THE UNIVERSITY OF NEW SOUTH WALES
SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

Implementing Hardware-supported Virtualization in OKL4 on ARM

Prashant Varanasi

Thesis submitted as a requirement for the degree

Bachelor of Science, Honours (Computer Science)

Supervisor: Gernot Heiser

Submitted: November 30, 2010

Abstract

Virtualization is an already popular trend in the desktop and server markets, and is becoming increasingly important on mobile devices, where ARM is the leading architecture. This thesis presents the design and implementation of a hypervisor integrating ARM's recently announced virtualization extensions. This hypervisor is capable of running multiple concurrent unmodified guest operating systems such as Linux, and supporting efficient communication between guests. Benchmarking in ARM's Fast Models simulator show that virtualization overheads are small, although true overheads cannot be measured till hardware or a timing-accurate simulator is released.

Acknowledgements

I would especially like to thank my supervisor, Gernot Heiser, for his support and feedback throughout this thesis, and for the opportunity to undertake such an interesting thesis.

I would also like to thank the staff at Open Kernel Labs for supporting me throughout this thesis; Malcolm for helping me understand the OKL codebase, Carl for sharing his thorough knowledge of the ARM architecture, Peter for helping with porting and Daniel Potts for making sure I had access to everything I needed.

I would like to thank the many people at ERTOS who were always willing to help.

I would also like to thank ARM for responding to my queries and problems promptly.

Contents

List of Figures

Chapter 1

Introduction

Virtualization is the latest trend in many areas of computing as processing power increases. It allows consolidation of multiple resources, saving costs in hardware and power. Virtualization was invented in the 1960s to overcome the limits of a single-user operating system, so multiple user applications could be run at the same time. Interest in virtualization declined once multi-user operating systems became popular, which allow user applications to run using virtual memory. However, under-utilised servers have brought back virtualization to increase efficiency and reduce costs through consolidation of multiple servers while maintaining quality of service and isolation.

Servers are increasingly running multiple guests using virtualization to allow multiple separate services to run in the same physical machine, lowering the hardware requirements by taking advantage of the unused computing power. Personal workstations use virtualization for many reasons, from isolation of services to helping with cross-platform development.

Virtualization in embedded devices has recently taken off, with Open Kernel Lab's virtualization software in more than 750 million devices [OKL10]. Current virtualization solutions for ARM use para-virtualization, which requires modification of the guest operating systems [Hei09, HSH+08]. However, ARM's recently announced hardware extensions allow us to virtualize unmodified guest operating systems, and aim to reduce the performance overhead traditionally associated with pure virtualization. This thesis aims to implement virtualization in an existing microkernel, and use this implementation to evaluate the ARM virtualization extensions compare to para-virtualization and native execution.

1.1 Motivations

Rapid increases in processing power have left many machines under-utilised, with some reports claiming that servers are only 10% utilised [CB07]. Virtualization has been selected as a way to cut costs, as it allows hardware to be consolidated, and use less energy, without weakening the isolation properties of multiple machines. Virtualization saves costs by reducing the hardware that needs to be maintained and replaced.

Virtualization has moved from data centres into other areas such as standard desktops [MP07], the mobile space [Hei08], and the software delivery process, by distributing complete packages of running virtual machines [Kro09], saving the user the trouble of assembling the correct dependencies and versions of software manually.

An example use of virtualization is in mobile devices, where the modem software needs to be strongly isolated from the high-level operating system (often Linux). This isolation is traditionally achieved using separate CPUs and RAM for each system. Virtualization can cut hardware costs and reduce power usage by sharing a single CPU and RAM between these systems, while ensuring the modem stack is both isolated and scheduled in a real-time manner. If multiple cores are used, a hypervisor can achieve better power management by turning off idle cores, and by scaling loads between active cores, rather than restricting each system to only use their dedicated cores.

The work done in this thesis is also relevant to the desktop and server markets, which ARM is starting to target with the introduction of the Cortex A15 core, which includes 64-bit addressing, virtualization extensions, and higher clock speeds [ARM10a].

While para-virtualization has been the main focus for current hypervisors in the embedded space, this thesis will focus on the use of ARM's new virtualization extensions to implement pure virtualization. This alternative will allow for closed-source operating systems to run under the hypervisor, while also allowing open-source operating systems such as Linux to run unmodified, cutting the costs of OS modification.

1.2 Goals

The primary goal of this thesis is to develop a hypervisor that is able to concurrently run multiple instances of unmodified Linux. The virtualization overheads can then be measured by comparing it to Linux running natively, and para-virtualized on a high-

performing microkernel such as OKL4. However, performance numbers will be at best indicative as benchmarks will be run in ARM's Fast Models simulator, which does not provide accurate timings for memory or cache.

A secondary goal is to achieve efficient communication between virtual machines. This is useful for shared devices, where a device driver is hosted in one guest, and other guests communicate with the driver host to make use of the device.

To achieve these goals, I must:

- Familiarise myself with the ARM architecture internals, such as the VM system and the interrupt controller.
- Understand the impact of the ARM virtualization extensions on the existing architecture.
- Write a simple prototype hypervisor that uses the virtualization extensions to host a single guest.
- Integrate the virtualization extensions into the OKL4 microkernel, based on the simple prototype, while adding support for multiple guests.
- Implement communication between guests using hypercalls.
- Implement a shared device scenario with a driver hosted in one guest, and a Linux driver which communicates with the driver host running in a separate guest.
- Benchmark the resulting system running Linux against native and para-virtualized Linux.

The focus will be on running Linux virtualized, since getting a real-world operating system provides more information on real-world workloads and gives us a more thorough understanding of the virtualization extensions than some simple test applications. Due to time restrictions, this thesis will focus on the core devices required to get Linux running and will not emulate more complex devices such as Ethernet or VGA. These devices can still be used in a single guest by passing them through directly.

1.3 Thesis Overview

Chapter 2 provides background information relevant to the thesis, such as the different virtualization techniques, an overview of virtualization options on the ARM architecture, hardware extensions introduced to aid virtualization, and how hardware extensions can be used in a microkernel. This is followed by a brief overview of the ARM architecture and the new virtualization extensions.

Chapter 3 describes how the ARM virtualization extensions will be used to provide pure virtualization, and how these extensions will be embedded in the OKL4 microkernel.

Chapter 4 details implementation milestones achieved, and issues faced in achieving them.

Chapter 5 evaluates my implementation, and measures the performance overhead of pure virtualization compared to native Linux and para-virtualized Linux.

Chapter 6 briefly discusses what has been achieved, evaluates the virtualization extensions architecture, and proposes ideas for future work.

Chapter 2

Background

2.1 Virtualization Techniques

Virtualization refers to partitioning a single physical machine into multiple virtual machines, which are efficient, isolated duplicates of the real machine. The *hypervisor* is software running on the physical hardware to share resources (e.g. memory, CPU, devices) between the virtual machines. The hypervisor executes in privileged mode, while the guests are run de-privileged, as can be seen in Figure 2.1.

We first discuss a classification of instructions introduced in Popek and Goldberg's seminal paper [PG74]:

- *sensitive instructions*, instructions that attempt to change the configuration of resources in the system, or whose behaviour or results depends on the configuration of resources;
- *privileged instructions*, instructions that trap if executed in an unprivileged mode, but execute without trapping when run in a privileged mode.

The paper also defines *virtual machine monitor (VMM)* as the software that provides

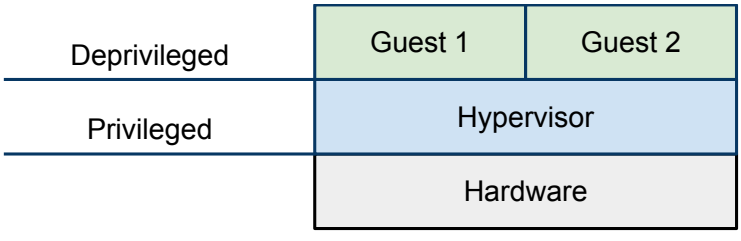


Figure 2.1: Hypervisor runs privileged while guests are deprivileged

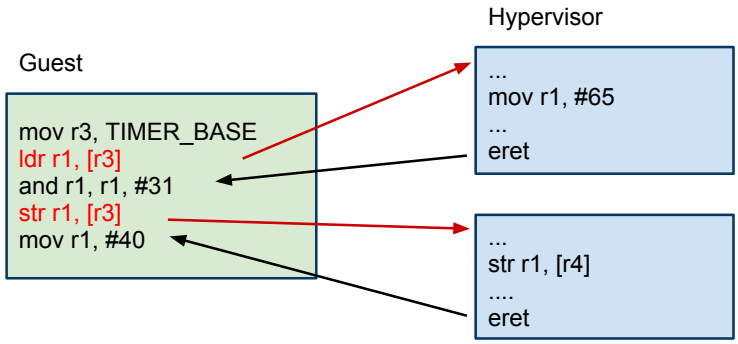


Figure 2.2: How trap-and-emulate works

the abstraction of a virtual machine. Both *VMM* and *hypervisor* refer to the same concept, although we use the term hypervisor throughout this thesis.

Popek and Goldberg claimed a hypervisor could be constructed for an architecture if the sensitive instructions are a subset of the privileged instructions; this criterion has now been termed *classically virtualizable*.

Following is a list of techniques used for virtualization depending on the architecture and particular situation:

- *Pure virtualization*, which is only possible if the architecture is classically virtualizable, relies on the hypervisor emulating all sensitive instructions on a trap; a technique called *trap-and-emulate*. When a guest tries to accesses privileged resources, the hardware generates a trap, invoking the hypervisor. The hypervisor emulates the access, then returns to the next instruction. This is shown in Figure 2.2, with the red arrows representing exceptions raised by the hardware. Trap-and-emulate can cause significant overheads as each privileged instruction is emulated with many more instructions.

- *Binary rewriting* is used when pure virtualization is not possible, because the architecture is not classically virtualizable. This technique relies on scanning the guest binary at load time or runtime, and replacing sensitive instructions that do not trap with either a trap, or emulation without using sensitive instructions. This is shown in Figure 2.3.

- *Para-virtualization* refers to a modified higher-level API being presented to the guest and the guest modified to replace sensitive instructions with calls into the

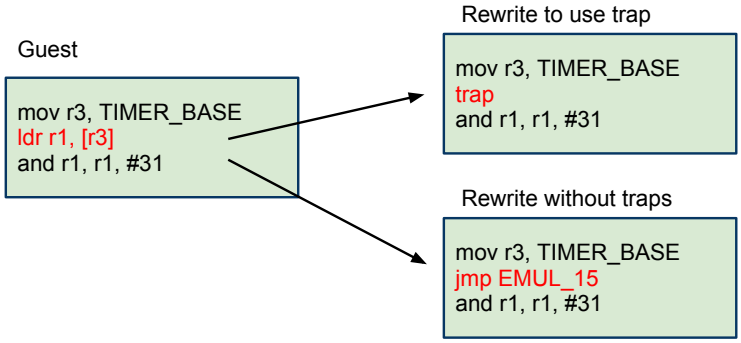
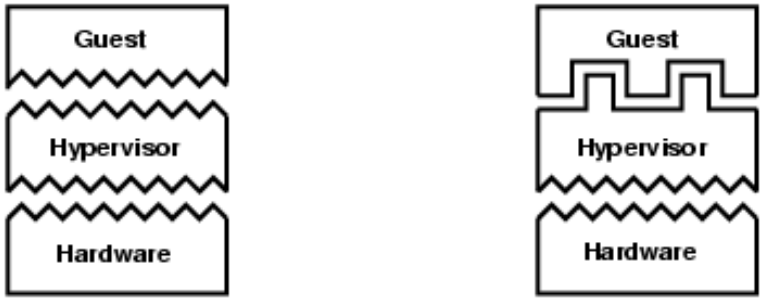


Figure 2.3: Binary rewriting used to replace sensitive instructions

hypervisor (also called *hypercalls*). Figure 2.4 shows that both the hypervisor and the hardware expose the same API in pure virtualization, but different APIs in para-virtualization.



(a) Pure virtualization (b) Para-virtualization

Figure 2.4: Pure virtualization vs para-virtualization

2.1.1 Pure virtualization and binary rewriting

Both pure virtualization and binary rewriting do not modify the machine API, so any guest that runs natively on the architecture will also run virtualized using these techniques. However, since a trap is caused on every privileged instruction, the total time taken for privileged instructions is much higher than when the instructions are run natively.

Architectures such as x86 and ARM, which are not classically virtualizable, can use binary rewriting to achieve virtualization, as VMWare has shown on x86 [VMW08]. Binary rewriting, when optimised, can virtualize architectures such as x86 with a reasonably small overhead (less than 10%) [VMW06], but the techniques used are quite complex. This complexity increases the size of code running in the highest privileged

mode, increasing the attack surface and chance of bugs which can undermine the security and isolation properties of the whole system.

2.1.2 Para-virtualization

Para-virtualization is not a new idea, although the term was only recently introduced in reference to the Denali virtual machine monitor in 2002 [WSG02]. The concept was implemented in the 1970's in IBM's CMS system, using a *DIAG* instruction to call the hypervisor, and has been used ever since, e.g. Mach [ABB+86], Xen [BDF+03] and L4 [HHL+97].

Para-virtualization is able to achieve better performance than pure virtualization due to the direct use of an API instead of multiple traps, instruction decoding, and hardware emulation. However, it has a major drawback—guest operating systems must be modified to use the new API, which can be a large task. The need to modify the operating system also means that a closed-source operating system cannot be modified by anyone other than the original vendor.

2.1.3 Virtual memory in pure virtualization

Virtual memory systems use the *memory management unit* (MMU) to convert a virtual address to a physical address. Translations are often performed by a hardware page-table walker that walks a page table set up by the operating system to convert a given virtual address to a physical address.

However with virtualization, the guest page table no longer maps virtual addresses to physical addresses, but only to guest physical addresses. The hypervisor has its own internal mapping of guest physical memory to actual physical memory, as can be seen in Figure 2.5. Since the MMU traditionally only performs a single translation, the hypervisor builds and exposes a page table that maps guest virtual addresses to actual physical addresses. The hypervisor needs to trap on any access to guest page tables, and then build shadow page tables that map the guest processes to actual physical memory. This shadow page table is then walked by hardware for virtual memory translations.

This process is inefficient as it requires trapping on every single page-table access, and so virtualization extensions from Intel and ARM add support for two stages of address translation. The first stage translates guest virtual to guest physical addresses, and

the second stage allows the hypervisor to set up page tables that map guest physical addresses to a different real physical address. This method is more efficient for setting up the page table, as it does not require shadow page tables or traps on each page-table access. However, when a mapping is not found in the *translation lookaside buffer* (TLB), a performance overhead is incurred from the extra memory accesses required to walk the second-level page table.

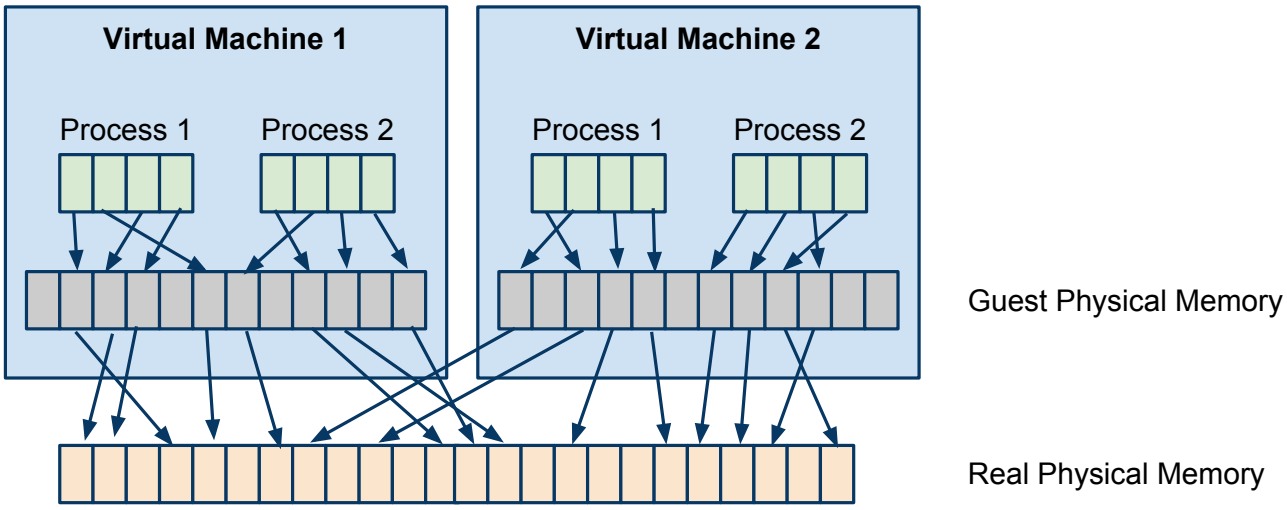


Figure 2.5: Virtual memory with pure virtualization

2.2 Virtualization on ARM

There are many products available for virtualization on the ARM platform such as Open Kernel Labs OKL4 [OKL10], Green Hills Integrity [Gre10], VirtualLogix VLX [Vir10] and Xen-ARM [HSH+08]. The architecture is not classically virtualizable, and so, para-virtualization is used to provide virtualization by all of the current products. Binary rewriting is not employed as it is less efficient than para-virtualization, and much more complicated to implement. The current leader in performance is the OKL4 microvisor [HL10], which merges the common concepts of a hypervisor and a microkernel.

There are multiple ports of Xen to ARM. The first was Ferstay [Fer06] which set up the basics of an ARM port, but it only ran a simple operating system rather than a a full high level operating system. Samsung then performed a full port of Xen to ARM [HSH+08], but performance has suffered significantly, being on average twice as slow as native.

Our implementation will be compared with OKL4, as OKL4 is much better performing than Xen-ARM and it is of production quality. It is expected that the OKL4 microvisor will perform better than our implementation, due to the use of para-virtualization. Our implementation will not require guest operating systems to be modified, which saves development time, and also allows closed source operating systems, such as iOS (the iPhone's OS), to be virtualized.

2.3 Hardware Extensions

Architectures that are not originally classically virtualizable, such as x86, have introduced extensions to allow traps for all sensitive instructions making them classically virtualizable [FO06].

Intel's virtualization extensions [UNR+05] are labelled Intel VT-x (Virtualization Technology), while AMD's extensions are labelled AMD-V. Both offer similar functionality [FO06], and so only Intel's technology is discussed. VT-x adds:

- separate modes of CPU operation for the hypervisor (VMX root operation) and the guest (VMX non-root operation);
- many configurable traps for sensitive instructions and events;
- storage of guest and hypervisor state (e.g. page-table pointer, interrupt descriptor table), and automatically switching state on VM entries and exits;
- (added later) extended page tables to perform the second-stage of translation in hardware;
- (added later) tagged TLBs with virtual machine identifiers to avoid flushing on every VM entry and exit.

The initial extensions made x86 classically virtualizable, but the lack of support for MMU virtualization meant that performance was worse than binary translation techniques [AA06]. Additionally, the lack of TLB tagging meant that the TLB was flushed on every entry and exit, which slowed down performance significantly, especially since every sensitive instruction causes a trap [FO06]. More features were thus added to reduce the

performance overhead. Further hardware extensions (Intel's VT-d in this case) introduced safe *direct memory access* (DMA) between devices and memory.

ARM is introducing hardware extensions [ARM10d] that allow the ARMv7 architecture to be virtualized using trap-and-emulate techniques. The extensions bring a new *hyp* mode as the highest privileged mode, a second-stage of translations, hardware support for virtualized interrupts, and some extra features to simplify virtualization. These extensions will be discussed in more detail in Section 2.8. Since the ARM extensions are similar to the x86 extensions in many ways, a comparison is made in Section 2.9.

2.4 Microkernels and Hypervisors

Microkernels aim to provide a minimal API with a focus on running minimal code in the highest privileged mode, while providing the required functionality for all other components such as drivers and the OS personality, to run in user mode [Lie95]. The aim is for a system to be completely component-based, such that all services are provided by user applications which communicate with each other via *inter-process communication (IPC)*. Since everything in the system is a service, and users must communicate to each other via IPC, Liedtke showed that a thin IPC layer could yield large real-world gains [Lie93]. He used this knowledge to develop L4, where high-performance of IPC was one of the primary goals.

Hypervisors aim to run multiple guests on a single system. Hypervisors either use the machine traps as the API (in the case of pure virtualization), or define extra *hypercalls* as the API, intended to provide a higher level yet simple abstraction of hardware.

The conceptual basis of microkernels and hypervisors have a high enough overlap [HL10] that microkernels are often used for virtualization [HHL+97, OKL10]. This thesis merges the two by implementing a hypervisor using a microkernel as a base. Currently, the microkernel API is only functional until the hypervisor syscall is invoked; however, this is only an implementation limitation.

2.5 Hardware Extensions in a Microkernel

Biemüller [Bie06] has integrated Intel's VT-x hardware extensions in L4 to provide full virtualization. Their thesis achieves full virtualization by making some changes to the

microkernel and using a user application to manage the guest. The virtualization is broken down into the following parts, shown in Figure 2.6:

- The virtual machine itself, executing standard unmodified guest binaries in VMX non-root mode, causing traps on all sensitive instructions.
- The L4 microkernel, with new support for running in VMX root mode, and modifications to allow faults and interrupts to be passed to a separate user application via IPC.
- A monitor application that runs at user level that sets up the actual virtual machine, and emulates all sensitive instructions as traps come in through IPC from the kernel, similar to NOVA [SK10].

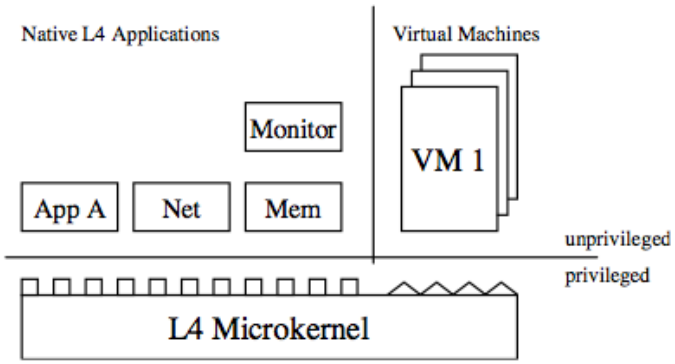


Figure 2.6: The virtual machine and monitor both run unprivileged

VT-x page tables were not available at the time, so the hardware extensions were only used to make the architecture classically virtualizable. The modifications in the microkernel allowed emulation and management of the virtual machine to occur at the user level. Unfortunately, performance numbers for their project have been removed and are unavailable.

The concepts in Biemüller's thesis are quite different to the approach undertaken in this thesis and so provide a good comparison. Both my thesis and Biemüller's thesis require modifications at the kernel level to run in the highest privileged modes (VMX root mode for x86, hyp mode for ARM), and both require the microkernel to handle traps. However, much of the emulation work has been moved to the user level in Biemüller's thesis, while this thesis leaves the emulation in the kernel. This is because the ARM extensions (described in Section 2.9) allow for a simpler hypervisor. We claim that the

simpler hypervisor would have a minimal effect on the *trusted computing base (TCB)*, the code that is running in the highest privilege mode that is essential to the security of the system. However, an x86 hypervisor would be much larger and more complex, as shown in NOVA which uses 27KLOC of code for the microhypervisor and user monitor. Future work can look at the impact of moving the hypervisor into user mode.

2.6 ARM Overview

We first look at some components of the standard ARM architecture, to help understand the impact of the virtualization extensions on these components. More information is available in the reference manual [ARM05].

2.6.1 Introduction to ARM

The ARM architecture is a 32-bit *reduced instruction set computer* (RISC) architecture. The term *reduced* refers to the amount of work done in each instruction, in contrast to *complex instruction set computer* (CISC) architectures. Most instructions complete in a single clock cycle, and it is a *load-store architecture*, containing separate data processing and I/O instructions. Data processing instructions only operate on register values, unlike CISC architectures.

The architecture contains 16 32-bit registers, some of which are banked for different modes. All ARM instructions are fixed-width at 32-bit, although ARM also supports alternative instruction sets which will be discussed later. Some notable features of the architecture include conditional execution, a barrel shifter, and the use of co-processors to implement extensions such as floating point operations, or control operations such as managing caches or setting up the MMU. These will be discussed in more detail in Section 2.6.4.

The virtualization extensions are implemented as an extension to the ARMv7 architecture; any references to the ARM architecture in this thesis thus imply this version.

2.6.2 Thumb instruction architecture

The ARM architecture includes support for an alternate instruction set to achieve higher code density, called Thumb. Initially, the Thumb instruction set was fixed-width with

16-bit instructions, which meant less flexibility and less functionality due to the small instruction size. There were many restrictions on the operands for instructions, while also limiting features such as conditional execution to only work on branches.

A newer version of Thumb, Thumb-2, was introduced to achieve code density similar to Thumb, with the performance and flexibility of ARM instructions. Thumb-2 adds some 32-bit instructions which allow it to support more features of the ARM instruction set such as conditional execution, while still achieving higher code density than the ARM instruction set.

2.6.3 Co-processors

The ARM architecture uses co-processors to extend the architecture without adding new instructions or registers. The architecture allows up-to 16 coprocessors, with co-processor 15 (CP15) reserved for typical control functions.

CP15 controls overall system configuration, cache and TLB management, MMU operations, and system performance monitoring. We discuss memory management on ARM in the following section.

2.6.4 Virtual memory system

The ARM architecture uses a hardware page-table walker which requires page tables to be in a specific format. Address translations performed by a walk are stored in the TLB. Every memory access is first checked in the TLB.

If the TLB contains an entry for the given virtual address and ASID, the physical address is returned immediately. However, if the TLB does not contain an entry for the given address, the following is performed:

1. the MMU performs a page-table walk for the given virtual address;
2. once a mapping is found, a TLB entry is inserted;
3. the physical address of the mapping is returned.

The ARM architecture specifies a two-level page table, with each table containing 32-bit entries. Page-table entries contain the physical frame the address maps to, access permissions including whether the page is executable. The page table also supports *block* mappings of 1MiB, which only require the first level of the page-table walk.

2.6.5 Processor modes and TrustZone

The ARM architecture specifies 8 processor modes, which are either privileged or unprivileged:

- privileged: FIQ, IRQ, supervisor, monitor, abort, undefined, system;
- unprivileged: user.

The operating system runs privileged, while applications normally run in user mode. The virtualization extensions require a processor with the ARM TrustZone extensions, so we will look at the architecture presented by TrustZone. TrustZone further separates the execution state into two separate worlds:

- *secure world*: used for running all trusted software;
- *non-secure world*: for running any untrusted code.

These worlds are orthogonal to the processor modes, and software can run in any of the processor modes, in either world.

A single core is able to securely execute code from both worlds, while ensuring that secure-mode software can be protected from the non-secure world, through partitioning of hardware and memory. The secure world controls all partitioning of devices, including interrupts and co-processor access. A new processor mode, *monitor* mode, which is run in the secure world, and is used to switch between non-secure and secure worlds. An overview of TrustZone is shown in Figure 2.7.

Virtualization can be implemented on top of TrustZone as it allows partitioning of memory, interrupts and ensures that privileged software in the non-secure world cannot access or modify configuration of software running in the secure world. However, without para-virtualization, only a single guest can be virtualized, with the hypervisor running in the secure world, and a single guest running in the non-secure world. Green Hills' INTEGRITY is an example of a hypervisor which uses TrustZone in this way [Gre10].

2.6.6 Interrupt controller

We discuss the *generic interrupt controller (GIC)* which is the interrupt controller used in the RealView emulation baseboard, and is supported by the virtualization extensions.

The GIC is split into two separate components:

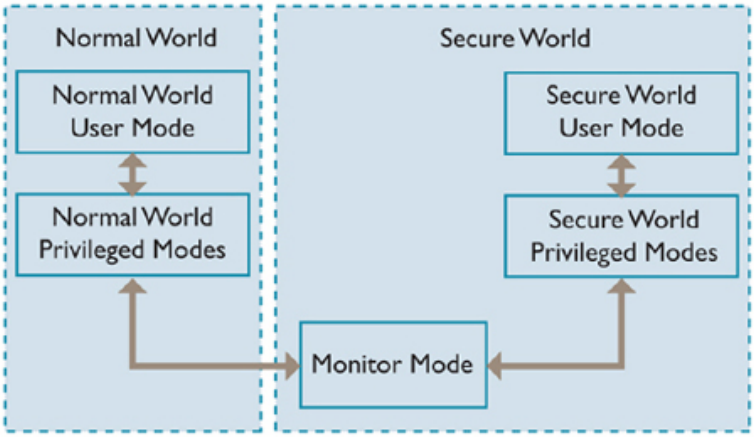


Figure 2.7: Separation of worlds using TrustZone [ARM10b]

- *Distributor*: the hardware component that receives the interrupt, and controls which interrupts are enabled, their priorities, and distribution of interrupts to a CPU interface.

- *CPU interface*: the hardware component which performs interrupt priority masking and preemption handling for each connected CPU.

When a device raises an interrupt, the interrupt is routed through the distributor to the correct CPU interface based on the distributor configuration. The CPU interface raises the interrupt on the processor if the new interrupt priority is higher than the current mask, and if interrupts are enabled. The CPU interface is used by software to acknowledge and clear the interrupt.

The CPU interface keeps track of what interrupts are active on the CPU, and ensures that lower priority interrupts cannot preempt a higher active interrupt.

2.7 Virtualization Issues with the ARM Architecture

The standard ARM architecture without the virtualization extensions is not classically virtualizable—there are many sensitive instructions that do not trap when executed in an unprivileged mode. An example instruction which does not trap is *CPS*, change processor state. When this instruction is executed in user mode, it has no effect and does not cause a trap.

Even if it were possible to trap on all sensitive instructions, virtualization on ARM would still be impractical due to the overhead of using trap-and-emulate techniques for

commonly used privileged resources such as the virtual memory subsystem, interrupt controller, and co-processors.

Since the ARMv7 architecture uses a hardware page-table walker, virtualization would require shadow page tables, and all accesses to the page table would need to be trapped. Similarly, the interrupt controller would need to be emulated completely, and this would cause significant overhead under workloads with a high interrupt frequency.

To overcome these virtualization issues, ARM has introduced virtualization extensions which we discuss below.

2.8 Hardware Extensions Overview

ARM's virtualization support has been added as an extension to the ARMv7 architecture, and requires TrustZone to be implemented. The basic model for virtualization using these new extensions is described below:

- The hypervisor runs in a new non-secure mode, called *hyp* mode. This mode allows the hypervisor to manage all other non-secure modes.
- The guest operating systems run in the non-secure privileged and user modes.

The extensions add features to make pure virtualization possible, and aim to improve the speed of virtualization. The important features are discussed below:

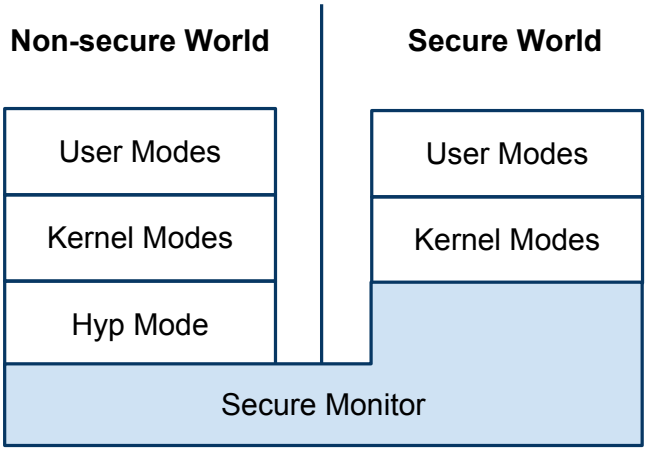


Figure 2.8: Separation of modes

- New mode: *hyp* mode runs in the non-secure world, and is the highest privileged mode, as can be seen in Figure 2.8. It is used to manage the guest operating systems.

The hypervisor, interrupt handling, and traps set up by the hypervisor are all run in this new mode. This mode is used to separate the hypervisor from the running guests, which continue to run in the non-secure kernel modes.

- Second-stage of translation: subjects all guest physical addresses to be translated to physical addresses mapped by the hypervisor, which avoids the use of shadow page tables as discussed in Section 2.1.3.

- Interrupt handling support: new hardware has been added to avoid emulating an interrupt controller, which would add substantial overhead to processing each interrupt. The hardware support removes the need to trap on common operations such as acknowledgement and clearing of interrupts.

- Emulation support: adds features to reduce the overhead of a standard trap-and-emulate approach by providing extra information to the hypervisor on a fault about the read or write, removing the need for the hypervisor to fetch and decode the faulting instruction. Since device emulation requires trap-and-emulate, reducing overhead is important for implementing virtual devices efficiently.

- Configurable traps: support for traps when accessing functionality that a hypervisor may need to intercept. Since we may not want to trap on all events, configuration allows us to only trap on events we need, which reduces the number of traps and reduces overhead.

2.8.1 Second-stage translations

We first describe the different address types encountered in virtualization:

- *guest virtual*: virtual addresses generated by the guest;

- *guest physical*: physical addresses generated by translating guest virtual addresses using the guest page table;

- *physical*: actual physical addresses which must be used to access memory and devices.

The virtualization extensions add support for a second-stage of translation, which allows guest physical addresses, termed by ARM as *intermediate physical addresses* (IPA),

to be subject to a second-stage of translation which converts it to the real *physical address* (PA). The virtualization extensions allow us to disregard the guest virtual addresses which are translated as normal and are unaffected by the virtualization extensions, as can be seen in Figure 2.9.

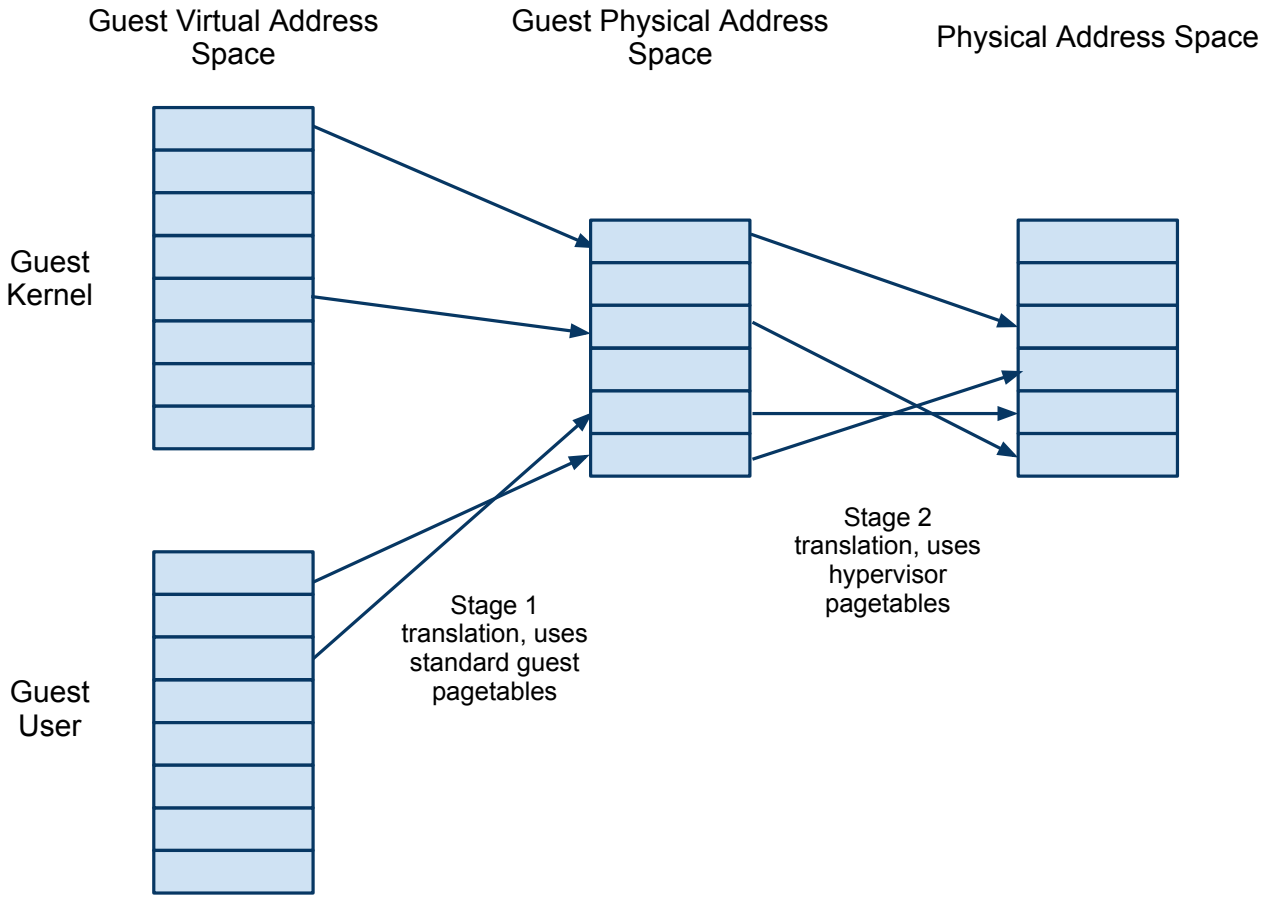


Figure 2.9: Overview of the second-stage translations

A second page-table walk is required for the second-level translations, with options for block mapping (2 MiB blocks), or page mapping (4 KiB blocks, a standard page). A new page-table format is used for second-stage mappings, described in Section 2.8.5. This second page-table walk causes a performance overhead, and so to minimise page-table walks, the TLB has been modified to store guest virtual address to physical address mappings.

2.8.2 Virtual interrupts

Interrupt handling has been significantly modified to deal with the introduction of virtual interrupts. Hardware modifications allow the guest to acknowledge and clear interrupts without the use of trap-and-emulate. This is done by creating a new hardware component, the *virtual CPU (VCPU) interface*, that can be mapped into the guest as the CPU interface, which avoids the use of trap-and-emulate to emulate the CPU interface. However, the interrupt distributor must be emulated using standard trap-and-emulate techniques, but this it not a big concern as the distributor is only modified initially to set up enabled interrupts, and is not modified often thereafter.

When interrupt routing is enabled, all interrupts are sent to the hypervisor, which can then raise virtual interrupts on the current guest using the VCPU interface. A virtual interrupt can be linked to a physical interrupt, allowing the guest to clear the physical interrupt without hypervisor intervention.

A brief interrupt handling scenario with an interrupt destined for a guest is shown in Figure 2.10 and described below:

1. a device raises an interrupt on the distributor;
2. the distributor routes the interrupt to the CPU interface;
3. the CPU interface routes the interrupt to the hypervisor;
4. the hypervisor checks the interrupt, finds that it is destined for the guest, and generates a virtual interrupt, linked to the physical interrupt, and adds it to the VCPU interface;
5. the VCPU interface generates an interrupt on the guest based on the interrupt added by the hypervisor;
6. the guest acknowledges and clears the interrupt on the VCPU;
7. the VCPU finds that this virtual interrupt is linked to a physical interrupt, and clears the linked physical interrupt on the interrupt distributor.

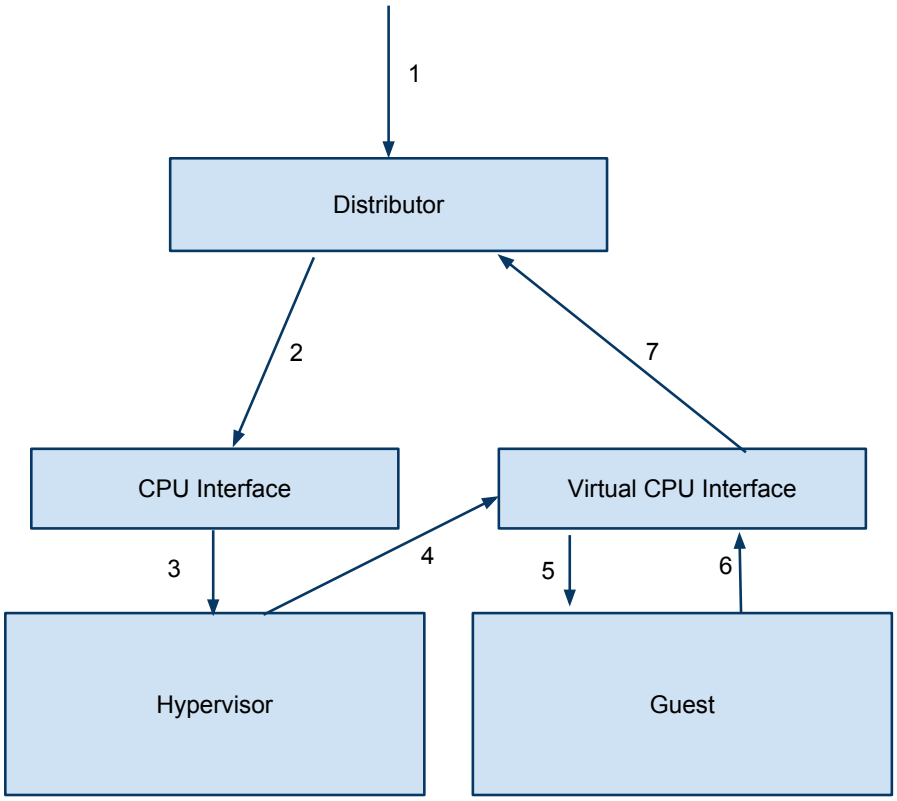


Figure 2.10: Virtual interrupt overview

Priority Drop

ARM has introduced a priority drop mechanism to ensure that interrupts destined for one guest do not block interrupts for other guests. Normally, when an interrupt is acknowledged, further interrupts cannot preempt the system unless they are of higher priority, until the interrupt is cleared. With virtualization however, we do not want any guest interrupt to block other interrupts, and we do not want to clear interrupts as they may need to be processed by guests first. The priority drop mechanism addresses this issue, as it only drops the running priority of the system, allowing other interrupts to preempt the system.

2.8.3 Emulation support

Virtual device support requires the hypervisor to use trap-and-emulate techniques to emulate the device. Standard trap-and-emulate techniques can have a large overhead as

the instruction being emulated needs to be fetched, decoded, then emulated. Fetching an instruction can be an expensive operation as the faulting instruction would, at best, be in the I-cache, whereas to decode it in software, it must be fetched into the D-cache. This means an L1 miss in the best case. Decoding the instruction then requires a simple emulator, which means a large amount of code in the hypervisor.

The virtualization extensions add support for acceleration of the more common load/store scenarios, by adding extra information about the faulting instruction:

- read or write instruction
- size of the read or write
- source or target register
- size of the faulting instruction.

This information obviates the need to fetch and decode instructions on most traps.

2.8.4 Other hardware features

A new identification concept for virtual machines is added, the *virtual machine identifier (VMID)* register, to identify each running guest. The VMID is also used as a tag in the TLB, allowing TLB entries from multiple guests to co-exist, avoiding the need to flush the TLB on a world switch.

Many new configurable traps are also provided to allow the hypervisor to ensure that any sensitive instruction can be trapped, although some traps may be disabled for performance reasons. Instructions that access virtualization-sensitive state that can be trapped include:

- reads of identification registers such as CPU ID and interrupt controller ID
- cache maintenance operations
- TLB maintenance operations
- waiting for interrupt operations
- access to co-processor registers

- trapping exceptions.

The trap reason is presented to the hypervisor in a single register, with a second register used for more specific information. This allows the hypervisor to quickly identify the trap, and process it. Since these traps are configurable, we have the option of letting a guest own a specific device, or control the system in some ways. This is useful in, for example, an embedded device with a *real-time operating system* (RTOS) and a high-level OS; it may make sense to only allow the RTOS to suspend the processor while waiting for an interrupt, or allow the high level OS to access a co-processor that provides extra features.

2.8.5 New page-table format

The page-table format has been replaced with a new page-table format [ARM10c], used for the hypervisor's second-stage translations, and optionally available for first-stage translations for a standard operating system kernel running natively, or as a guest. This new format is disabled by default and not required for the first stage of translations. However, all second-stage translations and hypervisor translations must use the new page-table format.

The new page table is introduced to support physical addresses over 32 bits, and to add support for a second-stage translation. Each page-table entry is now 64 bits, allowing for larger input and output addresses. A *block* mapping is now 2MiB instead of 1MiB. Due to the larger entry size, the new page-table format is also less cache efficient.

2.9 Comparison of Hardware Extensions

It is useful to compare Intel's VT-x extensions to ARM's virtualization extensions:

- New mode: both Intel's VT-x extensions and ARM's virtualization extensions introduce a new mode of execution. However, there are important differences. Intel's VMX root mode is orthogonal to the CPU execution modes, and so a processor can be in VMX root mode and any of the user or privileged modes. ARM's new mode is not orthogonal, but is a higher privileged mode above the supervisor.

- Virtual Memory: VT-x with *extended page tables* (EPT) is quite similar to ARM's extensions, as both support a hardware page-table walk to convert guest physical addresses into physical addresses. However, VT-x uses a pre-existing page-table format, while ARM requires a new page-table format.

- Guest Identification: Both the VT-x and ARM's extensions support guest identifiers which are tagged in the TLB, thus avoiding TLB flushes on a world switch.

- Virtual Interrupts: Intel's VT extensions allow events to be injected into a running guest, while ARM's interrupt support is more thorough, adding a new virtual CPU interface view that is mapped in to the guest. This hardware handles all acknowledgement and clearing of interrupts, including priority masking, without any traps into the hypervisor.

- Emulation Support: ARM's emulation support allows for much faster trap-and-emulate; VT-x does not have any comparable offering. Due to x86's CISC nature, trap-and-emulate requires a full featured ISA emulator in the hypervisor.

- I/O: Intel's VT-d allows secure DMA into a guest address space. ARM has no comparable feature currently, which means that DMA cannot be used by a virtualized guest safely.

The above comparison shows us that ARM's extensions allow for a simpler hypervisor, due to the extra hardware support for interrupts and emulation in particular. The ARM architecture's simpler RISC design allows for emulation acceleration, and leads to a simpler hypervisor overall.

Unlike Intel's initial release of VT-x, the ARM extensions are fully featured even in the first release, and contain much support to ensure efficient virtualization. The inclusion of second-stage translations, VMID TLB tagging, virtual interrupt support, and emulation support are all beneficial features for low overhead virtualization.

Chapter 3

Approach

The primary goal of this thesis is to achieve pure virtualization of concurrent Linux guests using ARM's virtualization extensions in the OKL4 microkernel. For example, the configuration shown in Figure 3.1 should be possible, where each of the running guests is also capable of running natively. A secondary goal is to achieve communication between guests using calls to the hypervisor (hypercalls). The microkernel will be modified to run as a hypervisor in *hyp* mode, with support for communication added through the new *hypervisor call* instruction (HVC).

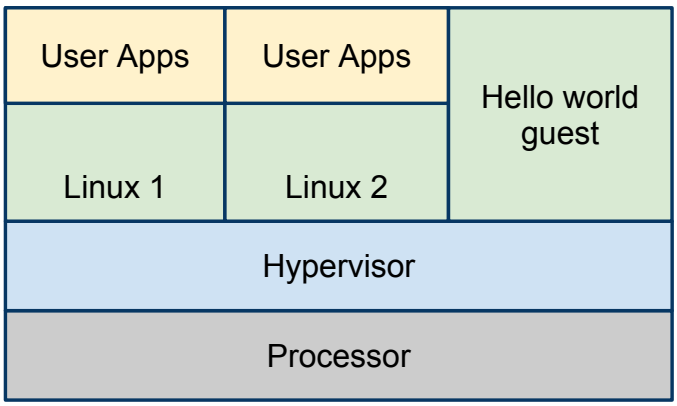


Figure 3.1: Concurrent unmodified guests

A basic overview of steps undertaken in this thesis:

- familiarise myself with the ARM architecture and the virtualization extensions;
- write a simple prototype hypervisor that can host a single guest (Linux);

- integrate the virtualization extensions into the OKL4 microkernel with multiple guest support;
- add support for communication between multiple guests;
- benchmark the resulting system running Linux against native and para-virtualized Linux.

3.1 ARM Familiarisation

While I had used some ARM assembler before, I was not familiar with the system level configuration such as virtual memory or the interrupt controller. To gain a better understanding of these components, I wrote some simple guests that run natively on hardware, which were also useful for testing the hypervisor functionality. These guests include:

- A hello world guest that tests console access and verifies memory by writing out different patterns, and verifying these patterns.
- An interrupt test that sets up interrupts from multiple sources such as the timer and RTC, which receives interrupts continuously.

These guests allowed me to gain an understanding of how to perform common operations such as switching between modes, access and modify the interrupt controller, and set up devices for which I would need shared drivers (timer and RTC). They were very useful for testing the hypervisor as they are much simpler than Linux.

3.2 Prototype

Although the ARM virtualization extensions are documented, I decided that a better understanding of the architecture would be gained if a simple prototype was written. The prototype was kept simple, with support for only a single guest, and pass-through support for all devices to the guest.

The prototype was also useful for an unexpected reason—finding bugs in the simulator. It was much easier to find, and ensure that problems found were caused by bugs in the simulator in a simple prototype than in a full implementation.

Once Linux booted in the prototype hypervisor, it was not developed any further. The knowledge gained was useful for making design decisions about the hypervisor which are discussed in the following section.

3.3 Design

With the experience of building a prototype hypervisor, I was able to consider design options for a more complete hypervisor capable of virtualizing multiple guests. We discuss a few options and tradeoffs before looking at the final design of the resulting hypervisor.

3.3.1 Dynamic guests

We have the option for statically deciding the number of guests at compile time, or allowing new guests to be started on-demand at runtime. While dynamically starting guests allows for more flexibility, it requires more complicated bookkeeping of virtual machine state and emulated device state.

Dynamic guests are useful in desktop and server markets, where the number of guests may change based on the load of the machine. Static guests are more common in embedded devices such as mobile phones, where the number of guests is set as design-time.

We chose static guest configuration, as this leads to a simpler hypervisor, while also modelling the more likely scenario for embedded devices, where ARM processors are most often used. The hypervisor design allows for any number of guests to run concurrently, although the number must be configured at compile time.

3.3.2 Virtual memory

The hypervisor can run with the virtual memory system enabled, or with a simple flat 1:1 mapping to physical memory. Virtual memory can be used to hide fragmentation, and relocate parts of memory easily. After implementing the prototype, I decided to run the hypervisor without virtual memory for the following reasons:

- virtual memory requires more code to set up a page table based on the new format;
- virtual memory requires page-table walks for translation of every hypervisor memory access;

- there is no benefit from hiding fragmentation or relocating pages, as the hypervisor only allocates pages;
- virtual memory has no effect on the security of the hypervisor.

The lack of virtual memory kept the implementation simple, which was another important factor due to the limited timeframe. Translations still occur for guests, as can be seen in Figure 3.2.

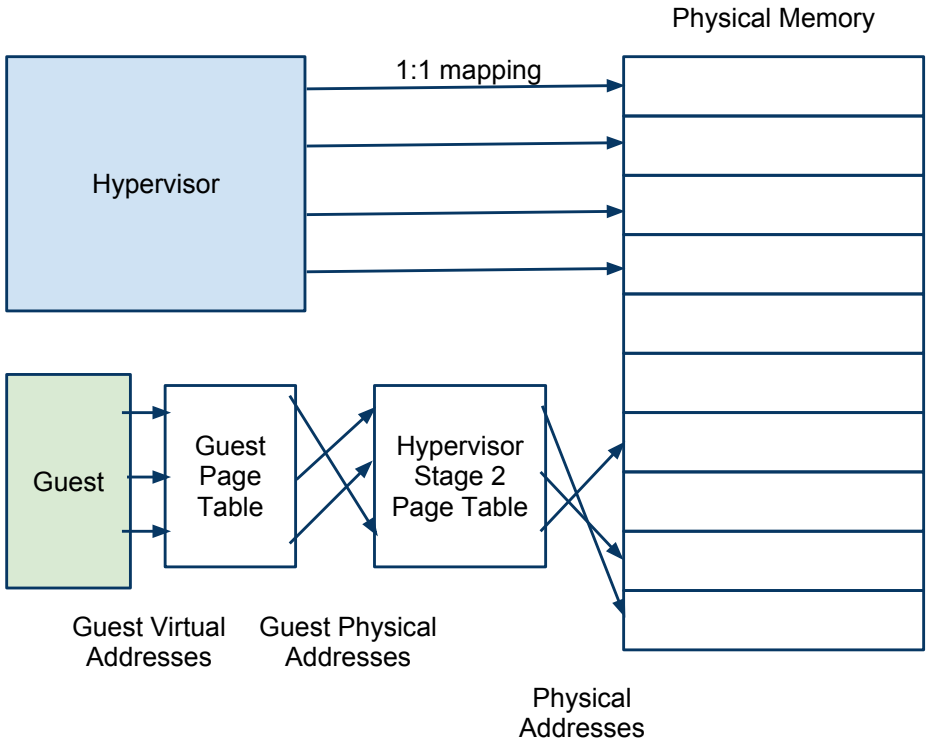


Figure 3.2: Virtual memory system

3.3.3 Separate hypervisor and virtual machine manager

Hypervisor functionality can be added to a microkernel in one of two ways:

- Embed all hypervisor functionality into the microkernel, and run the resulting hypervisor in the highest privilege mode. This improves performance but at the cost of increasing the trusted computing base.
- Separate hypervisor functionality into multiple components with most of the guest management performed by a user mode component [Bie06]. This approach loses

performance due to the required communication between the separate components, although it maintains the microkernel philosophy of keeping the TCB minimal.

As discussed in Section 2.5, Nova and Biemüller use a user-level monitor application, as there is a large amount of emulation required for x86 virtualization. The user level monitor in NOVA is required to perform:

- instruction emulation: a full x86 emulator is required, which is much more complicated than an emulator for a simpler architecture like the ARM architecture;
- device emulation;
- BIOS emulation.

An ARM hypervisor does not need any BIOS emulation, as there is no BIOS on ARM, and due to the instruction emulation support provided by hardware, only a very simple emulator is required. If a user-level monitor was written for ARM, we could only export the device emulation, and the instruction emulator (which currently is less than 50 LOC). Interrupt management, second-stage translation tables, and state saving would still need to be done in the hypervisor, so we decided to embed the hypervisor functionality completely in the hypervisor.

3.3.4 Multi-core

It is possible to increase performance of multiple guests on a multi-core machine by running separate guests on each core. However, this increases hypervisor complexity significantly. Due to my lack of familiarity with multi-core programming on ARM, and the short timeframe, it was decided to assume that only a single core is used.

In mobile phones, a single core running both the user applications and modem software is one of the goals of virtualization, as it would cut costs and power usage.

The design of the hypervisor would not change significantly for multi-core support. The hypervisor would still set up guests in a similar way although complexity of the hypervisor would increase due to the concurrency issues of running the hypervisor concurrently as another guest. Further work can look into modifying the hypervisor to allow multiple guests concurrently on separate cores.

3.3.5 Inter-VM communication

Running virtual machines require explicit communication with the hypervisor and other guests to achieve the following in an efficient manner:

- network between virtual machines;
- hosted drivers, where one guest hosts the driver, and other guests communicate with the hosted driver to interact with the device;
- share data between applications running on separate guests efficiently.

Communication between virtual machines is similar to inter-process communication and can be implemented via similar mechanisms:

- asynchronous messages using kernel buffers;
- synchronous messaging, with messages transferred directly between guests;
- buffer sharing, where a single guest's pages are mapped to another guest.

Efficient transfer of large amounts of data is best implemented with buffer sharing, and so we chose to implement buffer sharing. However, buffer sharing is excessive for small one-off messages. Hence, a simpler messaging system, asynchronous messages, was also implemented. Synchronous communication was not implemented, as it would cause guests to block waiting for other guests, which forces one guest to trust the other. We discuss the full design of inter-VM communication in Section 3.7.

3.3.6 Other limitations

Due to the timeframe for this thesis, there are a few limitations which will be discussed below. These limitations are due to a lack of time spent optimising the hypervisor. This is because the primary goal of the thesis was to achieve virtualization of multiple guests in the short timeframe rather than optimising the hypervisor.

Page sharing

The hypervisor currently does not share any pages between guests. Even when booting the same operating system twice, it requires pages to be duplicated. Optimisations can

be made to share pages, and use a copy-on-write mechanism to copy the page if it is modified. This was not of high priority as memory usage was not a big concern for this thesis. In embedded devices, page sharing is less relevant as it is unlikely that we will be running multiple copies of the same image.

No block allocations

All mappings use page mappings for simplicity, although the hypervisor supports creation of block mappings. Block mappings would help optimise runtime speed by only using a single TLB entry for a 2MiB range, while also reducing the number of page-table walks, as one less walk is required for a block entry.

3.4 Shared Devices

Since multiple guests expect to use the same hardware, the hardware must be shared between the different guests. This device sharing is implemented in the hypervisor, and the number of shared devices has been kept minimal, to avoid introducing a large amount of drivers in the hypervisor. The hypervisor emulates the following devices from the RealView platform:

- interrupt distributor component
- SP804 dual-timer (mostly multiplexed)
- PL031 PrimeCell real time clock.

Although each guest can access all available consoles, guest operating systems typically only use a single console. The RealView platform contains 4 consoles (UART0-UART3), and so consoles are assigned to guests by distributing the separate consoles, and remapping them so all guests see UART0.

Interaction between the guests and devices is shown in Figure 3.3.

3.5 Implementation in OKL4

The virtualization extensions have been integrated into the OKL4 microkernel while also adding support for multiple guests. The OKL4 microkernel is available in multiple

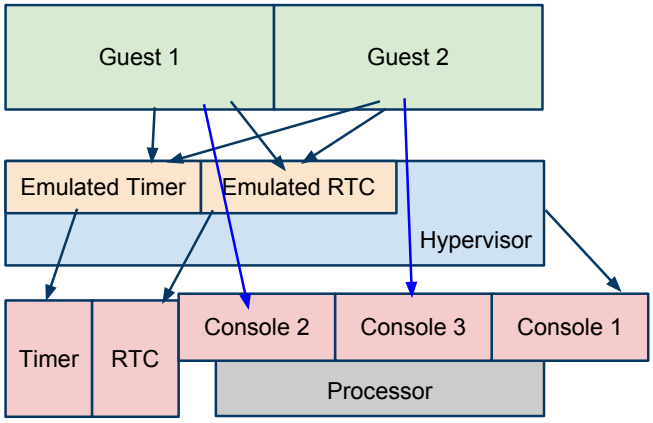


Figure 3.3: Interaction between guests and devices

configurations, although only two were considered for this integration: the microvisor, and the pico product. The microvisor is an embedded hypervisor supporting para-virtualization, while the pico product is a very minimal configuration with support for a static number of user applications configured at compile time, and no support for virtual memory.

The microvisor product contained more functionality than was required, as the API abstracted out hardware resources such as page tables and interrupt support, although these abstractions were not necessarily useful for pure virtualization. In pure virtualization, the hardware interface is the API, and so the inbuilt abstractions would need to be removed or modified significantly. Hence, it was decided to use the pico product, and build required functionality on top, rather than removing functionality from the microvisor.

Another advantage of using the pico product was due to it not using virtual memory. The OKL4 build system sets up all page tables statically, and so supporting a hypervisor built on the microvisor would require the static page table generator to be modified to support the new page table format introduced with the virtualization extensions. Since the pico product assumes a one-to-one mapping, no modification of the static build tools was necessary.

Here is an overview of what was done to embed the extensions in the microkernel:

- Add a new syscall that moves the kernel into *hyp* mode and sets up guests passed via arguments to the syscall.

- Implement guest page tables for each guest, which will allow a single hello world test to run.
- Add interrupt and timer support and generate interrupts to use for a *world switch*, which replaces the current running guest with a guest waiting to be run. This allows multiple hello world tests to run concurrently.
- Extend interrupt support to pass virtual interrupts to guests if if they are not intended for the hypervisor. This allows a single instance of the interrupt test to run. Running multiple instances fails as multiple guests use the same physical device, which causes both guests to function incorrectly.
- Virtualize any devices required by the interrupt tests and get two instances of the interrupt test running concurrently. The timer and the RTC device are both used in the interrupt test, and so both of these devices are required to be emulated.
- Add Linux support and get multiple Linux instances running.

Interrupt support for guests requires most of the virtual interrupt controller state to be saved. This also required allowing pending interrupts to be added to the saved state, as an interrupt destined for a guest that is not running has to be stored in the saved VCPU interface.

World switches are expected to be slow, as the current guest state is saved, the microkernel state restored, and once the microkernel finds a new guest, the microkernel state is saved and state for the new guest is loaded. For each guest, the hypervisor saves and restores the following state:

- all core registers, including banked versions of registers;
- all virtual memory system state;
- all interrupt controller state;
- any other control state stored in the co-processors, such as the FPU state.

Since this is a large amount of state, work could be done to only switch values that have changed, and keep track of unchanged values using the configurable traps. This method may reduce the world switch cost to around half the current cost. However, since

world switches are infrequent, this is not of a high priority and can be considered for future work.

3.6 World Switching

The implementation of multiple guests requires guests to be switched periodically. It is possible to world switch on:

- timer interrupt;
- trap caused by a *wait for interrupt* (WFI) or *wait for event* (WFE) instruction, as it signals that the guest is idle until an interrupt or event;
- interrupt directed at a guest that is not running.

It was decided to world switch only on a timer interrupt. This ensured guests were given a fair share of time, without any extra bookkeeping. This is an implementation restriction that can be modified easily without requiring changes to the overall design of the hypervisor.

3.7 Inter-VM Communication

Communication between guests requires a basic framework for hypercalls. Hypercalls use the new HVC instruction to allow guests to trap into the hypervisor. The hypervisor uses registers as store for arguments and results of hypercalls.

Initially, communication was supported by a simple hypercall to pass a word to another guest, with notifications of new messages using interrupts. Messages in transit are buffered by the hypervisor, and may fail if the hypervisor buffer is full. There is a limit to the amount of data that can be transferred in each hypercall through registers, while also being limited by the buffer space in the hypervisor. An alternative page sharing method was also implemented to allow sharing of large amounts of data.

The following hypercalls are added to support simple message passing, and page sharing:

- get VM ID, return an ID that uniquely identifies the running guest;

- send message, to a specific guest by ID, or to all guests;
- get message, retrieves the last message, and the sender;
- share page, assigns a share ID that can be mapped in to other guests;
- map share, maps a share that has been previously shared by another guest.

One of the main motivations for communication between guests is to allow a single device to be shared between multiple guests. For example, the OKL4 microvisor is capable of running virtual guests and device drivers, and the para-virtualized guests communicate to devices through the hosted device drivers. Since I will be running on a simulator with no access to real devices, I decided to create a simpler version of the scenario where:

- The OKL4 microvisor runs a simple application that hosts a virtual console device.
- A Linux guest is modified to include a console driver that communicates with the virtual console application using the hypercalls.

This is much simpler than a real world hosted driver, although the basic communication remains the same. This proof-of-concept driver can be used as a base for more full-featured hosted drivers in future work.

3.8 Benchmarking

Benchmarking will initially be done with both the para-virtualized microkernel and the pure virtualized microkernel running a single instance of Linux. The results of running benchmarking suites such as *LMbench* on native Linux can then be directly compared with running the benchmarks on the virtualized Linux instances.

The hypervisor performance will be measured by benchmarking hypervisor functionality such as data fault traps, world switching, interrupt processing and hypercall entry.

Since no hardware is available, tests will only be performed in ARM's Fast Models simulator, which does not model memory access timing or cache timings correctly. However, the resulting instruction counts can still be used to get a rough idea of performance. The simulator is capable of logging memory traces which can be analysed

to measure the cache impact. However, performance of the simulator drops significantly when tracing is enabled, so benchmarking using memory tracing is not feasible.

Chapter 4

Implementation

I split the implementation into separate stages to understand the architecture and extensions:

- ARM familiarisation
- simple prototype hypervisor
- integration of extensions in the OKL4 microkernel
- hypercall support.

The implementation supports concurrent unmodified guests such as Linux, and the OKL4 microvisor, while also supporting hypercalls for message passing and shared memory.

4.1 ARM Familiarisation and OKL4

Understanding of the ARM virtualization extensions first required an understanding of the ARM architecture. No hardware with the virtualization extensions has been released, so all work was performed on ARM's Fast Models simulator, which models the *RealView emulation baseboard* configuration.

To gain familiarity with the ARM architecture, I first implemented some simple applications that allowed me to get a better understanding of the architecture including the system configuration of the interrupt controller and the RealView platform.

Since the goal was to implement the extensions in a microkernel, I also investigated the OKL4 microkernel codebase to investigate differences in available product configurations, and find the right configuration to use as a base for building a hypervisor.

4.1.1 Simple ARM applications

I implemented a simple *hello world* application running on the hardware with no operating system. There were no issues as I did not need to configure the VM system or the interrupt controller; I only had to set up the console device. The test application also did some simple memory tests by storing and verifying patterns.

The second test application, an interrupt guest, took longer to implement due to issues with documentation. The documentation I was using for the RealView baseboard specified the memory map for an A8 core, which consists of a *global interrupt controller (GIC)* for interrupt configuration. However, the older version of the documentation failed to mention that the A9 core uses a separate interrupt controller called the *distributed interrupt controller*. Although configuration of both components is similar, the base address for the components is different, and so the distributed interrupt controller was not enabled. This led interrupts generated by devices not to get forwarded to the CPU. Once the base address was updated to use the right interrupt controller, interrupts were forwarded to the CPU and the interrupt test functioned correctly.

The completed applications were useful for testing hypervisor functionality as they required:

- second-stage translations to map guest physical addresses to actual physical addresses
- mapping of devices such as the console and timer to allow guests to use the devices
- interrupt routing support to allow guests to receive physical interrupts.

4.1.2 OKL4 investigation

Initial OKL4 investigation began with the microvisor product—a microkernel based product developed for para-virtualization. I started modifying the OKL4 microvisor to run in *hyp* mode; however, the microvisor uses virtual memory, and the virtualization

extensions require the hypervisor to use the new page table format. This new format was not supported by the build system (which built and linked a page table statically), and so an alternative approach was required.

Rather than modifying the OKL4 build system to build new page tables, I investigated an alternative product, the OKL4 pico product. The basic differences between the pico product and the microvisor product are:

- the microvisor product is aimed at virtualization as it contains hypervisor functionality, while the pico product is a microkernel for running small applications in unprotected mode;
- the microvisor API uses abstractions required for virtualization such as vMMUs, vIRQs, and vCPUs, and allows dynamic configuration of applications. Pico does not allow any dynamic configuration, and is set up statically at compile time;
- the pico product has a footprint of under 4KiB, while the microvisor memory footprint is at least 10 times the size, due to the additional functionality;
- the microvisor product requires virtual memory to run, while pico does not use virtual memory, as it was developed for hardware without a MMU.

Since the OKL4 pico product does not use virtual memory, it was easier to modify to run in *hyp* mode. However, the pico product had not yet been ported to run on the RealView platform. I was faced with two options at this stage:

- modify the OKL4 build system to generate new page tables and get the microvisor to run
- modify the pico product to run on the RealView platform.

As discussed in Section 3.5, I decided to use the pico product, as the virtualization extensions did not require the abstractions presented by the microvisor. I was unable to port the pico product to the RealView platform due to my lack of familiarity with the OKL4 codebase, and lack of ARM knowledge. Hence, I decided to focus on building a simple prototype from scratch, and the OK-Labs engineering team ported the pico product to the RealView target.

4.2 Prototype

To gain a better understanding of the ARM extensions, I built a prototype to host a single guest. The aim of the prototype was to host a single Linux instance which would have control of all devices and interrupts. This simple hypervisor was helpful in understanding what design to use for the final integration of the virtualization extensions in the microkernel, as discussed in Section 3.3.

The prototype hypervisor was broken down into the following milestones:

- set up *hyp* mode
- load up guests
- set up page tables for second-stage translations
- set up the interrupt controller and insert virtual interrupts for all physical interrupts
- test Linux.

These steps are described in more detail in the following sections.

4.2.1 Initial bring-up

I wrote a simple prototype hypervisor that ran without virtual memory to keep my implementation simple. The processor starts in secure supervisor mode. However, *hyp* mode can only be entered from the non-secure world, and so we must first disable the secure monitor before entering *hyp* mode. I implemented the following steps to setup the processor in *hyp* mode:

- enable hypervisor mode by modifying the Secure Configuration Register (SCR);
- set up the base vector address register for the secure monitor (MVBAR);
- enter the secure monitor mode;
- in the monitor, enter non-secure mode by modifying the SCR;
- set up the base vector address register for the hypervisor (HVBAR);
- leave the monitor, entering non-secure supervisor mode;

- enter hypervisor mode by invoking a HVC instruction.

I faced some issues with the build system, as I did not have an assembler capable of assembling any new instructions (such as HVC), or accesses to new registers. I dealt with this issue by manually encoding all the new instructions, then using *.word* directives to represent the new instructions in the assembler code.

Once my prototype booted into *hyp* mode without any issues, I worked on embedding the guest binary into my prototype so I could load the guest directly from RAM, rather than storing the guest in flash memory and writing a flash driver in my prototype. I achieved this by wrapping binary data into an object file, and linking it in. The binary data could then be loaded by simply referencing the linked symbol. Guest binaries use the *executable and linkable format* (ELF), a commonly used generic executable format for binary images. I used a simple ELF loader provided by the ERTOS group at NICTA [NIC10] to parse the guest ELF binary.

4.2.2 Second-stage translations

To run the hello world guest on top of the hypervisor, I loaded sections from the ELF file and set up the second-stage translations to map guest IPAs to the physical location of the ELF segments. This also required a frame table mechanism to allocate frames for both the guest and for the page tables set up by the hypervisor.

I wrote a simple frame table that does not support deallocation of frames; it simply returns increasing addresses for the next free frame.

Once the frame table was ready, I implemented support for the second-stage translations page table. I initially wrote a simple block mapping, which only requires the first level page table to be set up, and maps a 2MiB region. The second-stage translation configuration was written to the new registers, VTCR and VTTBR, and enabled by writing to the HCE register.

I also added device mappings to allow the guest to communicate with the UART device. The hypervisor output was sent to UART1, so guests could use UART0 without redirecting page mappings. Once the page table was set up, the guest was started by modifying the return address, stored in the new *ELR_hyp* register, to the entry point of the ELF, and then returning from the hypervisor using the new ERET instruction.

This allowed the simple *hello world* guest to run with the second-stage translations, and the hypervisor allocating physical frames dynamically as required by the guest.

4.2.3 Interrupt support

The *hello world* test only required second-stage translations, but the interrupt test required interrupts to function correctly. This required emulating an interrupt distributor, while also supporting extensions added to the interrupt controller.

I did not initially realise that the interrupt distributor component would have to be modelled using trap-and-emulate techniques, assuming that the extensions had modelled them in hardware, as they had added a virtual CPU interface to the GIC. Fortunately, the new load/store emulation support allowed trap-and-emulate to be implemented efficiently and quickly, and I only had to focus on the functionality of the interrupt distributor.

The interrupt distributor is used to configure, enable and set priority of interrupts. The emulated interrupt distributor does not need to interact with the hardware, as it stores which interrupts are enabled, and virtual interrupts are only added to the VCPU interface if the guest has enabled the interrupt in the emulated distributor. This keeps the emulated distributor component simple, as it only stores interrupt configuration for each guest.

Once the distributor component of the GIC was modelled, I added support for the new VCPU interface of the GIC controller, and simply mapped the VCPU interface into the guest where the standard CPU interface of the GIC controller is expected. I then added virtual interrupts to the VCPU interface for physical interrupts taken by the hypervisor, and this allowed the interrupt test to run correctly.

However, there was an issue which caused virtual interrupts to never be marked clear by guests. Although the guest cleared interrupts, the virtual interrupts were never cleared in the VCPU interface, and so the hypervisor was unaware that the guest had completed processing the interrupt. This was a bug in the ARM simulator, and I was forced to work around the issue by clearing interrupts in the hypervisor after a certain period of time. This allowed the interrupt test guest to run without any issues.

ARM was notified of the issue, and a fix was provided at a later date.

4.2.4 Linux support

Bootloader support

Once my interrupt test was functioning, I focused on my next milestone, Linux. I initially aimed to create a single ELF capable of starting up the Linux kernel, but I was not able to get any existing tools to create a single runnable Linux ELF. Existing tools were only compatible with x86, or they did not function as expected. This led me to use *U-Boot*, a flexible boot loader, as the first guest which then loaded up Linux from another part of the hypervisor binary.

Initially, there were issues due to U-Boot accessing memory using the dynamic memory mirror on the RealView platform. The dynamic memory mirror allows RAM to be accessed at 0x0 or at a mirror address, 0x70000000. This caused issues as two separate physical frames were allocated by the hypervisor for the same guest physical frame, due to the different guest physical addresses used. I modified the hypervisor page table to keep track of dynamic mirror RAM entries by checking for entries at the standard address first. This resolved the issue, and allowed the Linux kernel to be loaded into RAM.

Kernel support

The Linux kernel did not start initially, due to the use of the *local timer* for scheduling, which was removed with the introduction of the virtualization extensions. The local timer was removed in favour of completely new timers introduced with the virtualization extensions. Emulation of the local timer could not be done efficiently, as it was located on the same page as the interrupt controller, which is accessed frequently for acknowledging and clearing interrupts. Emulation would require trapping all accesses to the interrupt controller as well as the local timer, which would incur a large performance overhead. I chose the simpler option which was to remove the local timer support in the Linux kernel by disabling the *CONFIG_LOCALTIMERS* option. This is a platform issue, as the virtual interrupt controller was added to the RealView platform only for testing the extensions. A new platform is expected to avoid this issue.

The Linux kernel now mostly started up, although the simulator crashed when attempting to mount the root filesystem. I debugged the issue and found that the simulator crashed when accessing the SMC flash chip, where the root filesystem was

stored. This seemed like a simulator bug, so I notified ARM, and was unable to progress till a fix was released.

ARM was aware of the issue, and so a fix was issued soon, allowing the Linux kernel to complete mounting the root filesystem, and start the *init* process, which is the first user-space process started by Linux.

There were other small issues, such as accesses to co-processors causing exceptions, which were resolved by configuring access control options while in secure mode. I added some initialisation code to the secure monitor which allowed the non-secure world to access all co-processors and modify all interrupt controller options.

User-space support

The *init* process did not start successfully, and instead caused segmentation faults which could not be pinpointed because the *PC* and *LR* registers were trashed. I debugged the issue by trying different user applications, different filesystems formats, different builds of busybox, and disabling caches, but was unable to pinpoint the fault. The segmentation fault always occurred with the same trashed *PC* and *LR* registers in all these cases, which led me to believe it was a simulator bug. After debugging for a couple of weeks, the issue suddenly disappeared. I then found out a new simulator update had been installed, which may have been the cause of the fix. However, even by running the old simulator with the old parameters, I was unable to reproduce the original issue. Due to time restrictions, I decided to move on, and debug the issue if it ever came back. The issue was not faced again, and I believe that a corrupted simulator install may have caused the issue.

Once the issue disappeared, Linux started up successfully, and was completely usable, while running virtualized under my simple prototype hypervisor.

4.3 OKL4 Pico Port

OK-Labs ported the pico product to run on the RealView platform during the period I spent working on the prototype. I was able to start integrating the virtualization extensions into the microkernel. The final hypervisor is capable of running multiple guests concurrently, with support for routing interrupts between these guests, and emulating some shared devices. The basic overview of what was done:

- set up the microkernel to run in *hyp* mode
- add support for second-stage translations
- switch between guests on a timer interrupt
- support interrupt storing/routing for guests
- emulate simple devices which are shared
- pass-through some devices to a single guest
- run two Linux guests.

A more detailed overview of each step, and issues faced during the steps is described in the following sections.

4.3.1 Prototype feature integration

I started integrating features from the prototype into the pico port one at a time. First, I added a new syscall which moves the kernel from supervisor mode to hypervisor mode. Once the microkernel is in hypervisor mode, the standard schedule is completely disabled, which causes (native) microkernel applications to stop running. This issue could be resolved by splitting the hypervisor into two parts: a hypervisor core for scheduling guests, and the microkernel still running in supervisor mode. However, this was not of high priority, and was not undertaken due to the limited timeframe of this thesis.

I then integrated support for the ARM extensions from the prototype in the microkernel, with a focus on multiple guest support during the integration. Guest state is stored in a static array, and is saved to and restored from on a world switch. To schedule between multiple guests, I implemented support for the new timers introduced by the virtualization extensions. Support for switching of second-stage translations and world switches allowed two *hello world* guests to run concurrently without any issues.

Interrupt support required some more work, as I had to context switch the interrupt controller between the guests. I save and restores all the registers in the VCPU interface. Switching the emulated distributor requires updating a pointer to point to the DIC state for the current guest.

4.3.2 Multiple Linux guests

Once my prototype was able to start a single Linux guest, and another simple application at the same time, my focus moved to getting multiple Linux guests running at the same time.

World switch

Switching between guests requires saving all guest state such as the virtual-memory system configuration, interrupt controller state, and any emulated devices state. The world switch is performed on an interrupt from the new timer introduced with the virtualization extensions. Since this timer is new, it is not used by Linux or any current guest operating system, and so it belongs solely to the hypervisor.

The largest amount of state switched on a world switch is the virtual-memory system configuration. Different guests may use different parameters for the virtual memory system, and all state (such as fault address registers and fault type registers) must be saved. This requires saving and restoring all of the state in co-processor 15 (CP15), which ARM uses to configure the virtual memory system. This state consists of approximately 25 registers, which must all be stored and restored on each world switch.

World switches require saving and restoring of the following state:

- user-mode registers and banked registers for all privileged modes
- virtual memory configuration (CP15 registers)
- all interrupt controller state
- FPU registers
- timer registers
- second-stage page table register
- virtual machine ID register.

In comparison, a standard context-switch only saves and restores the core registers, the process ID, possibly the FPU state, and the page table base registers. The extra state saved requires accessing co-processors and memory mapped devices, which is much slower

than accessing registers. For this reason, world switches are estimated to be 4-5 times slower than a simple context-switch.

The frequency of a world switch is currently set to 10Hz to ensure interactivity of guests without a large overhead. This value is completely customisable, but the higher the value, the larger the overhead of the hypervisor as more time is spent switching rather than executing guest code.

Device support

I first created a minimal Linux configuration so that I would not have to emulate many devices. I came up with a simple configuration that only required the serial, timer, interrupt controller and RTC devices to function.

The RealView baseboard is equipped with four console devices, and so consoles were distributed between the hypervisor and guests. However, guests expect to use UART0 and so routing of devices is used to map the specific UART device in the memory location of UART0. Routing of the UART devices is implemented by adding mappings in the hypervisor page table to map the UART0 page to another UART frame, and the result is shown in Figure 4.1. However, each UART device produces a different interrupt, and so interrupts also had to be routed so that a UART1 interrupt was added as a virtual interrupt with the ID of UART0 to the guest. This is displayed in Figure 4.2.

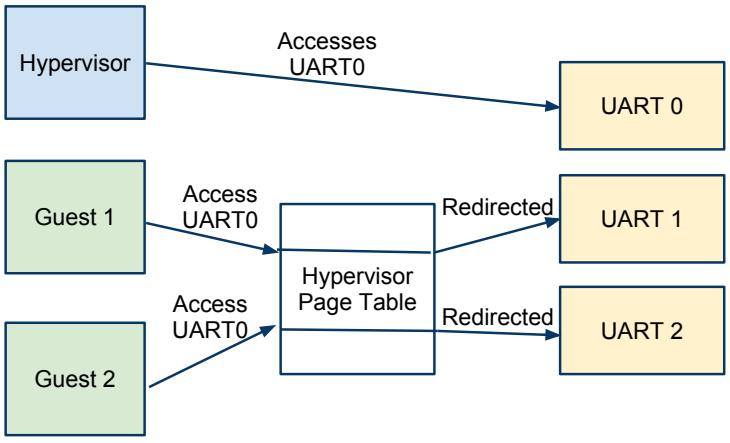


Figure 4.1: Redirection of UART0 for guests

Timers are completely switched by saving and restoring the whole state on a world switch, which means that guests see virtual time rather than real time. This was done for simplicity, rather than creating a virtual device that correctly generated interrupts for all

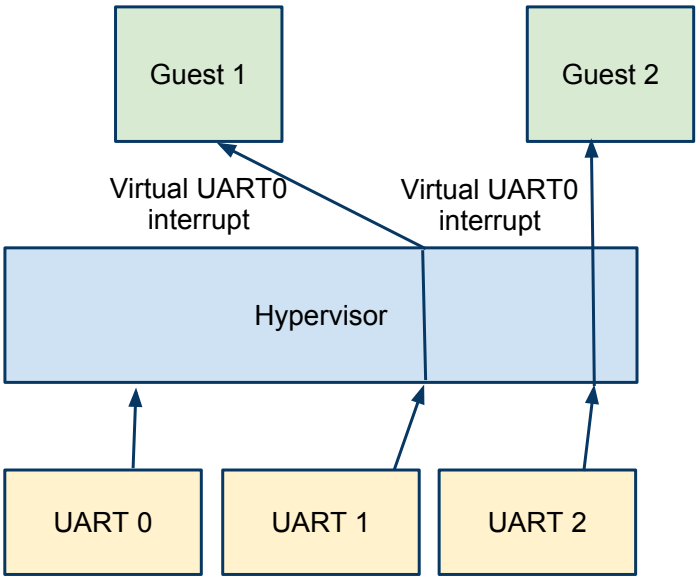


Figure 4.2: Virtual interrupts generated from UART interrupts

guests. It also meant that calculations, such as the start up BogoMIPS calculation loops in Linux, are correct even if there is a world switch during the calculation.

The interrupt controller is also world switched, and used most of the emulation from the prototype. I made sure that interrupt state for each interrupt controller was saved and restored correctly on a world switch, while also maintaining state for the hypervisor separately, as the hypervisor also received interrupts for a world schedule.

The RTC was emulated by a simple wrapper device that uses the original RTC to produce ticks every second, and sends virtual interrupts to all devices that have enabled the RTC device.

All devices were emulated using standard trap-and-emulate techniques, using a simple framework I added that allows emulated devices to register a memory region, and implement a simple trap handler of the type:

```
word_t trap_handler (word_t va, bool isRead, word_t regVal,
    word_t d_size);
```

Emulated devices use the given information (address of the fault, type of fault, and the current register value) to either perform a write on the emulated device, or return a value to be read by the access. The framework then updates any guest state, and resumes the guest from the following instruction.

Simple ARM emulator

Most trapped instructions have additional information such as the type of faulting instruction (read or write), the size of the read or write, the register involved, etc. from the load/store emulation support in the virtualization extensions. However, some instructions are not supported by the emulation extensions, and so must be decoded manually. The Linux RTC driver uses instructions which were not supported by the emulation framework, such as:

```
ldr rd, [rs], #imm
```

This instruction loads from the memory address stored in *rs*, and stores the value in the register *rd*. However, it also updates the source register, *rs* by adding the *#imm* value to it. This update of the source register is known as write-back, and is not supported by the emulation support. Since decoded information about the instruction is unavailable, the instruction has to be fetched explicitly from guest memory, and decoded.

I implemented a very simple ARM emulator that supports instructions that cause write-back, and this emulator was enough for emulation of devices used by a minimal Linux configuration. My ARM emulator did not need to support any other types of instructions, and so was extremely small at less than 50 LOC.

Interrupt issues

After virtual memory switching and emulated devices were completed, I was able to test multiple Linux guests. However, the boot process did not complete when running with multiple guests, as only the initial few interrupts were firing. I debugged the issue and found that the priority drop mechanism (added with the virtualization extensions) was not functioning.

As discussed in Section 2.8.2, the priority drop feature allows the hypervisor to drop the running priority of the system when an interrupt is received, while keeping the interrupt marked as active. This is used to ensure that guests cannot mask other guests' interrupts. However, this functionality was not working correctly, and was noticed as it blocked the hypervisor interrupts in the following case:

1. hypervisor schedules guest 1 to run;

2. an interrupt is received by the hypervisor for guest 2. Hypervisor activates priority drop so that further interrupts are not blocked;

3. the world switch interrupt is generated;

4. however, this interrupt is not received by the hypervisor, as the running priority still indicates the priority of the previous interrupt. This in turn blocks the system from making any progress, since no further interrupts are received.

I contacted ARM about the bug, but ARM was unable to fix the issue before the completion of this thesis, although they did acknowledge that there was a bug with the priority drop mechanism and it would be fixed in the next release.

I worked around the bug by modifying the priority of all interrupts to be lower than the world switch interrupt. This guarantees that the world switch interrupt is never lost. However, incorrect processing of an interrupt still allows a guest to stop other guests from making progress. If a guest does not mark an interrupt as complete, it will be running forever, and no other interrupts for other guests will be processed as the priority of all guest interrupts are the same.

This did not cause any serious problems for my thesis, and is only a problem when running in the simulator. The issue would not occur on hardware as it will be fixed before a hardware release, so this workaround would not be necessary.

4.3.3 Device pass-through

Once multiple Linux guests were supported, I worked on getting pass-through device support for a single guest. This allows some devices to be exclusively used by a single guest, without any virtualization overhead; other devices can still be shared. I enabled pass-through for the following devices to the first Linux guest:

- SMC flash chip
- RealView CLCD controller (simple LCD)
- network controller
- audio controller.

Adding support for these devices required mapping the memory regions for the devices directly to the guest, and ensuring all interrupts related to the devices are routed to the guest.

The CLCD controller required slightly more work to set up as the buffer pointer is set up by the guest, but the address is sets up is an IPA. To get around this, I added a trap-and-emulate layer which trapped accesses to the CLCD buffer pointer, translated the address to the actual PA, and stored the PA in the buffer pointer. This also required ensuring that the VGA buffer was stored contiguously in PA range, as the Linux guest can only ensure that the IPA is continuous. This allows passthrough of the LCD controller to function correctly, as can be seen in Figure 4.3.

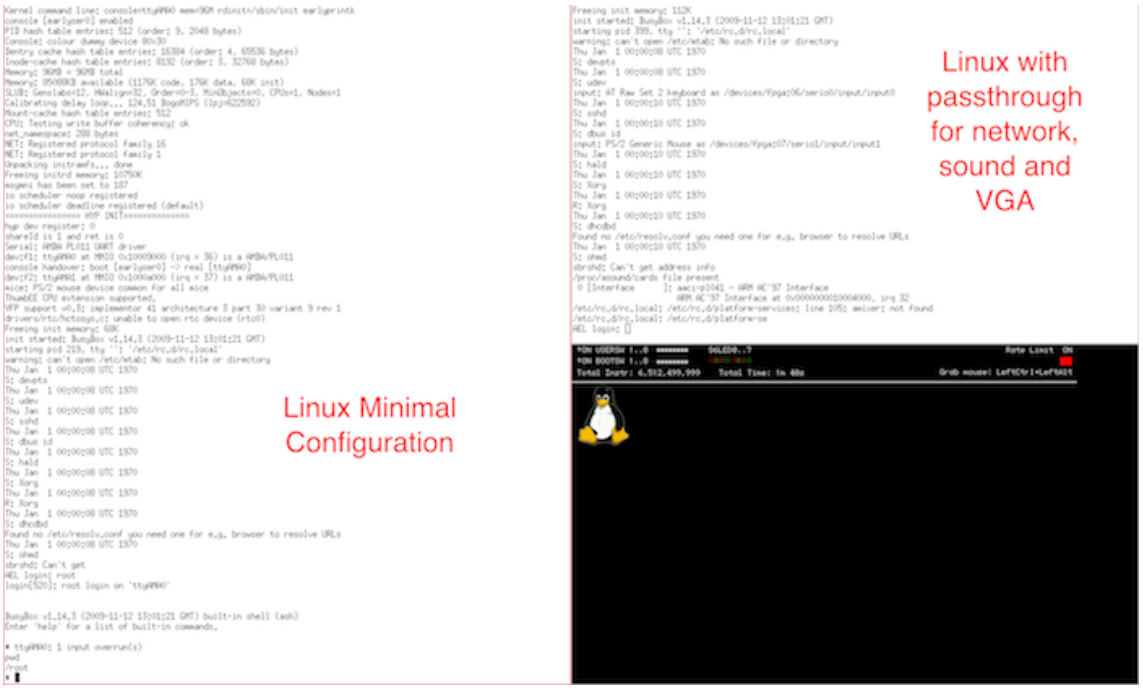


Figure 4.3: Linux with VGA passthrough

4.4 Hypercalls

I added hypercalls to allow for inter-VM communication between the running guests. Hypercalls were implemented using the new HVC instruction which allows a guest to invoke the hypervisor. I wrote a simple framework in the hypervisor to handle hypercalls, where arguments and the result are passed through registers.

4.4.1 Simple communication

I initially started with only the simpler hypercalls described in Section 3.7 required for basic communication. The following hypercalls were added:

- get VM ID
- send message
- get message.

This simple API was used to investigate the implementation and consider alternate approaches. The *hello world* test application was modified to test the functionality and ensure that messages could be sent between guests. I used polling to to check for new messages, as interrupt support was not yet written.

4.4.2 IRQ notifications

To avoid polling, I added interrupt support to hypercalls. Once a message was delivered to a guest, a virtual interrupt was set up by the hypervisor for the guest. I then modified the test guests to wait for interrupts rather than polling. This simple communication let me test that the IRQ notifications were functional.

4.4.3 Page sharing

I added support for sharing of pages between multiple guests for a more efficient method to share data between multiple guests, as described in Section 3.7. One guest sets up a page, and invokes a page share hypercall, which returns a share ID. This share ID can be used by other guests to map the same physical page into their address space.

To test the functionality, I modified my simple guests to implement a producer-consumer solution. One guest produced data, and stored the data on the shared page, while the other guest consumed the data. This test ensured that the shared page was functional even with multiple guests accessing data at the same time, and to ensure that simple load exclusives and store exclusives were enough to ensure safety with concurrent accesses.

4.5 Real World Inter-VM Communication

Although the communication hypercalls were functional, I had only tested them using very simple applications. To ensure that the hypercalls were flexible enough for more complex applications, a more real-world example was implemented which closely simulates a hosted driver scenario.

I implemented a shared device through a hosted driver running in a separate guest. Hypercalls were used for communication between Linux and the hosted driver. I used the OK-Labs microvisor product to run the hosted driver, as there are many existing drivers hosted in the microvisor such as drivers for frame buffers, timers, and flash devices. Due to time constraints, I kept the API simple by only implementing a simple console device.

4.5.1 Microvisor

To start writing my console host driver, I needed to run the microvisor on top of the hypervisor. The stages required to process a virtual console write and the configuration for the hosted driver is shown in Figure 4.4. Communication is broken into:

1. Linux sending a message using a hypercall;
2. the hypervisor notifying the microvisor application of the message using an interrupt;
3. the microvisor receiving the interrupt and sending an IPC to the user console application.

There were some issues that had to be resolved before the microvisor could be run. The first issue was the lack of support for Thumb and Thumb-2 instructions in my hypervisor. When an instruction traps, the hypervisor emulates the instruction and resumes execution from the next instruction. This requires knowledge of how large the instruction is. I initially assumed that all instructions were 32-bit, but Thumb instructions are only 16-bit. I modified my hypervisor to skip instructions based on the current execution state of the guest. However, it still failed to run as I had not taken the Thumb-2 instruction set into account. I had not realised that Thumb-2 instructions could be of different length (2 bytes or 4 bytes), and was unsure of how to detect the size of the current instruction. I

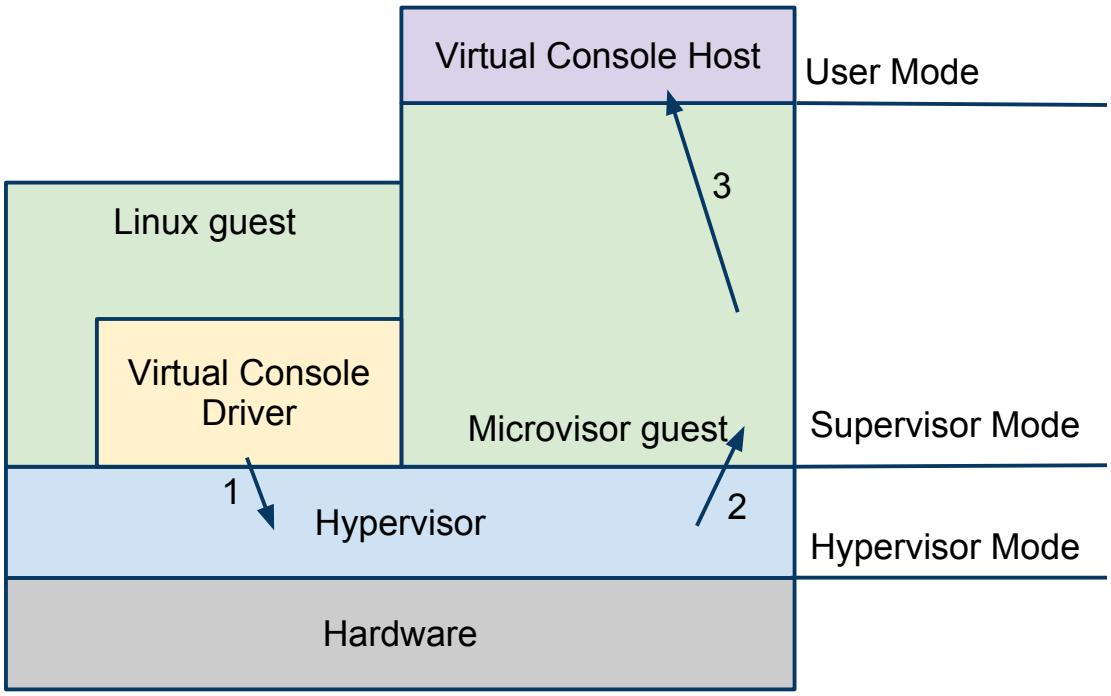


Figure 4.4: Components for inter-VM communication

then found that the ARM emulation support provided information about the size of the current trapped instruction.

Once Thumb support had been implemented, the microvisor still failed to start due to some TLB locking code. Since the TLB is shared, guests are no longer allowed to lock TLB entries, and instead the instructions faulted into the hypervisor. I decided that TLB locking was not of high priority for this thesis, and decided to comment out the offending code.

Unlike Linux, the microvisor uses two console devices—UART0 for the microkernel prints, and UART1 for the user application. I added a device mapping (to the mapping shown in Figure 4.1) such that the microvisor's UART1 was mapped to UART3. This allowed both consoles to be used by the microvisor.

Once the microvisor booted, I wrote a simple application that mapped in a shared page from Linux, and read data from the page when a hypercall message was received (via an interrupt). The application simply waited for interrupts, and when an interrupt was received, it read the characters from the shared page, and printed them to the physical console.

4.5.2 Linux driver

I created a Linux driver which creates a */dev/hypcom* device, implemented as a *char* device in Linux, which writes data to a shared page, and sends a message via hypercalls to the microvisor after each write call. The microvisor waits for these interrupts, then prints all data found on the shared page. Locking of data is performed in the same way as my producer-consumer example written earlier, using load-exclusive and store-exclusive instructions.

The Linux module shows that although the Linux kernel was unmodified, a driver can be inserted to add hypercall support (using the dynamic loading functionality in the Linux kernel, *insmod*). This para-virtualized feature support doesn't require any changes to the kernel.

4.5.3 Virtual console set up

The final set up is:

- the virtual console application (running on the microvisor) starts up, and waits for interrupts from the hypervisor;
- Linux starts, and chooses a physical page, and initialises the contents of the page;
- Linux uses the page share hypercall to create a share ID for the page;
- Linux sends a message to all running guests with the share ID;
- the virtual console application receives the share ID, and maps it in;
- the virtual console application waits for messages, and on each message, processes any new data from the frame;
- when data is written to the virtual console device in Linux, it writes data to the shared page, and then sends a message to the microvisor

The resulting communication between virtual machines can be seen in Figure 4.5. The Linux instance first prints a file to its normal console, then prints the same file four times to the */dev/hypcom* device. This is communicated to the microvisor, and is printed by the microvisor to its console.

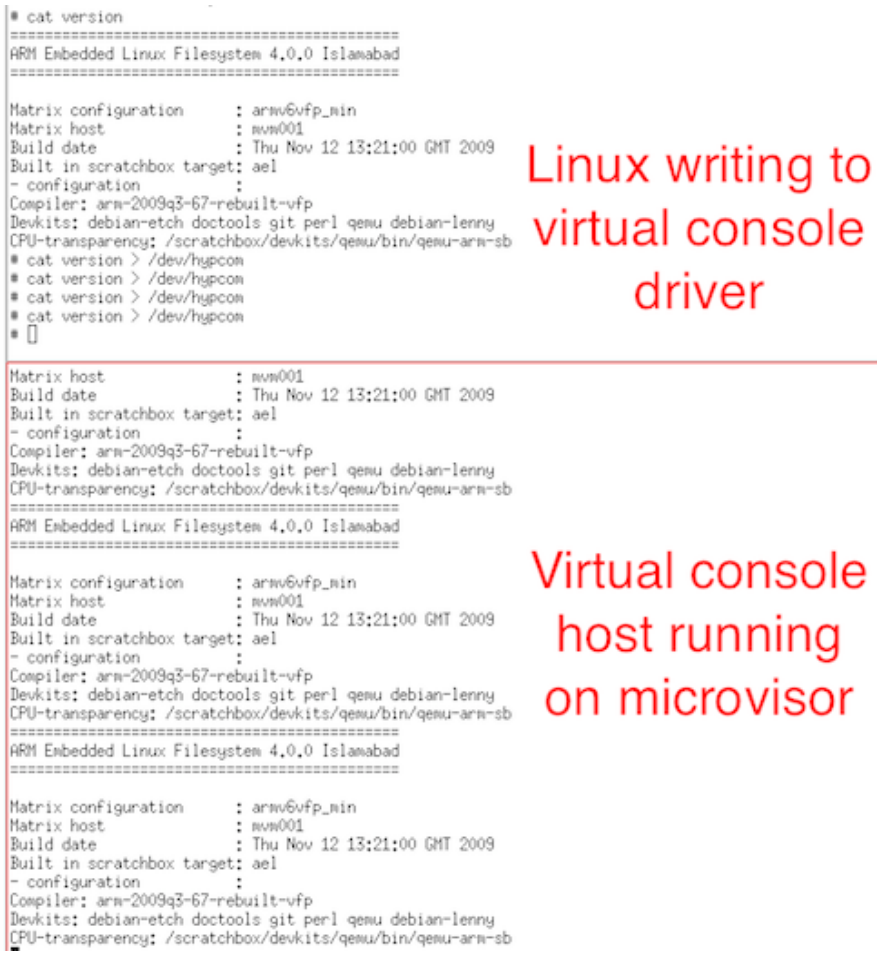


Figure 4.5: Virtual console driver in action

4.6 Summary

The implementation achieved all the core goals initially set. The hypervisor is capable of full virtualization of concurrent unmodified guests. Tested guests include:

- simple guests such as the hello world guests or interrupt tests
- Linux guests
- OKL4 microvisor
- simulator provided examples, e.g. Mandelbrot.

Communication between guests was also achieved, with full support for notifications using interrupts, and efficient communication for large amounts of data using page-sharing.

The final implementation supports some simple shared devices: consoles, timer, RTC and interrupt controller. It also includes passthrough support for devices to be controlled

by a single guest. DMA between devices and guests is not supported due to the non-physical nature of guest addresses. However, it is possible to write a simple trap handler in the hypervisor that modifies any guest physical addresses sent to devices to real physical addresses, as described for the CLCD controller in Section 4.3.3.

Due to time constraints, some non-core functionality was not achieved:

- continue running microkernel applications as well as guests scheduled by the hypervisor;
- emulate a shared driver for a more complex device such as VGA, or network controller;
- support for multiple cores.

These goals were not of a high priority, and can be explored further in future work.

Chapter 5

Results

We first show multiple guests running virtualized on a single simulator, and show inter-VM communication between guests. We then consider the performance impact of using the ARM virtualization extensions against running Linux natively and running Linux para-virtualized. We also consider the overheads of common hypervisor operations such as world switching and message passing. Since the extensions are new, hardware implementing the extensions is unavailable currently, so all benchmarks are run on a simulator.

5.1 Virtualization of Multiple Guests

The hypervisor is capable of running any two guests concurrently, and runs all guests without any issues.

Figure 5.1 shows two of the simple ARM applications running concurrently. The first application is the interrupt test, which receives interrupts from the timer and RTC devices. The second application is the hello world guest, with modifications to test memory accesses through the dynamic mirror address, ensuring that both accesses result in the same value.

Figure 5.2 shows multiple Linux instances running concurrently, with slightly different configurations. One guest is configured to only use the minimal devices, while the other guest is configured to access all devices. The hypervisor uses pass-through to allow a single instance of Linux to access devices such as the network controller and audio controller directly. The CLCD controller can be used by the guest with device pass-through, as shown in Figure 4.3.

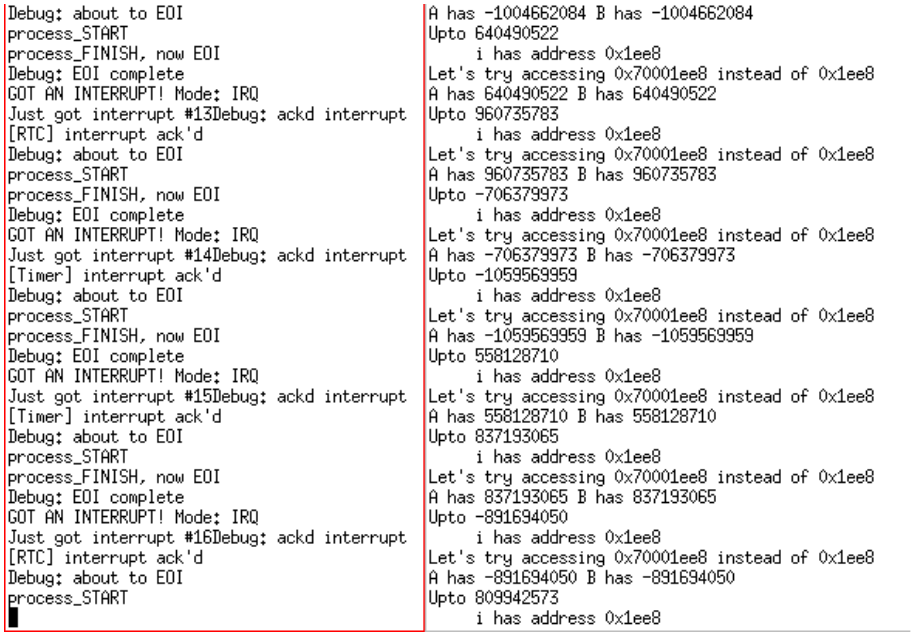


Figure 5.1: Two simple guests, interrupt test (left) and memory access tests (right)

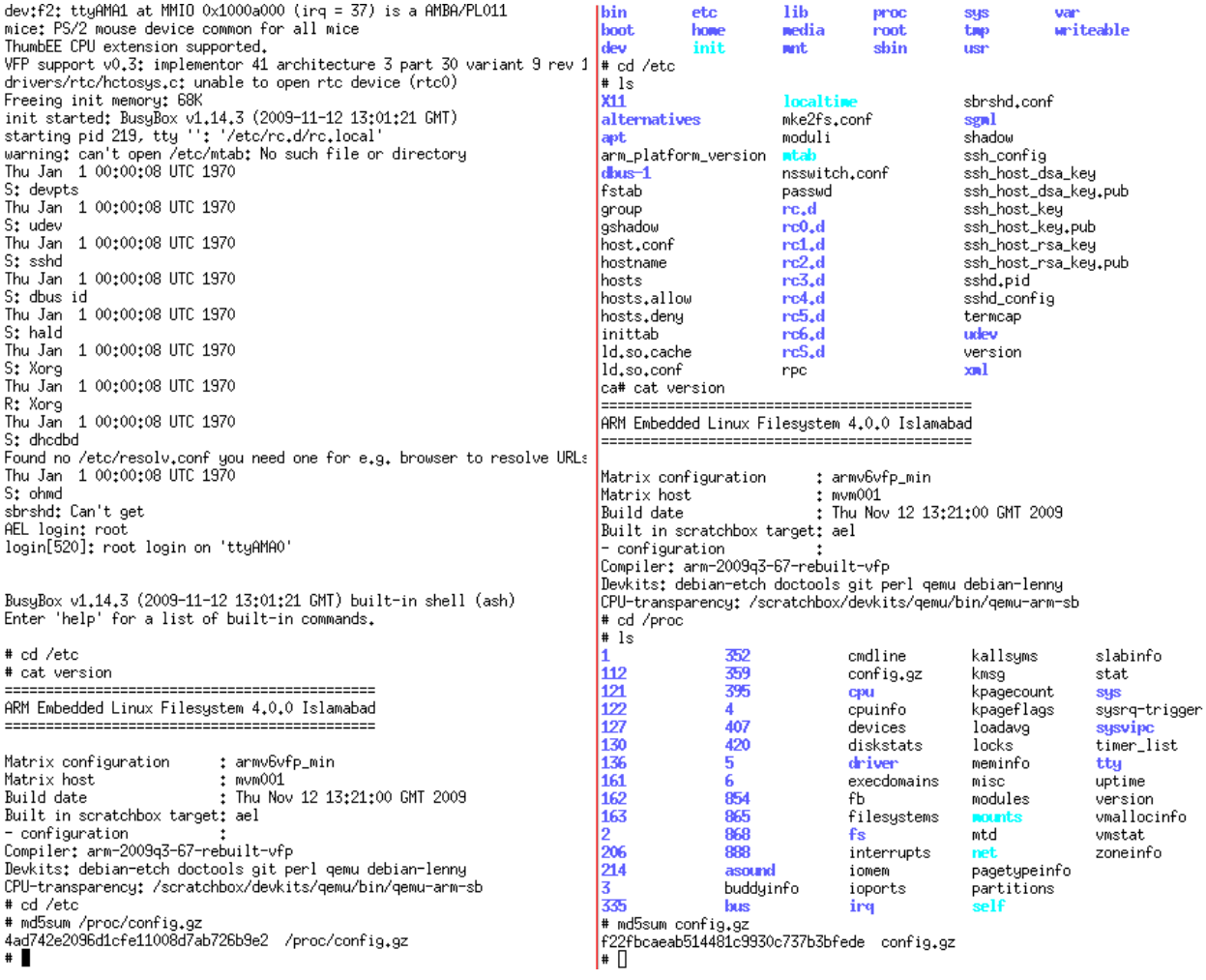


Figure 5.2: Multiple instances of Linux, with separate kernels

5.2 Inter-VM Communication

The hypervisor allows communication using synchronous buffers for small messages, or sharing pages for larger messages.

Figure 5.3 shows communication using small messages. The first guest sends messages, and then blocks till the hypervisor allows more messages to be sent. The hypervisor returns an error if the internal buffers are full, which ensures messages are not lost if the buffer is full. The second guest waits for interrupts, and prints messages received on every interrupt.

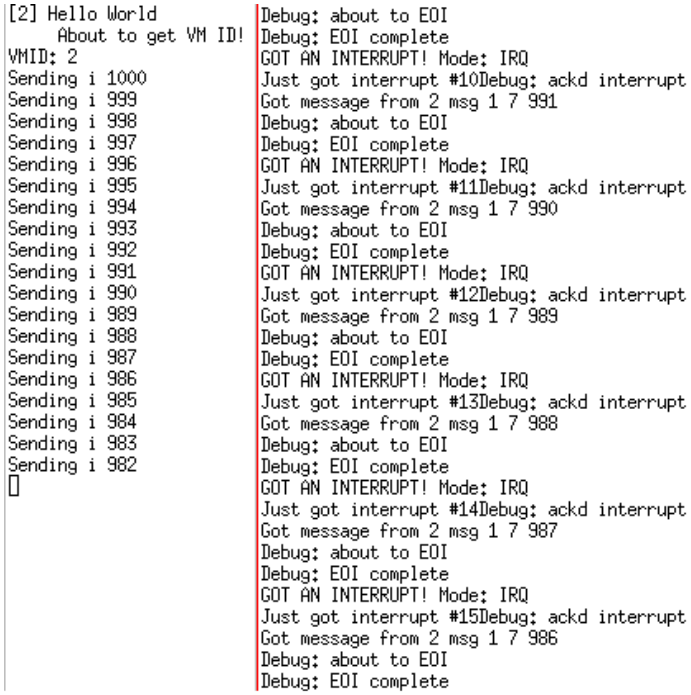


Figure 5.3: Simple message communication

The next example shows communication between a Linux instance and a microvisor application, through the virtual console driver written for Linux, shown in Figure 5.4. Writing to the */dev/hypcom* device causes output to be shown in the microvisor guest, as can be seen in Figure 5.5. The Linux driver shares a page with the microvisor application, and all data is communicated via the shared page for efficiency.

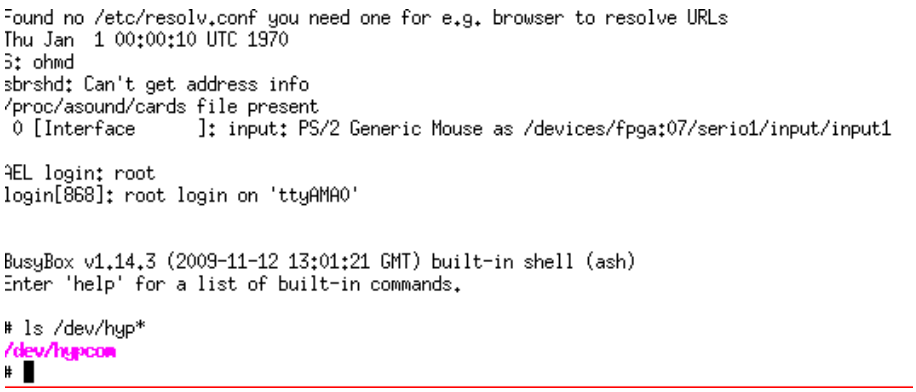


Figure 5.4: Virtual console driver for Linux

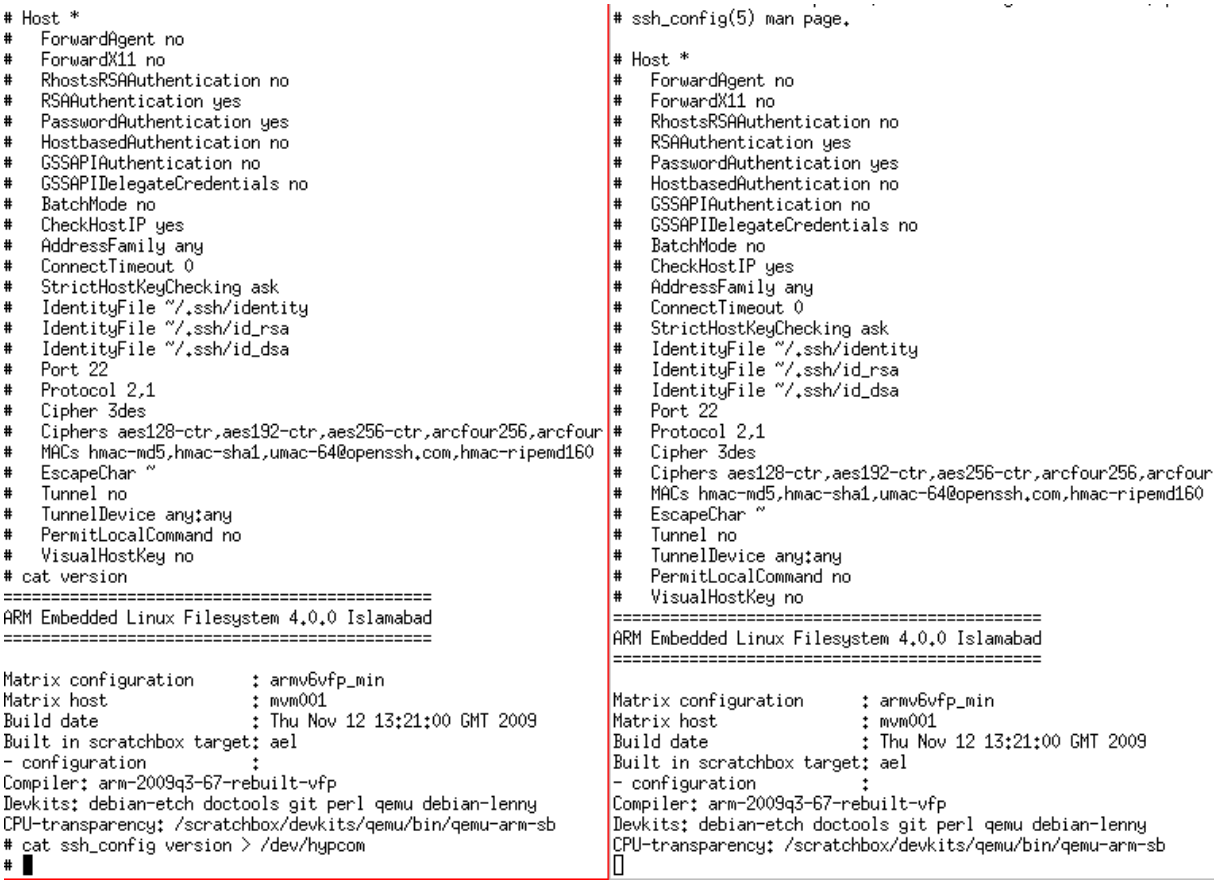


Figure 5.5: Communication between Linux (left) and the microvisor application (right)

5.3 Benchmark Configuration

All benchmarks are performed with the same version of Linux running natively and running virtualized on top of the hypervisor. The version of Linux used is 2.6.28, which was the latest build released by ARM at the start of this thesis. The para-virtualized version of Linux uses 2.6.29 running on top of the OKL4 microvisor.

The filesystem is based on the ARM minimal filesystem available on the ARM Linux website, and contains LMbench on top of the existing busybox-based filesystem.

The simulator is configured to run a single A9 core, and is reported as a 124Mhz processor. The exact list of parameters used to configure the simulator can be found in Appendix A.

5.3.1 Simulator timings

The simulator used is ARM's Fast Models, which does not accurately model cache timings, or memory access timings. However, the simulator does model the A9's 8-stage pipeline. All memory access is assumed to be zero wait state, and all instructions complete in one clock cycle. Memory is typically an order of magnitude slower than CPU speed. This leads us to only have access to instruction counts, rather than accurate cycle counts. The instruction count is useful for making sure that the implementation works as expected, and that there is no significant CPU overhead with the use of the new extensions.

When executing on hardware, factors such as TLB miss rate, cache miss rate, and memory accesses can increase the time taken for a single instruction. Furthermore, there are also instructions, such as co-processor accesses and supervisor calls, which do not complete in a single cycle and may require many more cycles.

Since all the benchmarks are run in the simulator, there is no variance when a benchmark is run multiple times. Hence the performance numbers given below are a single number, although benchmarks have been run multiple times to ensure that there is no variance.

5.4 LMbench

LMbench is a set of simple, portable benchmarks used to measure system performance [LMb10]. It consists of many bandwidth and latency micro-benchmarks which we use to

measure the overall impact of running Linux on top of the hypervisor.

The results of running the LMbench tests on a native instance of Linux, a pure virtualized instance of Linux, and a para-virtualized instance of Linux running on the OKL4 microvisor are shown in Table 5.1.

Table 5.1: LMbench test results

Test	Native	Virtualized	Para-virtualized
Syscall Entry (microseconds)	0.6012	0.6046	1.58
Context Switch (microseconds)			
2 processes	6.49	6.49	10.4
20 processes	8.75	8.83	13.78
Pipe Latency (microseconds)	34.9906	35.0727	49.3238
Pipe Bandwidth (MiB/s)	126.16	125.87	111.55
Process Creation (microseconds)			
Fork	886.0	869.43	2145.2724
Exec	914.5	917.5	2199.5283
Signal Handler (microseconds)			
Install	3.9677	3.9833	5.878
Catch	9.1019	9.1368	9.1368

The results shown are from the simulator, and as mentioned in Section 5.3.1, the instruction count is not an accurate indicator of real-world time. Hence the results only show the relative number of instructions executed, rather than the amount of actual time.

Para-virtualization shows large overheads when compared to native, although we expect overheads to be comparable to those described for the microvisor on the A8 core [HL10]. We use the newer and more optimised A9 core, although we are using an immature version of the microvisor; hence, we expect overheads to be similar to those described in the microvisor paper. However, our LMbench results show much larger overheads in some cases—null syscall shows 160% overhead compared to 60% in the microvisor paper, fork shows 142% compared to 8%! These large overheads are caused by the non-uniform amount of time taken by an instruction. Para-virtualization runs more instructions, although the amount of cycles each instruction takes is less on average.

We do expect higher overheads for para-virtualization in micro-benchmarks, due to way para-virtualization works. For example, with the syscall entry benchmark; when using para-virtualization, a syscall invokes the hypervisor, which then switches to the guest operating system running in user mode to process the syscall. Once processing is complete, the guest operating system invokes the hypervisor to return control to the

original user. In comparison, a standard syscall only switches mode once to the guest OS, then switches back to the user. Para-virtualization shows higher overheads due to the extra mode switches, and extra processing required.

There is almost no difference in the pure-virtualized LMbench numbers compared to native, and the average overhead of virtualization is less than 1% in all the tests. This is because the hypervisor is not invoked by any of the tests. All process creation and system calls are invoked without hypervisor intervention. The only performance overhead is from processing timer tick interrupts, as all interrupts invoke the hypervisor which sets up a virtual interrupt for the guest. However, we expect there will be some other performance overheads when run on hardware, described in Section 5.6.

These results show that pure virtualized Linux using the ARM extensions has very small overheads and may achieve better performance than para-virtualized Linux in micro-benchmarks.

5.4.1 Initial overhead

The above LMbench tests were run on a warmed up Linux instance which had physical pages allocated by the hypervisor. The hypervisor allocates physical pages on demand, and so when Linux first starts up, physical frames of RAM are not allocated by the hypervisor to Linux, and so the first access causes a trap into the hypervisor. These traps can cause significant overhead, but only occur the first time a page is accessed. For example, running the pipe bandwidth test on startup causes the performance to drop to 83.84MB/s.

This overhead is due to the on-demand allocation of physical pages by the hypervisor. The overhead can be completely removed by statically assigning pages before the guest is started, and can be optimised further by statically generating the page table at compile time.

5.5 Hypervisor Overheads

There are many common scenarios in which the hypervisor is invoked either due to a trap while a guest is running, or due to an interrupt request. We look at the instruction counts for common hypervisor operations such as:

- hypervisor entry
- interrupt latency
- page fault on a physical frame
- world switch
- inter-VM communication.

The specific tests and the purpose of each test is detailed in the following sections. The instruction counts below were calculated using the simulator instruction counter with carefully placed breakpoints. Some benchmarks were calculated using a new physical counter introduced with the new Eagle timers.

To get a rough idea of real-world performance rather than just instruction counts, we estimate hardware cycle numbers based on the type of instructions executed, and the average amount of cycles different types of instructions take on an A9 core with a board such as the Beagleboard.

5.5.1 Hypervisor entry

The hypervisor can be invoked by a guest using the HVC instruction. The *GetVMId* hypercall was used to check hypervisor entry performance as the hypervisor simply returns the current guest ID without any processing. This benchmark measures the total number of instructions to enter and leave the hypervisor.

Result: 170 instructions.

We break this down into different stages, and approximate how many cycles each stage would take on hardware in Table 5.2. We estimate 200 cycles for a hypervisor entry.

Table 5.2: Hypervisor entry cycles

Stage	Instruction count	Cycle count
Switch to *hyp* mode	1	20
Save all registers	45	50
Load reason for HVC entry	10	15
Handle HVC entry	60	60
Restore all registers	50	50
Leave *hyp* mode	1	20
Total	167	215

5.5.2 Interrupt latency

The hypervisor is invoked on every interrupt, and determines whether the current interrupt should be forwarded to the current guest as a virtual interrupt or not. This benchmark measures the number of instructions required before the interrupt is triggered to the guest.

Result: 460 instructions.

Interrupt entry requires much of same code as entering the hypervisor. However, it has an extra overhead to process the interrupt, route it to a guest, and add the interrupt to the guests' VCPU interface. This extra overhead should add less than 500 cycles to a hypervisor entry. In total, we would estimate interrupt latency to be approximately 700 cycles.

5.5.3 Page faults

When a guest accesses physical memory, the hypervisor may be invoked if there is no entry in the hypervisor page table for the guest IPA. The hypervisor then finds an unallocated physical frame, and adds a new IPA to PA mapping for the faulting address, then resumes the guest. This benchmark measures the number of instructions executed by the hypervisor for the whole page fault.

Result: 10350 instructions.

Once the CPU enters hypervisor mode, it checks the fault address, and processes the fault. Most of the instructions are memory modifications, although they modify a page at a time, so we expect a high cache hit rate. We assume that most instructions will not stall, and approximate page fault cycle count on hardware to be around 20000 cycles.

5.5.4 Device emulation

Shared devices are emulated in a hypervisor using trap-and-emulate techniques to present a virtual device to the guest. Since each access to the device is trapped, it is important to maintain good performance. With the ARM virtualization extensions, trap-and-emulate can be accelerated by emulation support, which provides details about the instruction that faulted. However, some instructions (such as those which load and write-back to a register) cannot be emulated. We benchmark both cases.

Result: Accelerated case: 510 instructions. Unaccelerated case: 620 instructions.

We have an overhead of about 200 cycles for entering the hypervisor, as discussed in Section 5.5.1. Once in the hypervisor, loading information about the instruction and decoding the register will take 100 or so cycles. We then need to find the specific device trap handler for the address, which takes another 200 cycles. Device emulation is device-dependant, although we assume an overhead of 500 cycles, which is approximately a slow timer device configuration change. This results in an estimate of 1000 cycles for a simple trap-and-emulate.

Note that the performance of an unaccelerated instruction would be significantly worse as the hypervisor must translate the guest virtual address to a guest physical address (IPA), followed by a translation to the PA, and then the instruction must be copied in for decoding by the hypervisor. This will cause an L1 miss, as the instruction will be in the I-cache rather than the D-cache. We assume the fetch and decode stages add approximately 1000 cycles in overhead, resulting in 2000 cycles.

5.5.5 World switch

A world switch occurs when the current running guest is switched to another guest. This is a crucial part of the hypervisor, as it switches between guests frequently, and so the performance affects total overhead significantly.

The benchmark is broken down into separate parts to show where most of the time is spent in Table 5.3.

Table 5.3: World switch parts

Stage	Instruction count
Enter hypervisor, save guest registers	50
Save current guest state	3880
Find new guest to schedule	45
Restore new guest state	3925
Leave hypervisor, restore guest registers	55
Total	7955

Saving guest state requires accessing the co-processor registers, which may require 4-5 cycles, although some co-processor accesses may use significantly more cycles. However, devices such as the interrupt controller and timer are also accessed for saving and restoring the guest state. We approximate that in total, it would take 5000 cycles for saving or

restoring of guest state. This leads to an approximate cycle count of around 11000 cycles on hardware. It is possible to reduce the amount of state switched by lazily switching devices such as the FPU.

5.5.6 Inter-VM communication

Inter-VM communication is very important for guests to share devices, and so this benchmark is useful to see the bandwidth that can be achieved for communication between guests.

The hypervisor currently only world switches on a timer interrupt, so to get a true idea of hypervisor overhead in passing a message from one guest to another, we modified the hypervisor to switch to the target guest when sending messages.

Result: 8340 instructions

Message passing has the same overheads as a world switch, with some extra overhead for modifying the VCPU interface to add a new interrupt and storing the message the hypervisor buffer. This requires some extra accesses to memory, but is approximately the same as a world switch. We estimate an overhead of 1000 cycles to buffer the new message and storing a new interrupt to the VCPU interface, and this results in an approximation of 12000 cycles on hardware for message passing.

The hypervisor by default does not world switch when passing messages. When it is configured to return to the calling guest, the number of instructions executed is 570 instructions. We approximate the number of cycles as 200 for the hypervisor trap, and an extra 1000 cycles to buffer the message and add the virtual interrupt, resulting in a total 1200 cycles.

5.5.7 Results overview

The above results have been collated for easier viewing of all the results in Table 5.4.

There were no big surprises in the above results, as it was expected that world switches would be slow, and message passing is limited due to the speed of the world switch. Page faults took longer than expected, but this is due to the extra code required to set up the page table, and due to the unoptimised C code currently used.

Table 5.4: Results overview

Test	Instruction count	Approximate cycle count
Hypervisor entry	170	200
Interrupt latency	460	700
Page fault	10350	20000
Device Emulation (accelerated)	510	1000
Device Emulation (unaccelerated)	620	1500
World switch	7955	11000
Message passing (with world switch)	8340	12000
Message passing (buffer, no world switch)	570	1200

5.6 Cache and Memory Impact

The above instruction counts are unable to show the true overhead of the hypervisor once cache and memory are taken into account, due to the lack of a timing-accurate simulator.

As the hypervisor is invoked on every interrupt, it will cause a slight impact on the guest. This can be minimised by creating a fast path for interrupts intended for the current guest that has a minimal footprint and so minimises the removal of guest TLB entries or guest cache lines.

Each memory access that is not in the TLB will also slow down significantly due to the extra page-table walk. If a guest is using virtual memory, each address is required to be translated initially by the guest page tables (typically 2 memory accesses), followed by the hypervisor page tables (typically 2 memory accesses), before retrieving the memory. The virtualized guest requires 5 memory accesses on a TLB miss compared to 3 memory accesses for a native or para-virtualized TLB miss.

5.7 TCB Complexity

We consider the impact on the trusted computing base of the OKL4 microkernel by integrating the virtualization extensions into the hypervisor. We look at the total lines of code, ignoring blank lines and comments, of the different components in the hypervisor in Table 5.5.

The hypervisor increases the existing TCB of the OKL4 pico microkernel, roughly 5000 LOC, by approximately 3500 LOC. This is much smaller than the impact of implementing virtualization extensions with other architectures such as x86 [SK10].

Table 5.5: TCB impact by component

Component	Line count
Initialisation	320
Device Emulation	480
ELF Loader	490
Interrupt Handling	1060
Page Table	320
Guest Switching	570
Hypercalls	190
Total	3430

Chapter 6

Conclusion

The main goal of my thesis was to run multiple unmodified Linux guests concurrently, using the ARM virtualization extensions. I was able to achieve my goal in creating possibly the first hypervisor running unmodified Linux guests using the new virtualization extensions.

I also achieved inter-VM communication through hypercalls, and was able to use the communication mechanisms to emulate a simple shared driver host and client example.

Due to the lack of hardware, and the lack of a timing-accurate simulator, I was unable to benchmark the real overhead of using the virtualization extensions for pure virtualization compared to para-virtualization and running native. However, I was able to give rough performance overheads based on the instruction counts and instruction types. The hypervisor has not been optimised, yet the instruction counts and cycle approximations show that the overhead is quite small, and can be reduced further in an optimised implementation.

6.1 Architecture Evaluation

This thesis shows that the ARM virtualization extensions allow for a simple hypervisor, while reducing the overhead of pure virtualization by adding hardware support for emulation. We discuss the specific extensions in more detail below, then discuss possible improvements to the architecture.

6.1.1 New mode

The new *hyp* mode for management of virtual machines is similar to the *VMX-root* mode added by Intel. It allows guest operating system kernels to continue running in supervisor mode, avoiding problems with sensitive instructions that do not trap when run in user-mode.

Since *hyp* mode does not use the standard virtual memory configuration, the configuration does not need to be modified on every entry, which allows for efficient entry into *hyp* mode.

This new mode is used for many purposes when the hypervisor is enabled including:

- entry on a hypervisor trap or HVC instruction
- interrupt handling
- memory fault caused by a guest in supervisor mode
- memory fault caused by the hypervisor in *hyp* mode.

These functions are typically separated into different modes, but the virtualization extensions only use a single mode for the different purposes. This change requires the hypervisor to check a new register (the hypervisor syndrome register) on every entry to find the reason for entering *hyp* mode. This new method is inconsistent with ARM's standard use of separate modes, and separate vectors for each trap.

TrustZone

This thesis shows that the hypervisor mode can be used to isolate virtual machines safely, achieving the same goal as TrustZone, but allowing multiple worlds rather than just a secure world and a non-secure world. The virtualization extensions and the new *hyp* mode bring a superset of functionality to the standard TrustZone extensions, while allowing more flexibility. However, ARM has chosen to require the TrustZone extensions for virtualization, which causes unnecessary complexity when initialising the hypervisor, and ensuring the hypervisor can access all guests, as can be seen in Section 4.2.1.

6.1.2 Second-stage translations

The support for second-stage translations is effective in reducing virtualization overhead normally associated with MMU virtualization. There are no traps for guest use of virtual memory, although there is a small overhead in extra page table walks. The support for block mappings allows us to reduce the number of page-table walks, hence reducing the overhead of second-stage translations.

A possible improvement could be to optimise the current save/restore for virtual memory configuration by allowing a faster way to save all state to memory, and restore state from memory. Currently, all state is saved by manually moving from co-processor registers.

6.1.3 Virtual interrupt support

The virtual CPU interface reduces overhead significantly for guest interrupts. Guests access the VCPU interface multiple times for each interrupt, and the hardware support removes the need to trap into the hypervisor for each of these accesses.

However, all interrupts currently invoke the hypervisor, which adds latency to interrupts.

Interrupt to guest

Interrupt latency can be reduced by not invoking the hypervisor on every interrupt. Extra configuration could be added to allow specific interrupts to be bypassed and delivered to the guest directly. This would reduce hypervisor traps, and allow the hypervisor to only be invoked on interrupts intended for it, or on interrupts intended for a guest that is not running.

6.1.4 Emulation support

ARM's emulation support allows for a simpler hypervisor which does not need a full ARM emulator, while also reducing the time required for device emulation using trap-and-emulate significantly.

Trapped instruction store

When the hypervisor traps on an instruction, and the instruction is not accelerated due to the type of instruction (e.g, it uses write-back), then the hypervisor currently translates the faulting address manually from a guest virtual address to a physical address, and then fetches the instruction. These steps could be removed completely if the architecture stored the trapped instruction in a register.

This optimisation would only improve performance if unaccelerated traps for emulated devices are common. I only implemented very simple devices, which are not enough to predict the performance benefit in the real world.

6.2 Future Work

There are many areas to consider for future work, due to how recently the ARM virtualization extensions were released. We only consider directly related work, some of which were not considered due to the limited timeframe of the thesis.

6.2.1 Microkernel and hypervisor integration

The current hypervisor stops running microkernel applications once the hypervisor starts scheduling guests. Future work can modify the design to ensure both guests and microkernel applications are run concurrently. This can be achieved in two ways:

- Split the microkernel API into a separate component running in supervisor mode that interacts with microkernel applications, while being scheduled by the hypervisor as a guest.
- Expand the microkernel API to allow guests to be mapped on to existing thread/address space concepts.

6.2.2 User-level device virtualization

Device emulation can be moved out of the hypervisor, and be run as a standard user application. This would allow for a smaller hypervisor, while ensuring that bugs in the emulation of a large complex device do not impact the security of the hypervisor.

6.2.3 Lazily switch guest state

On a world switch, we currently save all guest state—even when it has not been modified. It is possible to reduce the amount of state saved by using traps to detect state changes, and only switching state that has been modified.

6.2.4 Effect on para-virtualization

Although the ARM extensions allow pure virtualization, they can also be used to reduce the overheads of para-virtualization. Higher performance for para-virtualization is possible by using the new extensions support for running the hypervisor is a higher privilege mode and faster interrupt processing using the new virtual interrupt support.

Appendix A

AEM Configuration Parameters

```
coretile.core.cpuID=0x410fc090
coretile.core.delayed_CP15_operations=0
coretile.core.take_ccfail_undef=0
coretile.core.multiprocessor_extensions=1
coretile.core.mp_peripherals=1
coretile.core.num_cores=1
coretile.core.implements_ple_like_a8=0
coretile.core.vmsa.separate_tlbs=0
coretile.core.vmsa.implements_fcse=0
coretile.core.vmsa.tlb_prefetch=0
coretile.core.vmsa.infinite_write_buffer=0
coretile.core.vmsa.memory_marking_check=1
coretile.core.vmsa.cache_incoherence_check=0
coretile.core.vmsa.main_tlb_size=64
coretile.core.vmsa.main_tlb_lockable_entries=4
```

Bibliography

[AA06] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *ASPLOS-XII: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–13, San Jose, California, USA, 2006. ACM.

[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. In *Proceedings of the 1986 Summer USENIX Technical Conference*, pages 93–112, Atlanta, GA, USA, 1986.

[ARM05] ARM Ltd. *ARM1136JF-S and ARM1136J-S Technical Reference Manual*, R1P1 edition, 2005.

[ARM10a] ARM unveils Cortex-A15 MPCore processor to dramatically accelerate capabilities of mobile, consumer and infrastructure applications. `http://www.arm.com/about/newsroom/news-2010.php`, 09 2010.

[ARM10b] TrustZone - ARM. `http://www.arm.com/products/processors/technologies/trustzone.php`, 05 2010.

[ARM10c] ARM Architecture Group. *Large Physical Address Extensions Specification*, 2010.

[ARM10d] ARM Architecture Group. *Virtualization Extensions Architecture Specification*, 2010.

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art

of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 164–177, Bolton Landing, NY, USA, October 2003.

[Bie06] Sebastian Biemüller. Hardware-supported virtualization for the L4 microkernel. Master's thesis, Universität Karlsruhe, 2006.

[CB07] Simon Crosby and David Brown. The virtualization reality. *Queue*, 4(10):34–41, 2007.

[Fer06] Daniel R. Ferstay. Fast secure virtualization for the ARM platform. Master's thesis, University of British Columbia, 2006.

[FO06] Jogn Fisher-Ogden. Hardware support for efficient virtualization. Technical report, University of California, San Diego, 2006.

[Gre10] INTEGRITY Secure Virtualization. `http://www.ghs.com/products/rtos/integrity_virtualization.html`, 05 2010.

[Hei08] Gernot Heiser. The role of virtualization in embedded systems. In *1st Workshop on Isolation and Integration in Embedded Systems*, pages 11–16, Glasgow, UK, April 2008. ACM SIGOPS.

[Hei09] Gernot Heiser. Hypervisors for consumer electronics. In *Proceedings of the 6th IEEE Consumer Communications and Networking Conference*, pages 1–5, Las Vegas, NV, USA, January 2009.

[HHL+97] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of μ-kernel-based systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 66–77, St. Malo, France, October 1997.

[HL10] Gernot Heiser and Ben Leslie. The OKL4 Microvisor: Convergence point of microkernels and hypervisors. In *Proceedings of the 1st Asia-Pacific Workshop on Systems*, New Delhi, India, August 2010.

[HSH+08] Joo-Young Hwang, Sang-bum Suh, Sung-Kwan Heo, Chan-Ju Park, Jae-Min Ryu, Seong-Yeol Park, and Chul-Ryun Kim. Xen on ARM: System virtualization using Xen hypervisor for ARM-based secure mobile phones.

In *Proceedings of the 5th IEEE Consumer Communications and Networking Conference*, pages 257–261, Las Vegas, NV, USA, January 2008.

[Kro09] Kirk L. Kroeker. The evolution of virtualization. *Communications of the ACM*, 52(3):18–20, 2009.

[Lie93] Jochen Liedtke. Improving IPC by kernel design. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 175–188, Asheville, NC, USA, December 1993.

[Lie95] Jochen Liedtke. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 237–250, Copper Mountain, CO, USA, December 1995.

[LMb10] LMbench - tools for performance analysis. `http://www.bitmover.com/lmbench/whatis_lmbench.html`, 10 2010.

[MP07] Karissa Miller and Mahmoud Pegah. Virtualization: virtually at the desktop. In *SIGUCCS '07: Proceedings of the 35th annual ACM SIGUCCS conference on User services*, pages 255–260, Orlando, Florida, USA, 2007. ACM.

[NIC10] libelf. `http://ertos.nicta.com.au/software/kenge/libelf/devel/`, 05 2010.

[OKL10] Open Kernel Labs Website. `http://www.ok-labs.com/about/about-ok-labs`, 10 2010.

[PG74] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):413–421, 1974.

[SK10] Udo Steinberg and Bernhard Kauer. NOVA: A microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th EuroSys Conference*, Paris, France, April 2010.

[UNR+05] Rich Uhlig, Gil Neiger, Dion Rodgers, Fernando C. M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kägi, Felix H Leung, and Larry Smith. Intel virtualization technology. *IEEE Computer*, 38(5):48–56, May 2005.

[Vir10] VirtualLogix - Real-time Virtualization for Connected Devices. http://www.virtuallogix.com/solutions/product/arm.html, 05 2010.

[VMW06] VMWare. A performance comparison of hypervisors. Technical report, VMWare, 2006.

[VMW08] VMWare. Understanding full virtualization, paravirtualization, and hardware assist. Technical report, VMWare, 2008.

[WSG02] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Denali: A scaleable isolation kernel. In *Proceedings of the 10th SIGOPS European Workshop*, pages 9–15, St Emilion, France, September 2002.