counts all the instructions which go into one TB and saves this information. A little piece of code is prepended to all TBs that increments the global instruction counter by the number of instructions in the TB when the TB is executed.

During my tests I was able to get very precise numbers and therefore I think that I can use the guest instruction counter of Qemu for my measurements.

5.1.2 Instrumentation of Qemu

To get reproducible numbers I have to control the measurements from the guest code of the inner Qemu.

This means that triggering the measurements involves two stages. First, the inner Qemu has to notice that the guest code wants to start the measurement. Then the inner Qemu has to inform the outer Qemu, which in turn has to do the measurements.

I chose to create a magic instruction in order to trigger measurements. Qemu recognizes the magic instruction and can then react accordingly. I implemented this behavior in two steps. I wrote a little helper function, which implements all the behavior I want Qemu to show when the magic instruction is executed. I also altered the translation process, which is done on TB creation, to put a call to my helper function in every TB containing the magic instruction. When the guest executes the magic instruction my helper function will be invoked.

I implemented this in both the inner and the outer Qemu. The helper function of the inner Qemu executes only the magic instruction. That way the helper function of the outer Qemu is invoked, which will then print the current instruction count.

I disabled the timer interrupt in Qemu for the time of the measurements. That way the overhead of scheduling did not influence the results. All measurements were done repeatedly to make sure that there is no influence through page faults.

I measured the overhead of the instrumentation to be only 37 guest instructions.

As a magic instruction I chose an instruction without effects, a so called NOP instruction, which is preceded by loading two special values into registers 1 and 2.

5.1.3 Measurements of Single Instructions

To estimate the speedup of Kqemu-l4, I measured how many ARM host instructions are needed to execute a single ARM guest instruction. I implemented a micro benchmark which executed hand coded assembler code. That way I knew exactly how many instructions were executed in emulation and could then examine how many host instructions were issued. Figure 5.1 shows the measured number of host instructions per guest instruction on the unmodified version of Qemu.

Figure 5.1: Numbers of Host Instructions per Guest Instruction

Guest instruction	Host instructions, unmodified Qemu	Host instructions, modified Qemu
instrumented NOP	37	37
ADD	4	4
MOV	2	2
MUL	3	3
SUBS	20	20
LDRB	11	11
STRB	12	12
B	118	16
BNE	122	20

The instrumented NOP takes 37 instructions because it includes an upcall to a helper function in the inner Qemu that checks two register values and issues another magic instruction to notify the outer Qemu.

During execution of the SUBS a helper function is called, which means that execution steps out of the TB, into Qemu code and back. That is why SUBS takes that many instructions. Interestingly, loads and stores take only a little more than 10 instructions indicating that the soft MMU implementation of Qemu is very streamlined.

One may notice that branches either conditional (BNE) and unconditional (B) take 118 respective 122 instructions in stock Qemu. I expected this number to be much smaller because Qemu does direct block chaining (see 3.5.1). Through further inquiry I was able to get those numbers down significantly by disallowing Qemu to use direct jumps. In the modified instance of Qemu the costs for unconditional branches are 16 and for conditional jumps 20 host instructions.

5.2 Instruction Ratio

Given that the system speedup cannot be measured in my experiment setup, the ratio between instructions executed natively and those executed by the emulator is used as metric. The instruction ratio is computed by equation (5.1). Ir denotes the instruction ratio, N_u the number of instructions executed on the unmodified system and N_m the number of instructions of the modified system. An instruction ratio of one means the system runs as many instructions as the original system, a ratio greater than one means the system needs less instructions and a fraction of instruction ratio one indicates that more instructions are executed.

$$I_r = \frac{N_u}{N_m} \tag{5.1}$$

Unlike system speedup, the instruction ratio does not take different execution times of instructions into consideration. For example a state transition might take longer than an addition. Effects induced for example by different cache footprints or direct and indirect context switch costs are not reflected by the instruction ratio.

The speedup by *Kqemu-l4* on x86-hosts has been measured to be 28 for CPU intensive computations and about 4 for IO intensive computations [Sch08].

An estimation on how big the speedup by *Kqemu-l4* on ARM can be, has to be based on those numbers. However, the ARM platform has different characteristics than the x86 platform. For example the ARMv5 platform has virtually indexed caches, which means that every address space switch has to be accompanied by a cache flush. Later versions of the ARM platform do not have this restriction. In general, x86 is an ISA with a complex instruction set (CISC). So in order to implement the instructions a lot of micro operations are needed. The ARM has a reduced instruction set (RISC) with less complex instructions, so less micro operations are needed. That is why typical RISC binaries are bigger than the CISC binary of the same program.

Differences in the cache architecture cannot be captured by my measurements because Qemu does not model caches. However I expect the mandatory cache flushes on address space switch on the ARM platform to have significant impact on the performance of the system on real hardware.

With all those considerations, I expect my system to reach an instruction ratio of about 20 for CPU intensive tasks. For IO intensive tasks the instruction ratio will be lower.

5.2.1 Measured Figures

To measure the performance of my system, I used three different benchmarks that are implemented as L4 tasks on top of Fiasco and run as guests of the inner Qemu. I implemented a little benchmark application, which implements the search for prime numbers, to measure the speedup for CPU intensive tasks with little IO. To measure the impact of IPC, I used a little ping pong application that contains two threads which constantly exchange information through IPC. As a third benchmark I used an

application which computes fractal images. With this benchmark I wanted to show the performance of a program that both does IPC and CPU intensive calculations.

A graph of the measured number of host instructions per benchmark can be found in Figure 5.2. *Primes* is a benchmark application which searches for prime numbers. *Ping pong* is an application which exchanges information between two threads and *fractal* is an application which computes fractal images. For both CPU intensive benchmarks the acceleration is significant. The amount of instructions needed to run the programs with the use of Kqemu-l4 closely resembles the amount needed when the benchmarks are run directly on hardware. For the ping pong benchmark the amount of host instructions needed with Kqemu-l4 is only slightly lower than the number of host instructions issued when the benchmark is run in emulation only. However, this measurement ran only in an emulated environment, so the additional costs for address space switches which are incurred on real hardware do not show up. In hardware, I expect the mandatory TLB and cache flushes to result in worse numbers in the scenario with Kqemu-l4 enabled.

The instruction ratio computed from the same numbers can be seen in Figure 5.3. The instruction ratio for the primes and fractal benchmarks is 20 and 16, respectively. In the ping pong benchmark only a small portion of code is accelerated. Because it is constantly doing IPC, the overhead incurred on every context switch negates the acceleration.

Figure 5.2: Measured Number of Host Instructions on Different Benchmarks

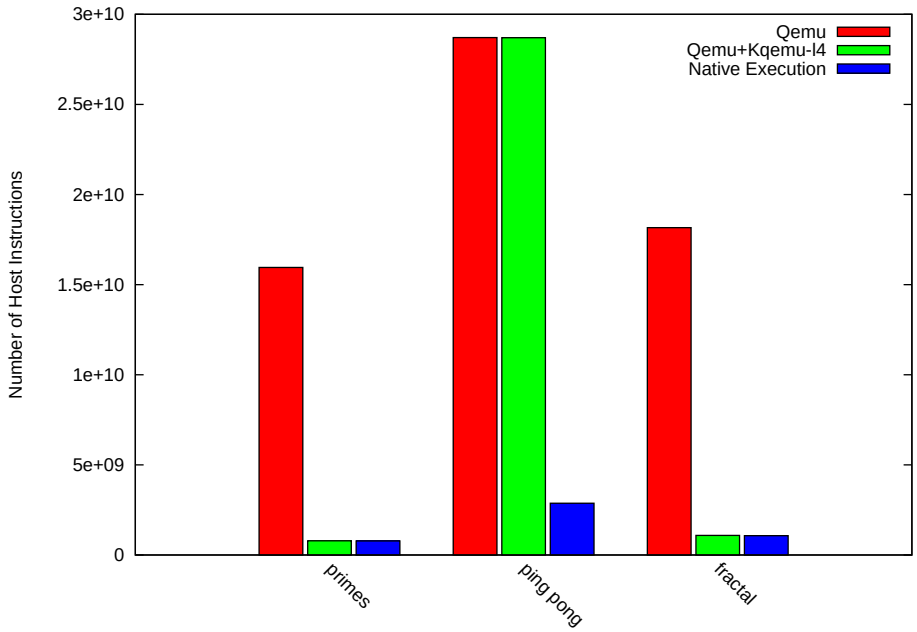


Figure 5.3: Instruction Ratio for Different Benchmarks

Benchmark	Instruction Ratio
primes	20.33
ping pong	1
fractal	16.76

5.3 Evaluation of the Design

5.3.1 Complexity

An option to quantify the complexity of a program is the number of lines of code (LOC). Figure 5.4 shows the LOC I wrote for the different components of my virtualization solution. Kqemu-l4 on ARM is about 500 LOC smaller than on x86 because ARM does not do segmentation. Another reason is that I implemented some checks in Qemu rather than in Kqemu-l4. The absolute number of changes made to Qemu, Fiasco and L4Linux is small. More importantly, the trusted computing base of the system has not increased significantly.

Figure 5.4: Lines of Code Written for Kqemu-l4

Component	LOC
Kqemu-l4	~2000
Qemu	~320
Fiasco	~200
L4Linux	~10
L4Env	~10

5.3.2 Supported Guest Operating Systems

I used a little DROPS system as a testing guest exclusively. Because my testing environment was not powerful enough, I did not try to run Linux or other legacy operating systems. I expect embedded ARM platforms to be comparably slow and to not have a lot of memory, which would be needed in order to run more complex guest operating systems.

Leaving the practical memory and speed limitations aside any commodity operating system designed for the ARM should work.

6 Outlook and Conclusion

6.1 Outlook

Several optimizations of Kqemu-l4 are possible. The management of guest kernel-user transitions in Kqemu-l4 actually accounts for a lot of overhead. It would be possible to have Fiasco resolve shadow page faults by making it aware of a secondary page table that is consulted on a page fault in the accelerator address space. This would significantly reduce the number of address space switches and kernel-user transitions. However, it would increase the in-kernel complexity of the system.

Instead of sharing one accelerator task among multiple guest address spaces it is possible to employ one accelerator task per guest address space. Doing so would significantly reduce the amount of shadow page faults because there would be less contention on memory locations. Both Qemu and Kqemu-l4 would require significant modification to be made aware of the accelerator tasks.

As has been shown by Kinebuchi et al. [KYK+07], Qemu can be ported to run directly on a L4 kernel. So by implementing the functionality of Kqemu-l4 inside of Qemu, the dependency to L4Linux can be avoided, thus saving one address space switch on every context change of the guest. In theory this could reduce the complexity of the VMM. However the amount of work required for porting Qemu and implementing all the functionality of Kqemu-l4 inside of Qemu is significant.

6.2 Conclusion

I was able to show that an efficient full virtualization environment on the ARM platform is feasible. My solution achieves a good speedup compared to emulation, is of low complexity and does not increase the trusted computing base.

The changes to Fiasco and L4Linux are reasonably small, and do not significantly increase the trusted computing base.

The speed improvements compared to emulation are significant and make virtualization feasible on less efficient processors. With the appearance of ARM based netbooks, my virtualization solution could gain even more importance as users will be able to run Windows Mobile side by side with Linux and other ARM based operating systems.

I also see the need for virtualization in embedded devices such as mobile phones. Legacy applications can be reused with little overhead while new user interfaces and applications are implemented as Linux applications. With Android Google created a Linux based OS for mobile phones. With the inclusion of my solution support for Symbian OS and Windows Mobile can be achieved in an efficient manner.

Glossary

API Application programmer interface, 1

DRM Digital Rights Management, 1
DROPS Dresden Real-time Operating System, 5

GPL Gnu general public license (`http://www.gnu.org/copyleft/gpl.html`), 5

inner Qemu Qemu instance which runs inside of the emulated ARM environment, 28
IPC Inter-process communication, 3
IRQ Interrupt request queue, 14

KIP kernel info page, 17

LOC Lines of code, 34

MMU Memory management unit, 6
MPU Memory protection unit, 6

OEM Original device manufacturer, 1
OS Operating system, 2
outer Qemu Qemu instance, which runs on x86 and emulates an ARM environment, 28

PSR Program status register, 6

SOC System on Chip, 6

TLB Translation lookaside buffer, 6

UTCB User thread control block, 17

VM Virtual Machine, 2
VMM Virtual-machine monitor, 2

Bibliography

[ABB+86] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A New Kernel Foundation for UNIX Development. pages 93–112, 1986. 3

[arm] Jazelle Technology. `http://www.arm.com/products/multimedia/java/jazelle.html`. Online, accessed 17-02-2009. 6

[ARM00] ARM Limited. *ARM Architecture Reference Manual*, 2000. 6

[Bel] Fabrice Bellard. QEMU Emulator User Documentation. `http://www.nongnu.org/qemu/qemu-doc.html`. Online, accessed 30-04-2009. 11, 19

[Bel05] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. 2005. 19

[BHWH97] Martin Borris, Michael Hohmuth, Jean Wolter, and Hermann Härtig. Portierung von Linux auf den Microkern L4. 1997. 5

[boc] The Bochs IA-32 Emulator Project. `http://bochs.sourceforge.net/`. Online, accessed 30-04-2009. 11

[BRG+06] Rishi Bhardwaj, Phillig Reames, Russel Greenspan, Vijay Srinivas Nori, and Ercan Ucan. A choices hypervisor on the arm architecture. Department of Computer Science, University of Illinois at Urbana-Champaign, 2006. 11

[bus] About Busybox. `http://www.busybox.net/about.html`. Online, accessed 26-02-2009. 28

[Hoh] Michael Hohmuth. Readme. `http://os.inf.tu-dresden.de/fiasco/download/README`. Online, accessed 12-02-2009. 5

[hyp] Hyper-V. `http://www.microsoft.com/windowsserver2008/en/us/hyperv-main.aspx`. Online, accessed 30-04-2009. 8

[jpc] JPC - Computer Virtualization in Java. `http://www-jpc.physics.ox.ac.uk/`. Online, accessed 30-04-2009. 11

[KYK+07] Kinebuchi, Yuki, Koshimae, Hidenari, Nakajima, and Tatsuo. Constructing machine emulator on portable microkernel. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 1197–1198, New York, NY, USA, 2007. ACM. 37

[Lac] Adam Lackorzyński. L4Linux overview. `http://os.inf.tu-dresden.de/L4/LinuxOnL4/overview.shtml`. Online, accessed 12-02-2009. 5

[Lac04] Adam Lackorzyński. L4Linux Porting Optimizations. Master's thesis, University of Technology Dresden, 2004. 5

[Lie96a] Jochen Liedtke. *L4 Reference Manual*, September 1996. None. 3

[Lie96b] Jochen Liedtke, 13c44c. Toward real microkernels. *Commun. ACM*, 39(9):70–77, 1996. 3

[PG74] Gerald J. Popek and Robert P Goldberg. Formal Requirements for Virtualizable Third Generation Architectures. 1974. 7, 8

[qem] QEMU - open source processor emulator. `http://www.nongnu.org/qemu/about.html`. Online, accessed 20-04-2009. 11

[Sch08] Henning Schild. Sichere Virtualisierung auf dem Fiasco-Mikrokern. Master's thesis, University of Technology Dresden, 2008. 12, 19, 33

[sgTD] Operating systems group TU-Dresden. Overview on L4 implementations. `http://os.inf.tu-dresden.de/L4/impl.html`. Online, accessed 12-02-2009. 4

[tru] Trustzone Technology Overview. `http://www.arm.com/products/security/trustzone/`. Online, accessed 12-02-2009. 11

[vir] VirtualBox. `http://www.virtualbox.org/`. Online, accessed 30-04-2009. 8

[vmwa] Vmware esx. `http://www.vmware.com/products/vi/esx/`. Online, accessed 30-04-2009. 8

[vmwb] VMware Workstation. `http://www.vmware.com/products/ws/`. Online, accessed 30-04-2009. 8

[xen] Xen hypervisor. `http://www.xen.org/`. Online, accessed 07-04-2009. 9