# Fast Secure Virtualization for the ARM Platform

by

Daniel R. Ferstay

B.Sc., University of British Columbia, 2001

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

**Master of Science**

in

THE FACULTY OF GRADUATE STUDIES

(Computer Science)

**The University of British Columbia**

March 2006

# Abstract

In recent years, powerful handheld computing devices such as personal digital assistants and mobile smart phones have become ubiquitous in home and office environments. Advancements in handheld device hardware have driven the development of the software that runs on them. As these devices become more powerful, increasingly connected, and the tasks performed by their operating systems more complex there is a need for virtual machine monitors.

Virtual machine monitors such as the Xen hypervisor developed at the University of Cambridge could bring an increased level of security to handheld devices by using resource isolation to protect hosted operating systems. VMMs could also be used to place constraints on the resource utilization of hosted operating systems and their applications.

Xen is closely tied to the x86 computer architecture and is optimized to work well on desktop personal computers. One of its design goals was to provide virtualization on x86 computers similar to that which was previously found only on IBM mainframes designed to support virtualization. We aim to provide this same style of virtualization on mobile devices, the majority of which are powered by the ARM computer architecture. The ARM architecture differs considerably from the x86 architecture. ARM was designed with high performance, small die size, low power consumption, and tight code density in mind.

By migrating Xen to the ARM architecture, we are interested in gaining insight into the capacity of ARM powered devices to support virtual machines. Furthermore, we want to know which of the StrongARM's architectural features help or hinder the support of a Xen-style paravirtualization interface and whether guest operating systems will be able to run without modification on top of a StrongARM based hypervisor.

In this thesis, we describe the design and implementation issues encountered while porting the Xen hypervisor to the StrongARM architecture. The implementation of a prototype has been carried out for SA-110 StrongARM processor and is based on the hypervisor in Xen version 1.2.

# Contents

# List of Tables

# List of Figures

# Acknowledgements

A tremendous thank you to Norm for making this work possible as my supervisor and mentor. Thank you to Mike for his time and input into this thesis. Thanks also to Andy Warfield for his thoughts on possible applications of the Xen hypervisor in a mobile environment and to the members of the DSG lab for their thoughts and for listening when I had something on my mind. Finally, thank you to my family for all of their love and support.

Daniel R. Ferstay

*The University of British Columbia*
*March 2006*

To the loving memory of my grandfather.

Chapter 1

# Introduction

In recent years, powerful handheld personal computing devices have become ubiquitous in home and office environments. Personal digital assistants and mobile smart phones are equipped with as much processing power as mainframes had fifteen years ago. They have also become increasingly connected with support for wireless networking protocols such as Bluetooth and 802.11b built in. The advancements in handheld device hardware have driven the development of the software that runs on them. Users now enjoy applications such as streaming audio and video, audio and video capture, still image photography, and networked games. Such a variety of applications would not be possible without a powerful general purpose operating system to manage the hardware and provide a standardized set of services to applications. A variety of operating systems are currently being used by different hardware vendors. [1]

As handheld devices become more powerful and the tasks to be performed by the operating systems more complex there are opportunities to make use of

[1]Sharp makes PDAs that run Linux, HP makes PDAs that run WindowsCE, and Nokia smart phones run the Symbian OS.

Virtual Machine Monitors. VMMs could bring an increased level of security to handheld devices by using resource isolation to protect hosted operating systems. VMMs could also control the resource utilization of hosted operating systems and applications. VMMs are extremely useful for operating systems development and debugging where working with a virtual machine gives the systems developer many more development tools and more control over the execution of the system than is provided by bare hardware. For example, developing new drivers for the Linux operating system is easy using the Xen VMM. In Xen a module can be isolated into its own virtual machine [FHN+04]. If the driver crashes or hangs, its state can be immediately inspected by debugging the virtual machine. To restart testing, only the virtual machine that hosts the driver needs to be restarted instead of rebooting the entire machine, saving precious time.

In this thesis we intend to provide a virtualization infrastructure for handheld devices. The ARM [Sea00] architecture is the most widely used in the embedded design space. In addition, ARM processors offer a mix of performance and low power consumption that makes them attractive as the basis for our virtualization infrastructure. Thus, we ported the Xen hypervisor to the ARM computer architecture.

Such a port interesting for several reasons. Most importantly, determining the capacity of the ARM architecture to support virtual machines and which of its architectural features help or hinder the support of a Xen-style paravirualization interface. We also hope to gain some insight into how portable the Xen hypervisor is by examining the issues that arise while attempting the port. Finally, we would like know whether operating systems hosted on the ARM port of the Xen hypervisor will be able to run unmodified (such as the PowerPC and IA 64 versions of Xen) or if they require further porting (such as the x86 version of Xen). These are the interesting issues that we plan to address during the course of this thesis.

## 1.1 Overview

Xen was designed specifically for the x86 family of processors [Int05a]that power most desktop personal computers today. One of the goals behind Xen's development is to provide virtualization on x86 hardware that is similar to what was previously only found on IBM mainframe computers specifically designed to support virtualization such as the z-series [Fra03]. By porting the Xen hypervisor to ARM processors we hope to achieve a similar style of virtualization on small handheld devices.

## 1.2 Motivation

It is the employment of general purpose operating systems on handheld devices that allows intruders to use the same attacks that have been proven successful against desktop and server systems. Moreover, as the power and connectivity of mobile devices increases and their use becomes more widespread, the opportunities and motivation for the creation of viruses, worms, and other malware will also increase. At the time of this writing, a variety of worms have been created that affect the Symbian OS. For example, the Cabir.a, Mabir.a [Wro05], MetalGear.a [Gar04], and CommWarrior.a [Sun05] worms are all capable of propagating themselves via Bluetooth and/or Multimedia Message Service (MMS) connectivity available on Symbian powered smart phones. Others, such as the Cardtrap.a worm [Kaw05] can spread from smart phones to personal computers via a memory card when a synchronization operation between PC and mobile device is attempted. Moreover, the volume of mobile-device malicious software is rising rapidly. As of September

2005, 83 different viruses have emerged within a 14-month period [Kaw05]. This clearly illustrates the vulnerability of mobile devices and the need for a host based security system that can harden the device from attack.

Some companies such as SimWorks [Sim05] have attempted to achieve this by producing anti-virus applications software for mobile phones. However, these applications depend upon the integrity of the operating system. In other words, if a system is compromised the output generated by anti-virus software running on the system can no longer be trusted. For example, the payload carried by worms such as MetalGear.a simply disables any anti-virus software.

There has been a lot of previous work on host based security for general purpose computers. The most common security systems are applications that run on top of the host operating system that monitor the state of certain sensitive data on the system. The Tripwire intrusion detection system [Tri05] monitors the file system of the host for any suspicious activity and writes its observations to a log file protected by the host operating system kernel. Another such application is checkps [che05]. It checks the validity of the output of the ps program on Unix-like operating systems. This is effective since it is common for intruders to hide their activities by providing fake ps output.

While these tools are useful, they do have their limitations. For instance, there are pieces of the system that cannot be inspected from the application level. For this reason security tools have been built into the kernel of various operating systems. One such tool is KSTAT (Kernel Security Therapy Anti-Trolls) [s0f05]. It lives inside of a Linux kernel module and as such it has all of the privileges afforded to other parts of the kernel. It can also perform sanity checks on kernel data structures to check for things such as system call modifications and Trojan kernel modules.

Application and kernel based security systems both provide detection of malicious applications and kernel intrusion. However, they both suffer from the same problem; they rely on the integrity of the host operating system. Application level security logs (as used in Tripwire) might be good for identifying intruders, but what if an intruder gained access to the host OS kernel? At that point the intruder could remove or modify any logs written by an application-based security system. Similarly, once an intruder is inside of the kernel they could check for kernel based security systems and either disable them or modify their output.

It is for this reason that kernel integrity monitors have been developed. A kernel integrity monitor watches the operating system as it operates in an attempt to identify and correct any deviant behavior. One such system for general purpose computers is Copilot [JFMA04]. In Copilot, the monitor program runs not on the CPU and memory of the main system, but on a separate CPU and memory located on a PCI expansion card. The monitor program watches the kernel operate and is able to detect various forms of attack on the kernel. However, the key benefit to the Copilot system is that it is isolated from the host operating system. Thus, it is able to continue correctly monitoring the system and detect intrusions even after the host operating system has been compromised. Unfortunately, most current handheld devices do not support expansion cards that can contain their own CPU and memory. Most of the expansion slots on handheld devices are for increasing the memory or persistent storage capabilities of the device. In fact, even if handhelds did support more heavy weight expansion cards, there would still be drawbacks to using a system similar to Copilot. The main drawback being that using the extra CPU on the expansion card would consume energy. While most general purpose computer systems are plugged into a wall outlet and do not have to worry about running out of batteries, handheld devices do not have such a luxury. It seems that if there was a software solution to the problem of monitoring an operating system kernel for intrusion detection that had the properties of the Copilot kernel monitor, then the result could be used with some success on a handheld device.

Virtual machines provide a software abstraction of a real machine that operating systems software may be run on top of. In this way, the virtual machine isolates the operating system from the real machine. A virtual machine monitor is a thin layer of software that manages one or more Virtual Machines. The VMM is isolated from the Virtual Machines that run on top of it, can determine the inner workings of a hosted virtual machine (introspection), and can intercept information entering and leaving the virtual machine (interposition). Essentially, a VMM can provide in software what the Copilot kernel integrity monitor provides in hardware. Of course, there are tradeoffs between a hardware based system and a system that is software based. The hardware based monitor is less flexible but might have a smaller negative impact on the performance of the system. It is accepted that both hardware and software approaches to machine monitoring are useful for different application domains and that both are worthy of future research. This paper focuses on the use of software based machine monitor systems because it is my belief that they are better suited to hand held devices and that their flexibility makes for easier adoption by vendors of said devices.

It is the properties of isolation, introspection, and interposition that have lead researchers to examine the use of VMMs as the basis for kernel integrity monitors, intrusion detection systems, or secure services. Not only this, but a VMM is superior to a general purpose OS for trusted security applications because a VMM is much simpler in terms of code size and complexity. The Disco VMM was implemented in about 13000 lines of C code. VMMs also present a small, simple interface to the guest operating systems that run on top of them: that of the hardware they are running on or something very similar. Compare this to the size and complexity of a Unix system call API and you will understand that a VMM's behavior is cleanly defined by the hardware/software interface whereas an OS system call API leaves many opportunities for corner cases and undefined behavior to be exposed. On the ARM processor there are 32 well defined instructions that can be executed by the machine. In comparison, the Linux kernel contains a system call API that is made up of roughly 1100 different system calls. What follows are some examples of recent research that has used VMMs for host based security purposes.

In ReVirt [DKC+02] a secure logging facility was built into the VMM for intrusion analysis. By placing the logging facility inside of the VMM the integrity of the logs and the logging system are not compromised even if the hosted operating system has been infected by a virus, cracked by an intruder, or has been otherwise rendered untrustworthy.

In Livewire [GR03], an intrusion detection system was created out of a variety of services built at the VMM level. The usefulness of Livewire depended on its isolation from the hosted OS kernel, but it also used the property of introspection to examine the state of the OS as it ran. There are many resources that are common for intruders to acquire for the purpose of breaking into the system and Livewire simply watched for these types of resource acquisitions. For example, raw sockets are commonly used for two purposes: network diagnostics and low level network access with the intent to break and enter a computer system. Livewire simply watched for and flagged any access to raw sockets. In addition to watching for resource acquisitions, the Livewire system could also interpose itself between the OS and the potentially harmful applications. This type of system-call interposition was useful for providing sanity checks on the arguments of system calls to avoid buffer overflow attacks.

## 1.3 Methodology

Processors based on the StrongARM computer architecture are 32-bit RISC style chips that offer the following features:

- High performance.
- Small die size.
- Low power consumption.
- Tight code density.

These features have lead to ARM being an attractive option for use in embedded systems. Indeed, many manufacturers use ARM in devices such as hard drives, routers, calculators, children's toys, mobile phones and PDAs. Today over 75% of all 32-bit embedded CPU's are based on the ARM design. Therefore, we chose the ARM as our target architecture to explore the use of VMMs in a mobile computing environment. More specifically, we ported the Xen Hypervisor to the SA-110 implementation of the StrongARM architecture [Int98a]. It should be noted that the SA-110 core is not widely used in industry, with companies opting to use the SA-1100 core in its stead. However, the SA-110 core is identical to the SA-1100 processor sans integrated controllers. In affect, the SA-1100 can be thought of as an SA-110 processor surrounded by integrated controllers for memory, interrupts, DMA, serial, and real-time clock [Int98b][2]. While writing the port there was a great effort to modify only the architecture dependent portions of the Xen source code whenever possible.

## 1.4 Synopsis

In the following chapters we present the issues related to porting the Xen Hypervisor to the ARM architecture and explain the design decisions that we made. Chapter 2 presents related work and gives a background in resource virtualization. Chapter 3 presents a comparison of the Intel x86 and StrongARM computer architectures with respect to virtualization. Chapter 4 explains the issues encountered during the implementation of the Xen Hypervisor port to ARM. Chapter 5 evaluates the performance of our Xen hypervisor on a StrongARM based platform. We present conclusions in Chapter 6 along with suggestions for future work.

[2]We targeted the SA-110 CPU because we did not have access to hardware powered by the SA-1100 processor.

# Chapter 2

# Related Work

In this chapter we present a variety of Virtual Machine Monitor Architectures and make a case for using paravirtualization on small mobile handheld devices.

## 2.1 Virtual Machine Monitor Architectures

This section presents a few different VMM architectures. However, before they are discussed in detail some basic terminology should be understood. The VMM is a software layer that manages one or more virtual machines. A virtual machine is a software abstraction of the real machine that is isolated from the VMM and is also referred to as a domain. Domains can contain any type of software application, but in the context of this paper the application will be an Operating System. The operating system running in a domain is referred to as the guest operating system. The applications running on top of the guest operating system are called the guest applications. The VMM itself can be run on top of hardware or software; we name both the host architecture. Figure 2.1 shows the basic organization of a system that uses a VMM; the hardware hosts the VMM while the guest OS and applications make up a domain. Next, we present a few of the different VMM architectures being used today.

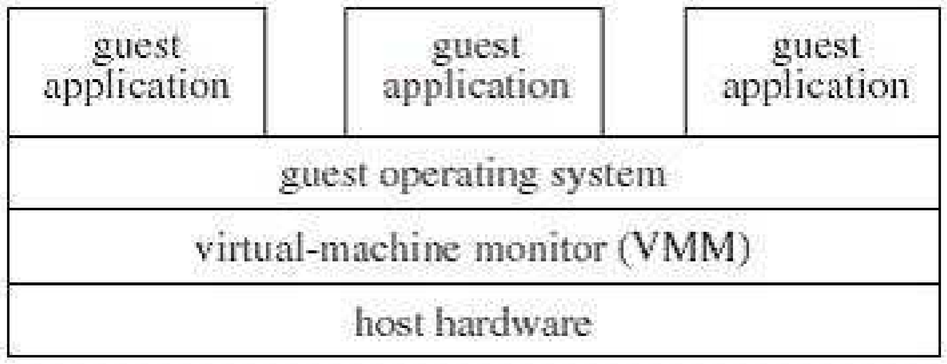


Figure 2.1: The organization of a system that uses a hardware hosted virtual machine monitor to support a single domain.

### 2.1.1 Nano-kernel systems

The basic idea behind Nano-kernels is that they are a separate, small bit of core functionality that resides within the OS to do some specialized processing (e.g. Real-time interrupt processing and scheduling). Nano-kernels can be considered a lightweight version of a VMM because typically they do not virtualize very many resources. Examples of nano-kernel systems are Jaluna [Jal05] and RTLinux [FSM05]. In both of these systems, the virtualized resource is interrupt handling and scheduling. In RTLinux, there is a small real-time core that is loaded as a Linux kernel module and runs alongside of the regular Linux kernel. When the RTLinux core is loaded, it takes over all of the interrupt processing and scheduling for the machine by interposing itself at the interrupt handler routine entry points. The RTLinux core then handles all interrupt processing and schedules the actions of the system -including the Linux kernel and its scheduler- according to hard, real-time requirements dictated by the system administrator.

It is possible extend nano-kernel systems with monitoring capabilities for security purposes. For example, at a recent visit to UBC, Victor Yodaiken (CEO of FSMLabs; creators of RTLinux) stated that they were attempting to modify the RTLinux core so that it could monitor the Linux kernel and perform integrity checks of the system for security purposes. Essentially, the proposed scheme had the RTLinux core take a snapshot of the running Linux kernel. Then, between every real time scheduling decision, ( 1ms) the RTLinux core could perform a quick integrity check on the kernel snapshot, possibly in the form of a hash computation. There are two problems with this scheme. First, an intruder may be able to break into the system and cause damage before you can notice it. In other words, the intruder's actions could happen within the bounds of a 1ms scheduling window. This problem becomes exacerbated as the speed of CPUs increase since applications will be able to execute more instructions in the 1ms scheduling window. Thus, the scheduling period must be changed depending on the speed of the machine. Second, if an intruder does break into the system and is able to compromise the Linux kernel, it can also compromise the RTLinux kernel based monitor system. This is because the RTLinux kernel is not isolated from the Linux kernel in any way. Both kernels run alongside each other in the same address space (kernel space) with the same privileges (kernel mode execution). For the most part, nano-kernel architectures have been dubbed cooperative virtualization environments. This is because the VMM is not isolated from the domains that it hosts, and thus makes exchange of information between the VMM and any domain very efficient. However, it is the same cooperation that leaves nano-kernel based systems vulnerable to attack. Therefore a nano-kernel based monitor system may not be the best choice for a trusted VMM based security system.

### 2.1.2 Full Virtualization Systems

Full virtualization systems virtualize every resource on a computer system. Guest operating systems do not have access to physical devices, physical memory, or physical CPUs. Instead, the guest operating systems are presented with virtual devices, virtual memory, and virtual CPUs. In a full virtualization system, the virtualized interfaces presented to the guest operating system look and feel exactly like the interfaces of the real machine and therefore the guest OS and applications may run on the virtual hardware exactly as they would on the original hardware. One such system is VMware [SVL01, Wal02, VMw05]. Full virtualization systems have another benefit besides allowing guest operating systems to run unmodified on top of the VMM. The virtualized resources provide a layer of isolation that protects the VMM from the actions of guest operating systems. The virtual memory implementation in the VMM protects it from the guest OS in the same way that the virtual memory implementation in the guest OS protects it from the guest applications [20]. In a similar way, the virtual CPU implementation in the VMM uses the processor's operating modes in the same way that the guest OS uses the CPU operating modes to protect itself from guest applications [SS72]. In fact, VMMs built in the full virtualization style are considered the most secure of all VMM architectures because of the strong isolation they provide. The Livewire project is the first attempt to build a security service into the full virtualization VMM; they used VMware as a base.

The major negative characteristic of full virtualization systems is degraded performance. This is obvious as every privileged instruction, every interrupt, every device access, and every memory access (etc.) has to be virtualized by the VMM. The performance measurements carried out during the evaluation of the Xen VMM [DFH$^+$03] confirmed that a guest OS running on top of VMware can be as much as a one hundred times slower than the same OS running directly on hardware.

### 2.1.3 Paravirtualization Systems

The architecture of paravirtualization systems has very much the same look and feel as that of a full system virtualization system but with major differences in design decisions. Paravirtualization was born out of the observation that full system virtualization is too slow and complex on today's commodity hardware. For the most part, this is because the VMM must intervene whenever a domain attempts to execute privileged operation. Paravirtualization aims to retain the protection and isolation found in the full system virtualization approach but without the implementation complexity and associated performance penalty. The main idea behind paravirtualization is to make the VMM simpler and faster by relaxing the constraint that guest operating systems must run on the VMM without being modified. In a paravirtualization system, the guest operating system code is modified to access the VMM directly for privileged access, instead of going to the virtual resources first and having the VMM intervene. One VMM implementation that uses a paravirtualization approach is the Xen hypervisor [DFH$^+$03]. Xen runs on x86 computer systems and supports commodity operating systems (Linux, NetBSD, WinXP (in development)) once they have been ported to the Xen-x86 hardware architecture. The Xen-x86 architecture is very similar to x86, with extra function calls into the VMM needed for virtual memory page table management and other features of the x86 that are difficult or slow to fully virtualize. Operating systems ported to Xen-x86 must also port their device drivers to use Xen's lightweight event notification system instead of using interrupts for communications. In addition, Xen is very lightweight at roughly 42000 lines of code [BDF$^+$03].

Another VMM that uses a paravirtualization approach is Denali [WSG02]. Denali has the same basic goals as Xen, although Denali makes no attempt to support commodity operating systems. For this reason, Denali can give up certain features that are difficult to fully virtualize or paravirtualize such as virtual memory. In Denali, a domain is meant to support a single application (or OS). If you need two applications to be isolated from one another then you must run them in separate domains (virtual machines). Denali forfeits features needed for operating system support in order to gain simplicity and security.

The main drawback to paravirtualization systems is that they cannot host commodity operating systems without first porting them to run on the VMM. To port an OS to a new architecture takes time, but work on the Xen VMM shows that if the target architecture is very similar to the original architecture and most of the major changes to the OS are architecture independent, then the cost of porting the OS is outweighed by the rewards of paravirtualization. Another worry for VMMs built using a paravitualization design is that the calls directly to the VMM must remain secure. By providing direct calls into the VMM for guest operating systems to use instead of transferring control indirectly through hardware traps it is possible to expose a security hole. This also adds complexity to the VMMs thin interface to guest operating systems.

It should be noted that some computer architectures have hardware support for virtualization which makes it possible for a VMM that utilizes paravirtualization to run hosted operating systems without modification. Processors that support Intel Virtualization Technology [Int05c, Int05b] or the AMD Pacifica Technology [Adv05] are two examples of such architectures. XenSource is currently working on migrating Xen to Intel VT. Hardware support for virtualization has also been injected into the IBM PowerPC architecture in the form of IBMs Enterprise Hypervisor Binary Interface. IBM is working on migrating Xen to the PowerPC 970 processor which supports the hypervisor binary interface [BX06].

### 2.1.4 Software Hosted Virtualization Systems

While the Denali and Xen VMMs run directly on top of the hardware, there is another architecture which has the VMM run on top of a general purpose operating system. Depicted in figure 2.2, software hosted VMMs leverage the services of the general purpose operating system (the host) to simplify the process of providing virtual hardware abstractions to guest operating systems. The hosted VMM can use any virtualization technique to host guest operating systems (e.g. full virtualization or paravirtualization). UMLinux [KDC03] is an example of a software hosted VMM. In UMLinux, the guest OS and guest applications run in a single process, the guest machine process. The guest machine process communicates with the VMM process via shared memory and IPC. As mentioned earlier, the key benefit to using a hosted VMM is that you get to use the abstractions and services of the host OS to provide virtualization to the guest OS. For example, in UMLinux, the guest-machine process serves as the virtual CPU; host files serve as virtual I/O devices; host signals serve as virtual interrupts; etc.

The one area where hosted VMMs fall short is performance. There is extra overhead associated with using the host operating systems services and abstractions instead of working directly with the hardware [CDD$^+$04]. Also, hosted VMMs do not make very much sense in a security setting as they rely on the services of the host OS to maintain their integrity in order to provide correct virtualization.

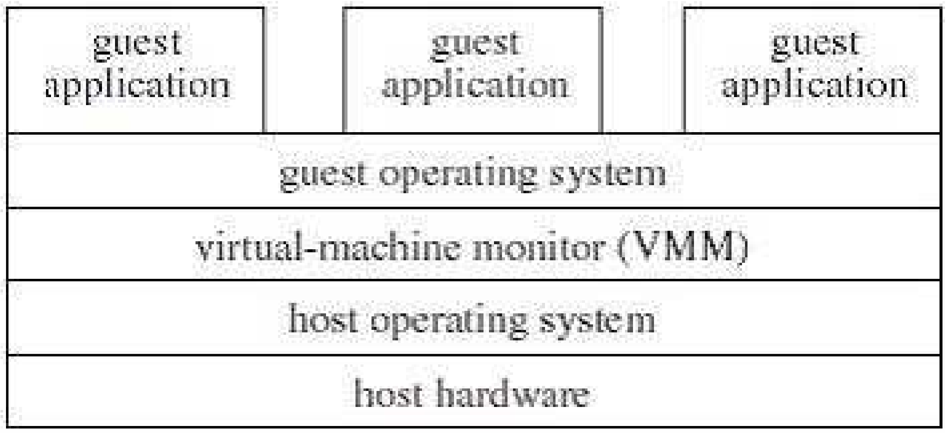


Figure 2.2: The organization of a system that uses a software hosted virtual machine monitor to support a single domain.

## 2.2 Taxonomy of Virtualization Techniques

We have divided up the VMMs into a taxonomy based on their features and requirements. Table 2.1 shows this taxonomy where the type is one of: NK (Nanokernel), FV (Full virtualization), or PV (Paravirtualization). Important aspects of VMM systems include the type of virtualization employed, how the VMM is hosted, whether it isolates the CPU and memory from hosted operating systems, whether virtualized devices are present, and how the VMM performs. Finally, it is important that we identify which VMMs are Open Source implementations and which are not. Choosing a VMM implementation to use as the basis for VMM-level research requires access to existing source code.

| VMM | Type | Host | CPU & Memory Isolation | Device Virtualization | Performance | Open source |
|---|---|---|---|---|---|---|
| RTLinux | NK | Linux Kernel | No | No | Very Good | Yes |
| VMware | FV | General purpose OS | Yes | Yes | Poor | No |
| UMLinux | PV | General purpose OS | Yes | Yes | Mediocre | Yes |
| Xen | PV | Hardware | Yes | Yes | Good | Yes |
| Denali | PV | Hardware | Yes | Yes | Good | No |

Table 2.1: A taxonomy of virtual machine monitor architectures.

The performance penalty imposed by certain VMM architectures cannot be ignored. Even as processor speeds increase, the performance penalties will still be noticeable. This is because the performance of a system running a VMM is greatly dependent on the performance of the memory subsystem. Even in a high performance paravirtualization environment such as Xen, memory performance becomes an issue because of the instruction cache, data cache, and TLB flushes that occur as a result of switching context from one domain to another. Additional overhead is even more important for handheld devices where every operation is paid for with battery power. A paravirtualization environment allows us to make more efficient use of the processor and memory subsystem on a handheld device, saving its most important resource: its battery.

Therefore, an open source paravirtualization architecture would be better than full virtualization alternatives on a mobile device. It would provide all of the isolation that is needed for a secure VMM while incurring minimal performance penalties and more efficient use of battery power. Currently, there are no hardware hosted paravirtualization-style VMMs available for the ARM architecture. A port of Xen from x86 to the ARM architecture would fill this void. Such a port would also be a good starting point for creating secure VMM-level security services.

# Chapter 3

# Discussion: Intel x86 vs. StrongARM

This chapter provides an explanation of the features provided by the x86 and StrongARM CPU's and how they relate to the paravirtualized interface presented by the Xen hypervisor.

The StrongARM is a RISC-style CPU [PD80] that originates from the embedded systems design space. As such, it was designed with small die size, low power consumption, and compact code density in mind. The origins of the x86 CPU design can be traced back to the 8086 processor introduced by Intel in 1978 and it's descendent the 8088, introduced in 1979 and used in the first personal computers by IBM[1]. All future members of the x86 family are backwards compatible with the 8086. The evolution of the x86 family of CPU's has been dominated by this insistence on backwards compatibility and features that make it more attractive as the basis for a multi-tasking personal computer system.

[1]Although the 8088 was designed later, it used an 8-bit wide data bus instead of the 16-bit bus used in the 8086 as a way of reducing cost

It is obvious that these two processors were developed with different goals in mind and this is reflected in the feature sets provided by each. However, there are also quite a few similarities between how systems based on these two CPU's can be paravirtualized.

## 3.1 Support for a secure VMM

First we examine the ability of the x86 and StrongARM CPUs to support a secure VMM. The key architectural features for supporting a VMM were outlined by Goldberg in [Gol72] as:

- two processor modes of operation.
- a method for non-privileged programs to call privileged system routines.
- a memory relocation or protection mechanism such as segmentation or paging.
- asynchronous interrupts to allow the I/O system to communicate with the CPU.

### 3.1.1 x86

The Pentium CPU has features that match each of the above requirements. It has four modes of operation known as rings that also represent privilege levels. Ring 0 being the most privileged, ring 3 the least privileged. The Pentium uses the *call gate* to control transfer of execution between privilege levels. It uses both paging and segmentation to implement protection. Finally, the Pentium uses both interrupts and exceptions to transfer control between the I/O system and the CPU.

Despite these features, it was shown by Robin and Irvine [RI00] that the Pentium instruction set contains sensitive, unprivileged instructions. Such instructions can be executed in unprivileged mode without generating an interrupt or exception. This opens the door for software running at a lower privilege levels (such as a hosted virtual machine) to undermine software running at a higher privilege level (such as a VMM) by reading or writing sensitive information. These instructions were placed into two categories:

- sensitive register instructions.
- protection system references.

Sensitive register instructions are those that read or modify sensitive registers and/or memory locations. By executing these instructions it would be possible for hosted virtual machines to undermine the VMM controlling the system. For example, the PUSHF and POPF instructions push and pop the lower 16 bits of the EFLAGS register to and from the stack. Pushing these values onto the stack effectively allows the EFLAGS register to be read. Popping these values off of the stack allows the EFLAGS register to be written. This prevents virtualization because the bits in the EFLAGS register control the operating mode and state of the processor.

Instructions that reference the storage protection system, memory, or address relocation system are also sensitive instructions. For example, the POP and PUSH instructions can be used to pop and push a general purpose register or a segment register to and from the stack. Pushing a segment register onto the stack effectively allows the segment register to be read. Popping a value off of the stack and into a segment register allows the segment register to be written. This prevents virtualization because the segment registers contain bits that control access to different memory locations.

There are also instructions that fail silently when executed in unprivileged mode. This prevents virtualization because the semantics of the instructions are different depending on the privilege level that the machine is executing in.

### 3.1.2 StrongARM

The StrongARM CPU has features that match the requirements needed to support a VMM as outlined above. It has two modes of operation: user mode and supervisor mode. The StrongARM controls transfer of execution from user mode to supervisor mode by using *software interrupts*. It uses paging to implement protection. Finally, the StrongARM uses both interrupts and exceptions to transfer control between the I/O system and the CPU.

In addition, the instruction set implemented by the StrongARM is extremely simple when compared to that of the Pentium. The StrongARM supports 32 instructions whereas the Pentium supports approximately 250 instructions. More importantly, none of the 32 StrongARM instructions can be classified as sensitive and unprivileged. Table 3.1 shows the ARM instruction set.

| Mnemonic | Instruction | Action |
|---|---|---|
| ADC | Add with carry | Rd := Rn + Op2 + Carry |
| ADD | Add | Rd := Rn + Op2 |
| AND | AND | Rd := Rn AND Op2 |
| B | Branch | R15 := address |
| BIC | Bit clear | Rd := Rn AND NOT Op2 |
| BL | Branch with link | R14 := R15, R15 := address |
| CMN | Compare negative | CPSR flags := Rn + Op2 |
| CMP | Compare | CPSR flags := Rn - Op2 |

|  |  |  |
|---|---|---|
| EOR | Exclusive OR | Rd := (Rn AND NOT Op2) OR (Op2 AND NOT Rn) |
| LDM | Load multiple registers | Stack manipulation (Pop) |
| LDR | Load register from memory | Rd := [address] |
| MLA | Multiply accumulate | Rd := (Rm + Rs) + Rn |
| MOV | Move register or constant | Rd := Op2 |
| MRS | Move PSR status/flags to register | Rd := PSR |
| MSR | Move register to PSR status/flags | PSR := Rn |
| MUL | Multiply | Rd := Rm * Rs |
| MVN | Move negated register | Rd := NOT Op2 |
| ORR | OR | Rd := Rn OR Op2 |
| RSB | Reverse subtract | Rd := Op2 - Rn |
| RSC | Reverse subtract with carry | Rd := Op2 - Rn - 1 + Carry |
| SBC | Subtract with carry | Rd := Rn - Op2 - 1 + Carry |
| STM | Store multiple registers | Stack manipulation (Push) |
| STR | Store register to memory | [address] := Rd |
| SUB | Subtract | Rd := Rn - Op2 |
| SWI | Software interrupt | OS call |
| SWP | Swap register with memory | Rd := [Rn], [Rn] := Rm |
| TEQ | Test bitwise equality | CPSR flags := Rn EOR Op2 |
| TST | Test bits | CPSR flags := Rn AND Op2 |
| CDP | Coproc data operations | CRn := (result of op) |

| | | |
|---|---|---|
| MRC | Move from coproc to ARM reg | Rd := CRn |
| MCR | Move from ARM reg to coproc | CRm := Rd |
| LDC | Load to coproc | CRn := [Rm] |
| STC | Store from coproc | [Rm] := CRn |

Table 3.1: The ARM instruction set.

All instructions that read or modify sensitive registers can only be executed in supervisor mode. For example, the MRS and MSR instructions that are used to read and modify the PSR (ARM Processor Status Register) can only be executed in supervisor mode. If either is attempted in user mode, an exception occurs. The PSR encodes the processor's operating mode, interrupt status, and state of condition code flags. Other instructions that modify sensitive registers are the Coprocessor access instructions such as MRC and MCR. These instructions move data to and from the ARM coprocessors which are used to control properties such as: the location of pagetables and the current access control privileges of the processor. Again, these instructions can only be executed in supervisor mode. It should also be noted that instructions such as CMP which modify the CPSR (Current Processor Status Register) are allowed to execute in both user or supervisor modes. This is safe because they only modify the condition code bits in the CPSR and not the mode or interrupt bits.

Finally, there are no instructions in the StrongARM instruction set that reference the storage protection system, memory or address relocation system. Taking these things into consideration it is clear that the ARM instruction set architecture is easier to virtualize than its x86 counterpart and a much better fit for use in a secure VMM.

## 3.2 Paravirtualized x86

The Xen hypervisor provides a secure VMM on the Pentium architecture despite the presence of sensitive, unprivileged instructions discussed above. Xen achieves this by providing a paravirtualized x86 interface to hosted virtual machines. This interface is similar to the regular x86 interface but different in that it works around the instructions and features in the architecture that cause problems for virtualization. Guest operating systems must be modified to run on top of Xen as a result of these work arounds.

### 3.2.1 CPU

#### Protection

CPU protection is virtualized by placing the hypervisor at the highest privilege level, while guest OSes and their applications run at lower levels. On x86 this is simple because there are four separate execution rings: Xen runs at ring 0, guest OSes run at ring 1, and applications run at ring 3. Guest OSes must be modified to run at ring 1, because they will no longer have access to all of the privileged instructions and will no longer be able to access sensitive, unprivileged instructions safely. In order to access privileged functionality a guest OS must ask the hypervisor to perform the instruction on its behalf via a hypercall.

### Exceptions

Exceptions are virtualized by requiring guest OSes to register a descriptor table for exception handlers with the hypervisor. For the most part, handlers may remain identical to their non-virtualized counterparts since Xen copies the exception stack frame onto the guest OS stack before passing it control. The only handler that must be modified is that which services page faults since it reads the faulting address from the CR2 register. The CR2 register is privileged and as such the guest OS will not be able to access it from ring 1. The solution used by Xen was to extend the stack frame to accommodate the contents of the CR2 register. The hypervisor reads the address from CR2 and pushes it on the stack; the modified page fault handler pops the address off of the stack.

Exception safety is guaranteed by the use of two techniques. First, the handlers are validated by the hypervisor at registration time. This validation checks that the handlers code segment does not specify an execution privilege reserved by the hypervisor. Second, the hypervisor performs checks during exception propagation to ensure that faults originate from outside of the exception virtualization code by inspecting the program counter on a subsequent fault. If the program counter contains an address inside the virtualization code then the guest OS will be killed.

### System Calls

System calls are typically implemented on x86 OSes using software exceptions. Xen improves their performance by allowing each guest OS to register a set of fast exception handlers. Fast handlers are executed directly by the processor without indirection through ring 0 and are validated by the hypervisor at registration time.

#### Interrupts

Hardware interrupts are replaced by a lightweight events delivery system. These allow for asynchronous notifications to be delivered from Xen to a guest OS. The hypervisor checks a per-domain bitmask to see whether there are pending events for a domain. If there are events pending, the bitmask is updated and a domain-specific event-callback can be executed. Events can be disabled by using a mask specified by the guest OS. This is similar to the way that interrupts can be disabled on a CPU.

#### Time

Each OS exposes a timer interface and is aware of 'real' and 'virtual' time. Time is available in granularities all the way down to the cycle count on the Pentium architecture.

### 3.2.2 Memory Management

#### Paging

Translation lookaside buffer (TLB) misses are serviced by the x86 processor by walking the pagetable structure in hardware. The TLB is not tagged with address-space identifiers (ASID) to associate mappings with the current addressing context. Therefore, the TLB must be flushed on every context switch. To avoid context switches when entering the hypervisor, Xen exists in the top 64MB segment of every address space. However, this mapping is not accessible by the guest OS because the segment is marked as accessible only by ring 0 (discussed above). A similar technique is used to avoid context switches for system calls in a standard OS without virtualization. Also, guest OSes allocate and manage the hardware page tables as usual with the restriction that all pagetable writes are validated by the hypervisor. Xen amortizes the cost of this validation by allowing guest OSes to batch pagetable updates.

Validation requires that guest OSes only map pages that they own and do not allow for writable mappings of pagetables.

#### Segmentation

Segmentation is virtualized in a similar way by validating updates to the segment descriptor tables.

Validation requires that the updates have lower privilege than the hypervisor and they do not allow access to the top 64MB of the address space where Xen resides.

### 3.2.3 Device I/O

In order to support asynchronous device I/O, Xen makes use of the lightweight events described above to notify guest OSes of device status. In addition, data is transferred via shared-memory in asynchronous buffer rings. This abstraction makes it possible to efficiently move data from the hypervisor to guest OS's. It also provides a level of security because it allows the hypervisor to validate certain properties of the data transfer for safety.

## 3.3 Paravirtualized StrongARM

When porting the Xen hypervisor to the StrongARM architecture it was beneficial to adopt a scheme similar to that used in Xen x86. We found that this was easiest in many cases because of the similarities of certain features in the StrongARM and Pentium architectures. Sometimes paravirtualization was needed to enforce safety. In other cases, paravirtualization was not needed for safety, but we opted to paravirtualize anyway as a way of improving the performance of the system.

### 3.3.1 CPU

#### Protection

CPU protection is virtualized by placing the hypervisor at the highest privilege level, while guest OSes and their applications run at the lower level. On ARM CPU protection is not simple because there are only two execution modes: user and supervisor. Therefore, Xen runs in supervisor mode while guest OSes and their applications both run in user mode. Not only does this require that guest OSes be modified to run at a lower privilege level (same as in Xen x86), but it also requires that control be passed from guest OS to applications indirectly through the hypervisor because the guest OS must protect itself from applications by living in a separate address space. The hypervisor will then be responsible for switching address spaces and maintaining the virtual privilege level of the executing entity, be it guest OS or application.

The hypervisor protects itself from guest OSes in a similar way on the x86/64 architecture. The lack of segment limit support on x86/64 makes it necessary to protect the hypervisor using page-level protection.

#### Exceptions

Exceptions can be paravirtualized in exactly the same way they are in Xen on the x86 processor. In addition, none of the exception handlers in the guest OS need to be modified because they do not access any sensitive registers to gain fault status information. This includes page faults.

Exception safety is maintained during exception propagation in a manner identical to the fixup's performed by Xen on the x86 processor. In addition, there is no need to validate exception handlers at registration time because a handler cannot specify execution at a more privileged level without causing an exception.

### System Calls

System calls cannot be paravirtualized in exactly the same way they are in Xen on the x86 processor because of the way protection is implemented at the page level (discussed in the memory management section below). More specifically, because application pagetables do not contain mappings for the guest OS, the application must call into guest OSes indirectly via the hypervisor. The hypervisor then performs a context switch to change the virtual address space.

Again, fast application system calls are implemented in a similar way on Xen for the x86/64 architecture. On both ARM and x86/64 the context switch requires a TLB flush. However, it should be noted that TLB flushes on the x86/64 can be avoided using the AMD64s TLB flush filtering capabilities [KMAC03]. Avoiding the TLB flush improves the performance of system calls dramatically.

On StrongARM CPUs there is a feature of the architecture that can save flushing the TLB on every context switch between applications and guest OSes. This feature is called Domain Access Control and it is discussed later in this chapter.

### Interrupts

Interrupts can be paravirtualized in exactly the same way they are in Xen on the x86 processor, by replacing them with the lightweight event notification system.

### Time

Time can be paravirtualized in the same way that it is in Xen on the x86 processor. The major difference is in the granularity of timing available. For example, some StrongARM systems do not provide a cycle counter. One of these is the SA-110 based system known as the DNARD Shark [Dig97] which only provides a real-time clock. As a result, OSes that rely on access to cycle counter for important computation will encounter problems. One such OS is the Linux kernel, which makes use of the cycle counter on the Pentium architecture to maintain *jiffies* used in the scheduler and other places as a timing unit. One solution is to simulate a cycle counter in the hypervisor on systems that don't support one in hardware.

However, more recent versions of the StrongARM CPU have become increasingly more integrated and support a wider variety of features. An example of such a newer revision is the Intel 80200 [Int03] which supports a cycle counter.

## 3.3.2 Memory Management

### Paging

The ARM architecture features virtually-addressed L1 instruction and data caches[2]. Unfortunately, entries in the ARM's TLB are not tagged with an ASID. To compound matters, TLB misses are serviced by the walking the pagetable structure in hardware similar to what is done on the x86. As a result, the TLB must be flushed on a context switch. In addition, data in the caches may be stale after a context switch. As a result, unless the kernel is certain that no stale data exists in the

[2]L1 caches on the ARM are virtually-indexed and virtually-tagged, but do not make use of ASIDs to differentiate between memory from different address spaces.

caches, they must be flushed on context switch as well[3].

The ARM CPU's paging is so similar to that of the x86 that we opted to use the same paravirtualization techniques in the ARM port. Namely, the hypervisor exists in the top 64MB of every address space. However, because the ARM does not support segmentation, we needed to use page-level protection to isolate the different memory regions in the system. There are two different cases for mapping the different memory regions in the system.

The first memory mapping strategy handles mapping the hypervisor region. Here, the top 64MB of every address space is mapped with read and write access in supervisor mode only. The fact that the hypervisor is co-mapped with guest OSes means that switching between a guest OS and the hypervisor can occur without a page table switch. The result is low latency performance for common operations such as the delivery of events and the execution of hypercalls.

The second memory mapping strategy handles mapping guest operating systems and their applications. A guest OS and its applications both share user mode, however the OS must remain isolated from the applications it hosts. To achieve this the guest OSes pagetables map memory belonging to the guest OS and its applications with read and write access in user mode. The applications own their own set of pagetables that map the memory that they own with read and write access in user mode. However, the application pagetables do not co-map the guest OS memory region. With this organization, the guest OS can access and modify the memory of the applications that it hosts, but the hosted applications cannot see or access the guest OS. As a result, a context switch must occur when execution transfers from guest OS to application or vice versa. The context switch has a negative impact

[3]The ARM Linux kernel flushes the TLB and I and D caches whenever a context switch occurs

on the performance of instructions that cause applications to trap into the guest OS because they can only enter the guest OS indirectly after the hypervisor has performed the appropriate page table switch. Despite the performance issues, this approach is similar to what has been adopted for Xen on the x86/64 architecture.

Pagetable writes done by a guest OS must be validated by the hypervisor for the same reasons as in x86. In addition, guest OSes may only map pages that they own and do not allow for writable mappings of pagetables.

### ARM Domains

As described above, the ARM uses a two-level hardware-walked TLB. Each entry is tagged with a four-bit *domain ID*. The *domain access control register* (DACR) can modify the access rights specified in TLB entries to have either:

- access as specified in the TLB entry.
- no access at all.
- full page access (regardless of what the TLB protection bits specify).

It was shown in [WH00] that it is possible to use ARM domains as a way of allowing mappings from different address spaces to co-exist in the TLB and caches as long as the mapped address spaces do not overlap. If two address spaces overlap, then after a context switch an access might be attempted to a page that is mapped with a domain ID that belongs to another process. Such an access will generate a fault which is then handled by the kernel by flushing the TLB and caches before updating the fast address space switching (FASS) paging data structure and continuing execution. After the flushes, only address spaces that do not overlap will co-exist in the TLB and caches until another conflict occurs.

In the FASS scheme, one of the 16 domain IDs is allocated to each process, in effect becoming an ASID. On a context switch the TLB and caches are not flushed. Instead, only the DACR needs to be modified to grant access to only the newly executing processes pages and deny access to all others. This is done by reloading the DACR with a mask that allows access only to pages tagged with the domain id associated with the newly running process.

Further work on FASS for the StrongARM platform [WTUH03] showed that the above technique is used to create what is almost equivalent to software managed TLB for the ARM processor. However, the scheme is limited because a domain ID is more restricted in the values it can take when compared to a classical ASID. This also leads to boundary conditions in the implementation where domain IDs must be reclaimed from mappings belonging to non-executing processes; domain IDs must be recycled. Despite these drawbacks, it was determined that the TLB and caches would only need to be flushed when:

- there exist mappings for two different address spaces that overlap.
- the system runs out of domain IDs and must recycle one that was previously used.

A similar FASS technique would not be useful at the hypervisor level because of the requirement for non-overlapping address-spaces. Switching between the hypervisor and a guest OS already avoids a context switch due to the fact that Xen is mapped to the top 64MB of every address-space. Switching between two guest OSes will require a TLB flush even when using the FASS technique unless their address spaces do not overlap. Non-overlapping address spaces are not likely when executing different versions of the same OS or even different OSes of the same family[4].

As a result, it is required that a pagetable switch and the associated TLB flush takes place whenever the hypervisor switches between domains. However, the work on FASS shows that making use of domains is useful when context switching between an OS and it's applications. We could make use of the same technique in Xen to avoid a full pagetable switch when moving from an application to a guest OS and vice versa. Instead, when such a context switch occurs we would reload the DACR with a mask that permits access to the appropriate set of pages.

**StrongARM PID Relocation**

The StrongARM architecture provides a way for small address spaces -those containing addresses less than 32MB- to be relocated to another 32MB partition in a transparent manner. Which partition they are relocated to is determined by the *PID register*. There are 64 partitions available. The use of PID relocation in combination with ARM domains was used in FASS to reduce the number of address space collisions. This was effective because whenever an address space collision occurred, the TLB had to be flushed.

However, a similar PID relocation technique would not be useful in the hypervisor across domains because many of the virtual addresses mapped for the guest OSes are too large to be relocatable; they fall outside of the lowest 32MB region. However, we could apply the FASS work to relocate the applications hosted by guest OSes to reduce the number of address space collisions and associated TLB flushes.

Applying the FASS techniques that utilize ARM domains and PID relocation to manage a domains pagetables could potentially be a big win for performance

[4]Linux, FreeBSD, and NetBSD are all Unix-like, and would likely produce many address-space collisions

because of the page-level protection needed. However, we did not implement such a scheme and it could be considered useful future work.

### 3.3.3 Device I/O

Devices would best be paravirtualized in exactly the same way they are in Xen on the x86 processor.

## 3.4 Summary

Systems powered by the ARM architecture lend themselves well to virtualization. When compared to the x86 ISA, the ARM ISA is much easier to virtualize. In addition, the features of the ARM architecture are a nice fit for Xen-style paravirtualization.

There are a few key differences in the x86 and ARM feature sets that require changes to the paravirtualization interface of the hypervisor. For instance, protecting the hypervisor from guest OSes must be done using page-level protection because the ARM CPU does not support segmentation. Protecting guest OSes from the applications they host must also be done using paging, but is complicated by the fact that there are only two modes of operation on ARM CPUs compared to the four that are available on x86 hardware.

There are also similarities between the two architectures that allow reuse of the x86 paravirtualization interface with little or no change. Paging is one such feature because the TLBs on both the ARM and x86 are walked by hardware. For this reason we adopt the technique of validating guest OS pagetable updates in the hypervisor. Other parts of the paravirtualization interface that can be reused with little or no change are interrupt virtualization via lightweight events and device virtualization using events for notifications and asynchronous data rings for the transfer of data.

# Chapter 4

# Design and Implementation

Porting the Xen hypervisor from the x86 to the StrongARM architecture was difficult for a few reasons. Since Xen is a VMM that sits directly on top of the hardware there is some architecture specific code for bootstrapping, hardware interrupt handling, device management, and low level domain handling that had to be ported to the StrongARM. There is also some architecture independent code that contained assumptions about the underlying architecture that required changes.

We have produced a version of the Xen hypervisor capable of loading a test operating system as Domain 0, servicing hypercalls from the domain, and delivering events to the domain. This chapter details the key issues and implementation decisions that were made in porting the Xen hypervisor to the StrongARM platform. Our prototype implementation is based on the Xen 1.2 codebase and it runs on the Digital Network Appliance Reference Design (DNARD)[1].

The DNARD makes for a good reference platform because of its simplicity and the fact that its CPU and memory specifications are comparable to that found in many low-power handheld devices. It utilizes the first generation StrongARM

[1]The DNARD units are also commonly referred to as "Sharks".

processor, the SA-110 CPU running at 233 MHz and has access to 32 MB of SDRAM.

Our choice to use the Xen 1.2 codebase was born out of necessity. Since Xen 1.2 was released in 2003, activity surrounding its development has been intense. The current stable releases are Xen 2.0.7 and Xen 3.0. While it would have been desirable to work with the current Xen 3.0 codebase this was not possible due to compiler compatibility restrictions.

The Xen team is moving away from GCC versions 2.95.x; Xen 2.0 and 3.0 build under GCC versions 3.3.x, 3.4.x, 4.0 only. However, the ARM cross compiler of choice in the ARM Linux [arm05b] kernel community is GCC version 2.95.3. This is because the ARM code generation in the GCC 3.x series of compilers is less than perfect. In fact, we verified this by building various versions of the GCC toolchain for the ARM platform. To automate the task we used the *crosstool* toolchain generation script [Keg05]. Using the generated toolchains we attempted to build Linux for the DNARD Shark [sha05]. We confirmed that only GCC version 2.95.3 built the Shark Linux kernel without issue. More recent GCC versions failed in nefarious ways. GCC 3.4.3 came closest to building a complete kernel but generated a segmentation fault during the last stage of the build; compiling a portion of the Shark Linux boot code that interfaces with the DNARD firmware. Since we planned to use this code to bootstrap our prototype hypervisor, the decision was made to use the version of GCC which could correctly compile it: GCC 2.95.3. Thus, we were restricted to the only version of Xen that builds under GCC 2.95.3: Xen 1.2.

Despite the fact that we are basing our port on a version of the hypervisor that is nearly three years old, our work is still relevant since it should not require major changes to pull into the Xen 2.0/3.0 world. This is because the changes to Xen from 1.2 to 2.0.x focus mainly around a redesign of the I/O architecture which will not have a large affect on our work. Similarly, the changes from Xen 2.0 to 3.0 focus on support for other architectures in the hypervisor (i.e. x86/64 and IA64), SMP capable Guest OSes, and running unmodified guest OSes via the Intel VT-x and AMD Pacifica extensions [PFH+05]. The feature which has the potential to affect our work the most is support for SMP capable Guest OSes.

The following sections describe the details of our port. Section 4.1 details the changes necessary to architecture specific portions of the code. Section 4.2 details the changes necessary to the paravirtualization architecture.

## 4.1 Differences in architecture specific code

There were many bits of architecture specific code that needed to be ported in order to have a functioning hypervisor. Much of this was written in x86 assembly and needed to be ported to the equivalent in ARM assembly.

### 4.1.1 Hypervisor boot code

The Xen boot code is quite simple for x86. It is 266 lines of assembly (LOC) in `xen/arch/i386/boot/boot.S`. It requires the GRUB boot loader [gru05] to load the hypervisor image file into the upper 64MB segment of virtual memory and additionally load any modules specified as arguments. Once loaded the hypervisor boot code (written in x86 assembly) is free to perform CPU type checks, perform CPU initialization, setup the initial pagetable entries, start paging, initialize the BSS, copy all of the modules to a well known address, and then jump to the *cmain* entry point. The only wrinkle is that the hypervisor's image file must be loaded into the upper 64MB region of virtual memory; a problem because the boot loader cannot access to the high addresses directly. To work around this, the image file is modified after it is compiled by a tool named *elf-reloc*. This tool rewrites the segment offsets in the code to allow for the load to succeed.

On the DNARD, things are slightly more complicated as GRUB is not available and the ARM architecture does not support segmentation. Therefore, to load the hypervisor in to the upper 64MB region of memory we needed to adopt a technique used in the Linux kernel. We wrap the compressed kernel image with a low memory bootstrapper that can relocate the kernel image to the upper 64MB region of memory. At boot time, the following steps are taken:

1. DNARD uses TFTP to grab the image file.

2. open firmware loads the image into memory and jumps to the start of the low memory boot loader

3. the low memory boot loader unzips the compressed kernel image to the upper 64 MB region and jumps to the start of the relocated image.

4. the hypervisor boot code takes control and initializes the hypervisor.

The hypervisor boot code on the Shark performs initialization similar to that on the x86 with the exception of processing kernel modules. Loading modules is important because the first module specified will be the guest OS to run on top of Xen as domain 0. In Xen/x86, modules are loaded into memory by GRUB and copied by the Xen boot code to a well known address where they can then be accessed by the hypervisor. On the DNARD, the boot loader does not have the ability to load modules, hence the hypervisor boot code would have to be modified to do so. To complicate matters the Sharks have no fixed storage, so modules have to be obtained from network accessible storage. We had the following choices for how to gain access to a module on the DNARD.

- Obtain modules the same way that diskless systems do when running Linux
- Link the modules directly into the Xen binary image

Diskless systems that run Linux obtain modules by mounting a network accessible device as root using NFS. We could do something similar in Xen, but we would have to import all of the NFS functionality into the Hypervisor.

To simplify the implementation we chose to link the modules into the Xen kernel image file as data. Thus, once booted the hypervisor reads the module information out of the image file without needing to go back to the network.

The hypervisor boot code for the DNARD spans two files. The low memory boot loader is found in `xen/arch/arm/boot/compressed/head-shark.S` and is 243 LOC. It was taken almost directly from the Linux Kernel implementation with small modifications. The main boot code for the hypervisor is found in `xen/arch/arm/boot/head-armv.S` and is 603 LOC. A majority of the bulk comes from the hard coded page table setup of the direct mapped region, which is discussed in the following section.

### 4.1.2 Hypervisor direct mapped region

Once the hypervisor hypervisor boot code has initialized the processor, it creates initial mappings for the hypervisor's direct mapped region in the level 1 pagetable. After the mappings are created the boot code can turn on paging and jump to the *cmain* entry point to continue start of day processing.

Each of the areas in the direct mapped region are mapped into physical memory as shown in Figure 4.1. Note that the DNARD's physical memory is divided into four memory banks, each 8MB in size. Also note that the physical addresses are

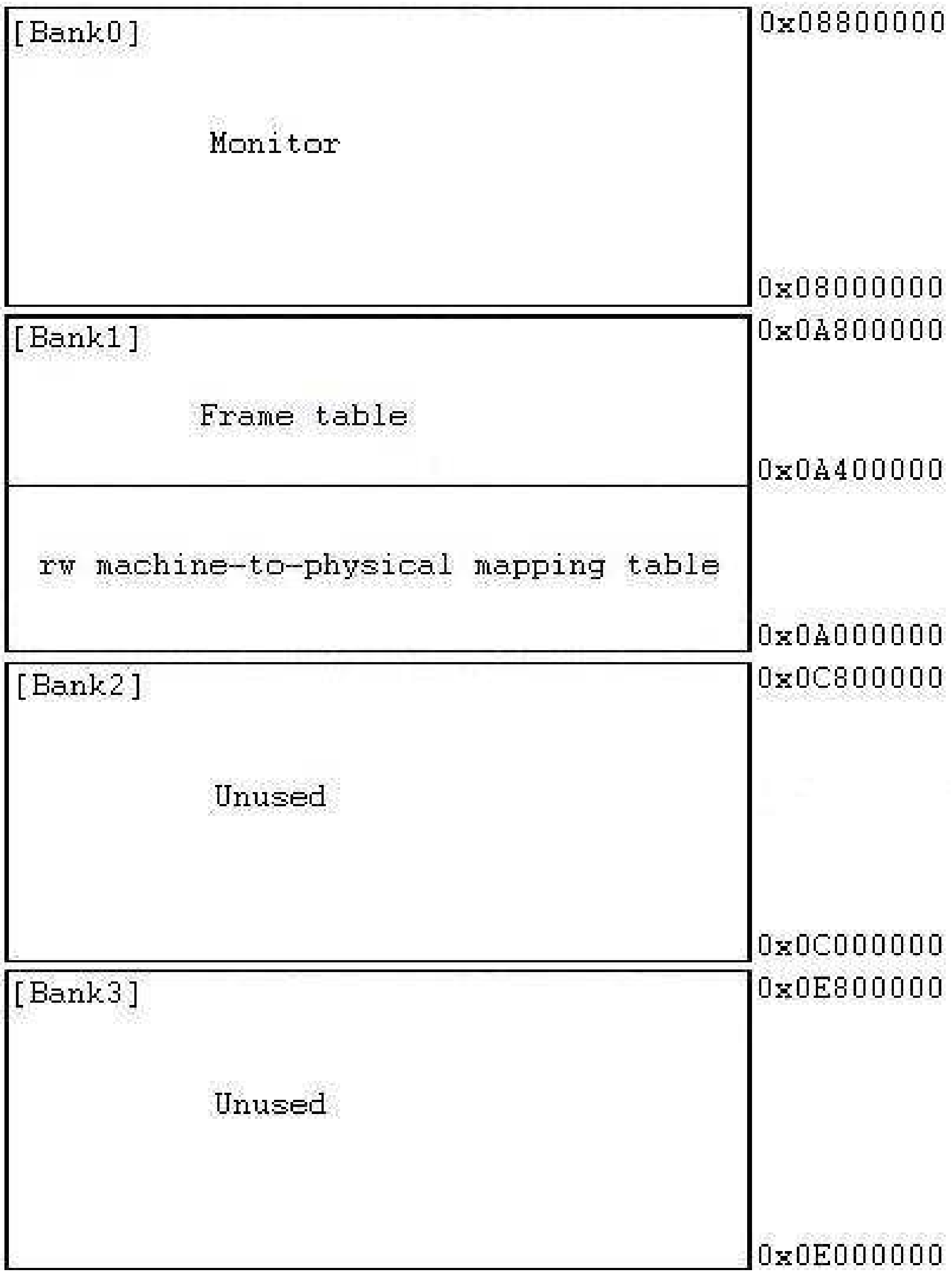


Figure 4.1: Memory mappings for the direct mapped region of the hypervisor.

non-contiguous. The virtual addresses for the different regions are shown in Figure 4.2

The direct mapped region consumes 16MB of physical memory. The first part of the region is an 8MB slice that is consumed by the monitor's code, data, and dynamic data structures. This is followed by 4MB for the read-write machine-to-physical mapping table. The final part of the direct mapped region is dedicated to 4MB for the frame table. It should be noted that all of these mappings are mapped with read-write access in supervisor mode and no-access in user mode, providing protection from user space processes.

This memory layout leaves 16MB of free physical memory on the device for use by domains. However, it should be possible to optimize this memory layout even further. For example, the frame table is used to track the allocation of each machine page being used by VMs in the system. For mobile devices supporting small amounts of physical memory, the number of machine pages available will be small. Thus, we could reduce the size of the frame table accordingly (see Chapter 5 for an evaluation of how much smaller we could make the frame table on the Shark).

There were some restrictions on how we chose these memory mappings. For example, the hypervisor's page allocator claims some of the physical memory allocated to the bottom of the monitor region. The page allocator assumes that the physical pages it manages are contiguous in memory. Thus, to be safe, we make sure the monitor region does not span multiple memory banks.

In fact, the assumption that physical memory addresses are contiguous is present in more than one place in the hypervisor code.

### 4.1.3 Idle task

After the hypervisor has initialized the direct mapped memory region, turned on paging and jumped to the *cmain* entry point it initializes its data structures under the guise of the *idle_task*.

The hypervisor uses a task to represent each of the domains that it can execute. It also keeps the *idle_task* as something that it can run when no other domain is capable of running. Each task is associated with its own pagetables and its own stack among other things. Thus, a task has its own address space and its own thread of execution.

### 4.1.4 Frame table

The first data structure that the hypervisor initializes is the frame table. It is used to track the allocation and usage of each machine page being used by VMs in the system.

The discontiguous physical memory of the DNARD (described above) caused the x86 frame table initialization and management code to break. Essentially, the frame table is a linked list of pfn_info nodes. Each node in the list corresponds to a physical page of memory and contains various information about how the physical page is being used by a domain. In Xen/x86 it is assumed that the frame table follows directly behind the monitor and machine-to-physical mapping table sections in physical memory. In addition, it is assumed that the physical memory region following the frame table -that to which the frame table nodes correspond- is also contiguous. Thus, the issue is that the list of frames can only track the use of a contiguous range of physical memory. We patched this up by hard coding the frame table to allocate memory from physical memory bank 2. Thus, whenever a physical page is allocated from the table we can determine the physical address by performing the following calculation.

```
phys_addr = frame_num * page_size + phys_offset
```

Where 'page_size' is 4096 (the size of one page) and 'phys_offset' is 0xC0000000 (the starting address of physical memory bank 2). The downside of this workaround is that the last 8MB of memory is inaccessible by the system (guest OSes included). However, 8MB is more than enough for us to load a simple test OS as a proof of concept. A description of how the frame table could be modified to support discontiguous physical memory can be found in the discussion of future work in Section 6.2. Our modifications to the frame table amount to 5 LOC being added to `xen/common/domain.c`.

### 4.1.5 Hypervisor address space layout

During hypervisor initialization the memory mappings for the rest of the hypervisor's virtual address space are added to the hypervisor's page table in the *paging_init* function. The address space needed to be modified to make use of the limited memory available on the DNARD. The layout implemented is shown in Figure 4.2. The read only machine-to-physical mapping table occupies the first 4MB of virtual space. It is mapped with supervisor read-write and user read-only permissions to the same chunk of physical memory as the read-write machine-to-physical mapping table. The direct mapped region follows and is as we specified earlier. The next part of the address space is used for linear page table mapping. It is nothing more than an alias from which to access the hypervisor's L1 pagetable. The next 1MB section is used for per domain mappings and is only used by domains, thus the hypervisor leaves the mapping for this region empty. The penultimate 1MB is used for temporarily mapping a physical page allocated to a domain into the hypervisor's address space and is dubbed the map cache. The memory that this region maps to is allocated by the hypervisor's page allocator. The final 1MB of the address space is reserved for ioremap(). Again, the memory that this region maps to is allocated by the hypervisor's page allocator. As such it resides somewhere inside the range of physical addresses in the monitor direct mapped region managed by the page allocator.

### 4.1.6 Trap handling

After finalizing its address space the hypervisor begins initializing other data structures. The first of these is the hardware trap table. The physical memory for the table is allocated by the hypervisor's page allocator and mapped to virtual address 0x0. On the SA-110, the trap table must be located at 0x0 and is not relocatable[2]. The trap table contains entries for handlers for undefined instructions, software interrupts (used for hypercalls), hardware interrupts (IRQs), fast hardware interrupts (FIQs), prefetch aborts, data aborts, and address exceptions. We use the ARM Linux code to create the trap table but modify the entries to point to our own handlers.

The trap table entry for software interrupts points to the code that handles hypercalls. The entry that handles prefetch aborts deals with faults when fetching code while the entry that handles data aborts deals with faults when accessing data. The undefined instruction abort is executed when the processor attempts to

[2]More recent versions of the StrongARM support trap table relocation; the trap table is not restricted to living at virtual address 0x0.

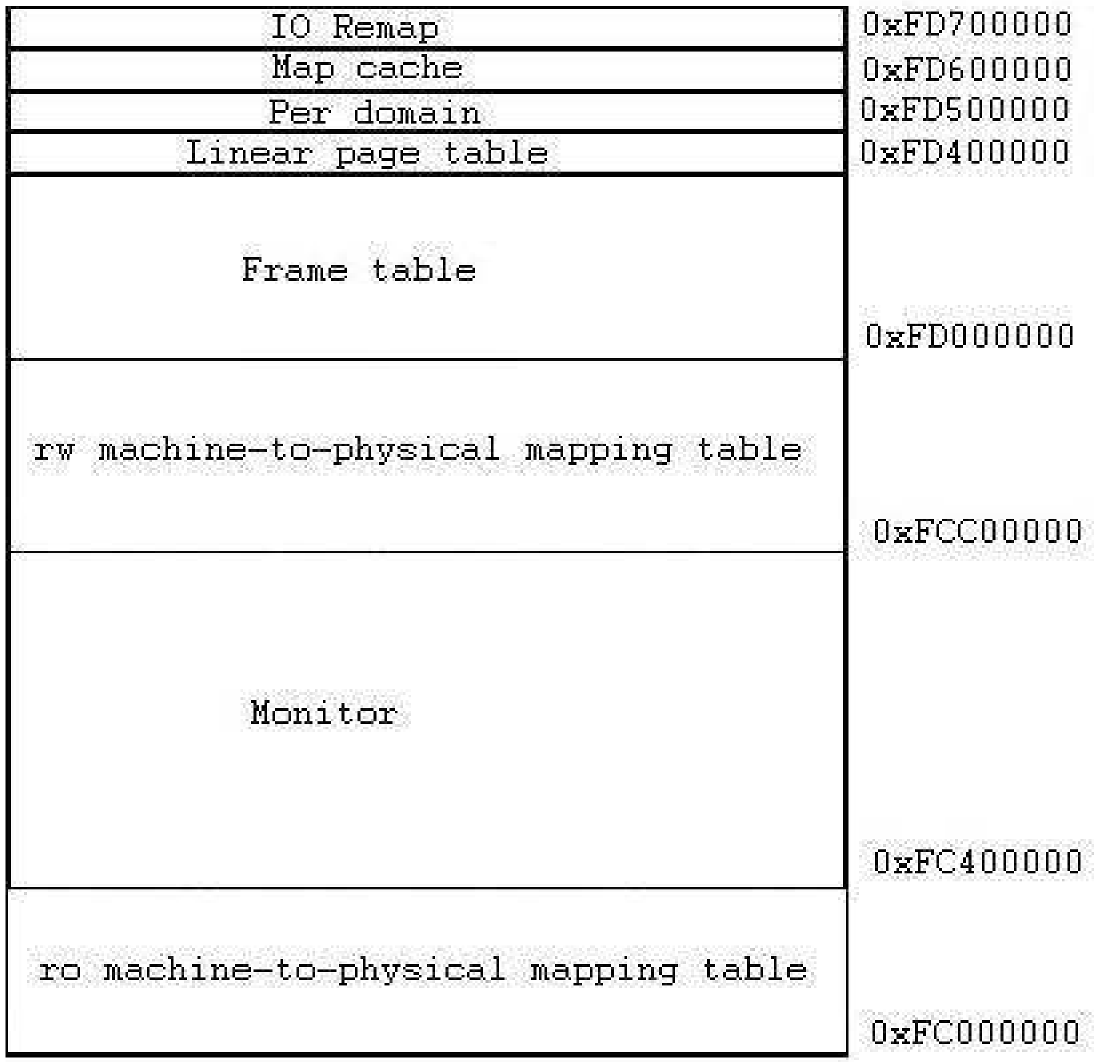


Figure 4.2: Hypervisor's virtual memory layout in the upper 64MB region.

execute part of memory that does not contain a valid ARM instruction. In each of the handlers we can determine whether the fault occurred in the hypervisor or in a domain by checking the mode bits in the saved CPSR (current processor status register) and the address in the PC (program counter)[3]. Once this is done we can take the appropriate action: forwarding the fault to the offending guest OS in the form of an event, terminating the guest OS, or executing a panic in the hypervisor.

The trap table entries that do not get used are the ones for FIQs and address exceptions. The hypervisor does not install any FIQ handlers and address exceptions never occur in 32-bit operating mode; they are a remnant of the old 26-bit operating mode.

The x86 implementation of trap handling and initialization is 822 LOC in the `xen/arch/i386/traps.c` file which is quite large due to the a fair amount of inline assembly code. The ARM implementation of trap handling is 361 LOC in `xen/arch/arm/traps.c` with all of the assembly pushed into `xen/arch/arm/entry-armv.S` which is 712 LOC.

### 4.1.7 Real time clock

The real time clock is the DNARD's only method of measuring the passing of time and the only device that our prototype makes use of on the system. We use code from the ARM Linux RTC driver and the code needed to install the interrupt handler for the device on IRQ 8 to initialize the device. After initialization, we program the real time clock to tick at a frequency of 128Hz, or once every 7.8ms. The RTC tick handler does the following:

- reads the RTC's interrupt stats register to acknowledge the interrupt.

[3]Xen/x86 determines whether the fault occurred inside the hypervisor by checking the CS (code segment) register.

- updates the hypervisor's notion of wall-clock time.
- updates the system time (number of milliseconds since boot).
- raises a softirq corresponding to the accurate timers.

We discuss why raising the softirq is necessary in the following section.

The clock initialization and clock tick handler are found in `xen/arch/arm/time.c` which is 441 LOC. Time management code on x86 can be found in `xen/arch/i386/time.c` and is 394 LOC.

### 4.1.8 Scheduler

The functionality of the hypervisor scheduler depends on the use of accurate timers which in turn depend upon the use of the APIC timer on x86 systems. However, the DNARD does not have an APIC timer. The solution we implemented was to periodically fire the softirq corresponding to the accurate timers in an effort to poll for its expiration time.

As mentioned above, we use the real time clock hardware interrupt handler to post the softirq for the timer. One wrinkle with raising the softirq from the interrupt handler is that it uses the architecture dependent and atomic set_bit() routine to set the corresponding flag in the hypervisor's softirq status word. On x86, set_bit() uses the BTSL instruction to atomically set the bit. There is no corresponding instruction on ARM. On ARM, set_bit() must load the required word from memory to a register, set the bit in the register, then store the result back to memory. To make this routine atomic, interrupts would need to be disabled at the beginning and re-enabled at the end of the routine. The code which disables interrupts assumes that the processor is in supervisor mode. This assumption is false in the real time clock tick handler because the processor is in IRQ mode when handling an interrupt. Thus, when interrupts are re-enabled at the end of set_bit(), the processor will no longer be in the correct mode. Things get worse when the interrupt handler exits and the system attempts to switch back to where it was interrupted assuming that it is currently in IRQ mode. Chaos ensues.

To solve the problems that using the architecture specific set_bit() routine creates we simply set the bit with regular C code. This does not introduce race conditions since irqs are already disabled when executing the handler. As a rule of thumb, interrupt handlers should not want or need to play with the CPSR to change the mode of the processor. The only thing that can preempt an IRQ on an ARM system is an FIQ (fast interrupt).

### 4.1.9 Task Management code

Once traps, timers, and the scheduler are initialized the hypervisor prepares to load a guest OS into domain 0. Before we can describe how this is done we must first understand how the hypervisor manages its tasks.

In Xen, a task corresponds to a domain. The task management code in the hypervisor is used to manage the state of the currently executing task. This state is contained on the hypervisor's stack and is accessed by the scheduler during context switches and the hypervisor entry code during hypercalls and event delivery. Both of these pieces of code are performance critical and must be as efficient as possible. Thus, the task management routines are written in assembly language and had to be ported to run on the ARM. Figure 4.3 shows the layout of the hypervisor's stack. The address of the structure describing the state of the current task is stored on the top. Below this, the state of the currently executing task is saved. As the hypervisor

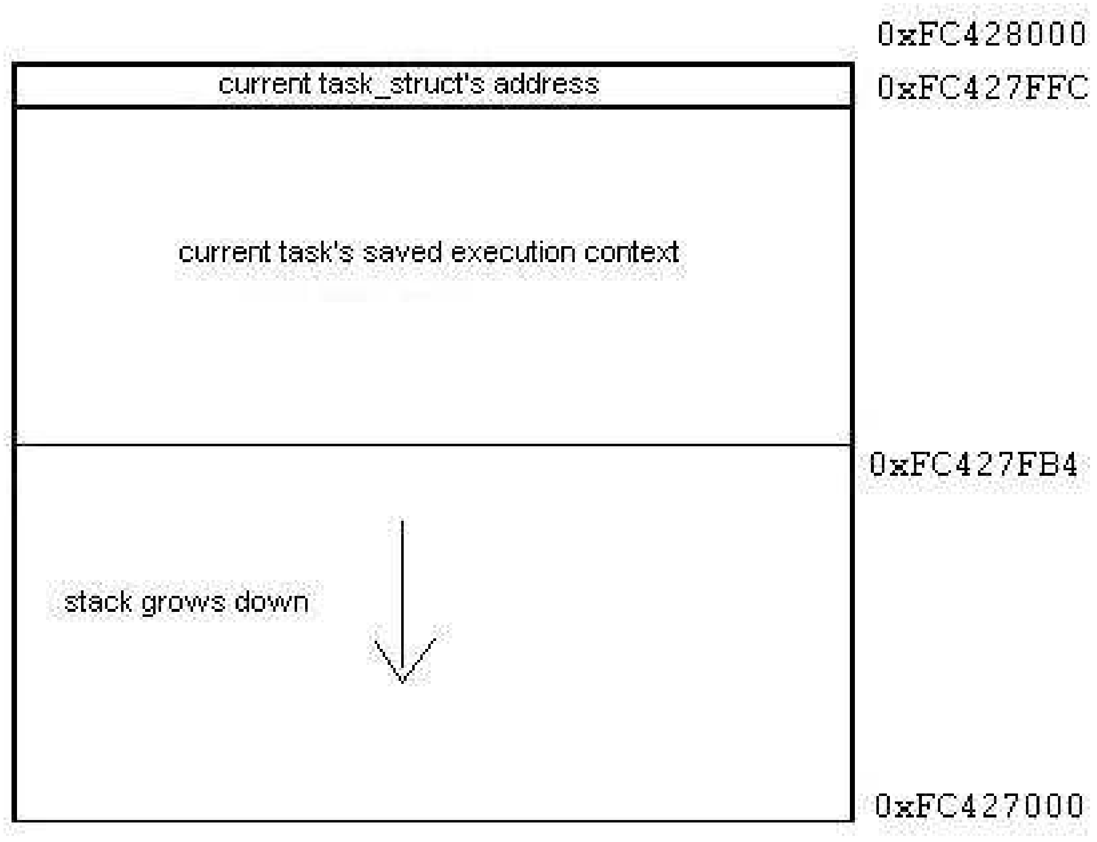

0xFC428000
current task_struct's address
0xFC427FFC
current task's saved execution context
0xFC427FB4
stack grows down
0xFC427000

Figure 4.3: Memory layout of the hypervisor's stack.

executes, the stack grows down.

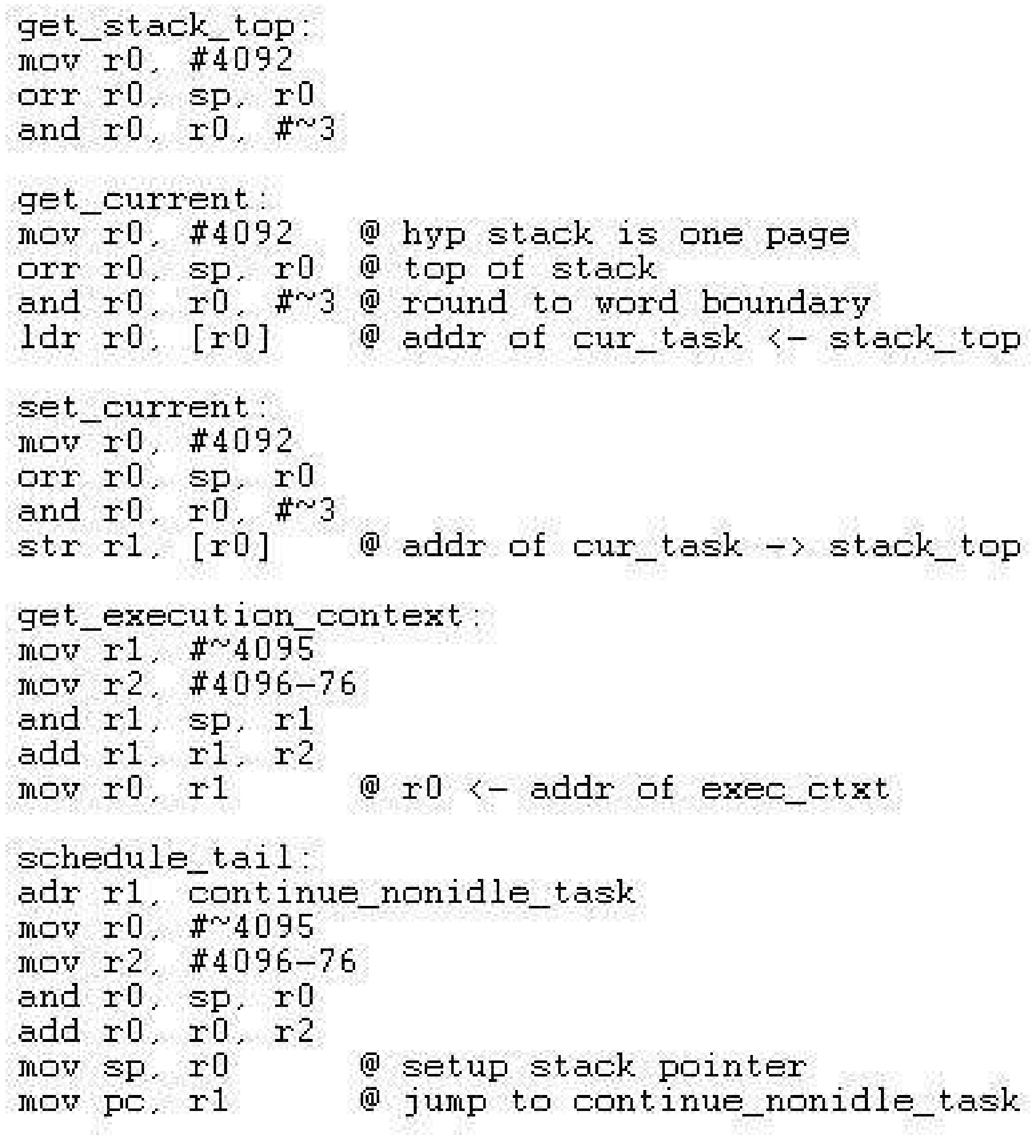
The five routines that manage this information are written in ARM assembly and are shown in Figure 4.4. Converting these from their x86 equivalents was straightforward. The only difference was that the execution context for a task in Xen/ARM contained one register more than in Xen/x86. Thus, the size of the execution context grew by four bytes to account for the extra word of information.

It should also be noted that these routines are presented in a simplified form. For example, the *schedule_tail* routine has the capability of continuing either a non-idle task (i.e., a domain), or the idle_task if there is no domain that can be executed. The task management routines for ARM are found in `include/asm-arm/current.h` which is 89 LOC. The corresponding routines for x86 are found in `include/asm-i386/current.h` which is 46 LOC.

### 4.1.10 Domain Execution Context Definition

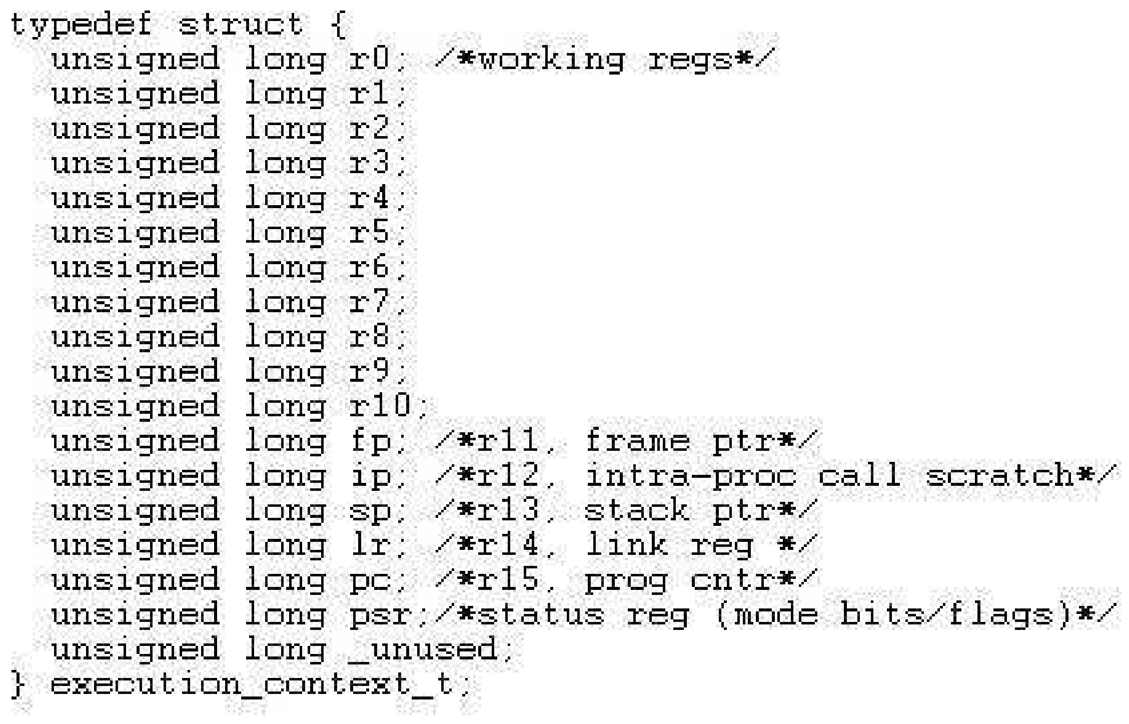
The execution context definition is architecture dependent because it depends upon the register set available on the system being paravirtualized. The execution context for ARM is shown in Figure 4.5. The field *_unused* is used internally by the hypervisor during the handling of hypercalls; there is an equivalent field in the x86 execution context definition. As stated earlier, the execution context for the ARM processor contains one more register than the execution context of the x86 processor.

The execution context definition is found in `include/hypervisor-ifs/hypervisor-if.h` where roughly 20 LOC were modified.



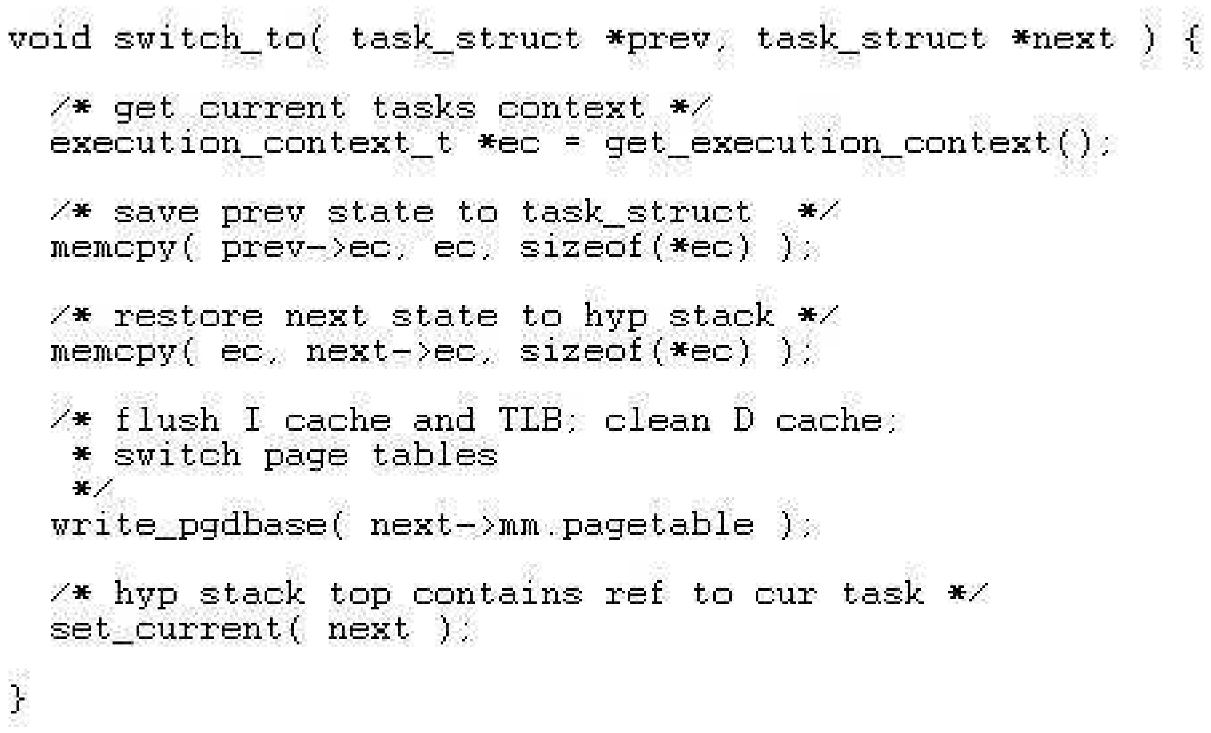
```
get_stack_top:
mov r0, #4092
orr r0, sp, r0
and r0, r0, #~3

get_current:
mov r0, #4092   @ hyp stack is one page
orr r0, sp, r0  @ top of stack
and r0, r0, #~3 @ round to word boundary
ldr r0, [r0]    @ addr of cur_task <- stack_top

set_current:
mov r0, #4092
orr r0, sp, r0
and r0, r0, #~3
str r1, [r0]    @ addr of cur_task -> stack_top

get_execution_context:
mov r1, #~4095
mov r2, #4096-76
and r1, sp, r1
add r1, r1, r2
mov r0, r1      @ r0 <- addr of exec_ctxt

schedule_tail:
adr r1, continue_nonidle_task
mov r0, #~4095
mov r2, #4096-76
and r0, sp, r0
add r0, r0, r2
mov sp, r0      @ setup stack pointer
mov pc, r1      @ jump to continue_nonidle_task
```

Figure 4.4: Assembly language routines for managing the hypervisor's stack.



```
typedef struct {
  unsigned long r0; /*working regs*/
  unsigned long r1;
  unsigned long r2;
  unsigned long r3;
  unsigned long r4;
  unsigned long r5;
  unsigned long r6;
  unsigned long r7;
  unsigned long r8;
  unsigned long r9;
  unsigned long r10;
  unsigned long fp; /*r11, frame ptr*/
  unsigned long ip; /*r12, intra-proc call scratch*/
  unsigned long sp; /*r13, stack ptr*/
  unsigned long lr; /*r14, link reg */
  unsigned long pc; /*r15, prog cntr*/
  unsigned long psr;/*status reg (mode bits/flags)*/
  unsigned long _unused;
} execution_context_t;
```

Figure 4.5: Execution context definition for tasks managed by the hypervisor.



```
void switch_to( task_struct *prev, task_struct *next ) {

   /* get current tasks context */
   execution_context_t *ec = get_execution_context();

   /* save prev state to task_struct  */
   memcpy( prev->ec, ec, sizeof(*ec) );

   /* restore next state to hyp stack */
   memcpy( ec, next->ec, sizeof(*ec) );

   /* flush I cache and TLB; clean D cache;
    * switch page tables
    */
   write_pgdbase( next->mm.pagetable );

   /* hyp stack top contains ref to cur task */
   set_current( next );

}
```

Figure 4.6: Pseudo code for context switching in the hypervisor.

### 4.1.11 Context switch code

The hypervisor makes use of the task management code and the execution context definition when switching between tasks. The task switching happens in the switch_to() method, pseudo code for which is shown in Figure 4.6. First the execution context of the current task is retrieved from the hypervisor's stack and saved to the task struct representing it. The execution context for the next task to be scheduled is then placed in hypervisor's stack and the pagetables for the newly scheduled task are installed. When this operation takes place the instruction cache and TLB on the SA-110 processor are flushed; its data cache is cleaned as well. Finally, the address of the task struct being scheduled for execution is placed on the hypervisor's stack using set_current().

The context switch code lives in `xen/arch/arm/process.c` which is 312 LOC compared to corresponding code for x86 that is 296 LOC

### 4.1.12 Domain Building and loading code

The domain loading code handles the details of loading a domain into its own, isolated paravirtualized environment. The code for building domain 0 is found in do_createdomain() function. The steps taken are outlined below and remain similar to those taken in the x86 version of Xen:

```
task_struct* do_createdomain( int dom_id ) {
    allocate a task structure and fill in its domain id;
    allocate a page for the shared_info portion of the task;
    mark the shared_info page as shared between the hypervisor and
        the domain in the frame table;
    allocate a page for the domain's per domain page tables;
    add the task_struct to the scheduler's list of runnables;
    initialize the domains list of allocated physical pages to 0;
    map dom_id to the task_struct ptr in the task_hash map;
    return the task_struct ptr;
}
```

This code is quite straight forward. The one thing that might not be entirely obvious is the use of the task_hash map. It is used so that the hypervisor can find the task structure representing a domain in constant time given its domain id. This is useful for performing operations on domains, such as killing them. It is also useful for finding the domain for which data on a pseudo network device is destined.

The code for loading domain 0 is found in the setup_guestos() function. The steps taken are outlined below and remain similar to those taken in the x86 version of Xen:

```
int setup_guestos( task_struct* tsk ) {
    if ( specified domain to be loaded is not 0 )
        return error;
    if ( specified task_struct already in 'constructed' state )
        return error;
    if ( first 8 bytes of guestOS image != 'XenoGues' )
        return error;
    /* allocate pages in the frame table for the domain */
    if ( tsk->phys_page_list = alloc_new_domain_mem() )
        return error; /* not enough free mem */

    /* build the page tables for the VM */
    allocate a phys page from tsk->phys_page_list to be the domains L1 pt;
    map the phys page into hypervisor's address space using the map cache;
    create entries in the L1 pt for the perdomain_pt and linear_pt regions;
    copy the hypervisor's entries into the guestOS's L1 pt, make them read-only;
    zero out all domain entries (all entries below upper 64MB);

    for ( each of the pages allocated to the domain ) {
```

```
    create L2 pt entries starting at virt_load_start (0xC000000);
    this involves allocating pages for the L2 tables;
    update pages usage flags and ref counts in the frame table;
    mark the page number down in the machine_to_phys mapping table;
}

/* now mark the pages used as page tables read-only */
for ( all pages used as L2 page tables ) {
    modify pgtbl entry's permissions to be read-only by the VM.
    update page's usage flags in the frame table to be of type 'pagetable';
}

setup shared_info area for the domain, resetting virtual time to 0;
unmap the domain's L1 pagetable from the map cache;

install the guestOS pagetables;
copy guest OS image to virt_load_start (0xC0000000);
setup start_info area with number of pages, the address of the
    shared_info struct, the page table base address, the domain id
    and it's flags;
reinstate hypervisor's page tables

mark the task_struct associated with the domain as 'constructed';
create a new thread for the task_struct;
return success;
```

```
}
```

One thing that we removed was the creation of virtual network interfaces and their addition to the start_info structure. The network VIFs have not been ported to the ARM due to time constraints and as such we cannot include them in the domain loading. For the same reason we have not give Domain 0 access to any of the real block devices via the virtual block device interface.

Another major difference in the domain loader code is that the flags values for page protection used in the page tables are different on ARM then they are on X86. In addition, the L1 and L2 pagetables are different in structure on ARM compared to x86. For example, the x86 L1 pagetables contains 1024 entries each mapping a 4MB region, whereas the ARM L1 pagetables contains 4096 entries each mapping a 1MB region. The x86 L2 pagetables contain 1024 entries each mapping a 4KB region, whereas the ARM L2 pagetables contain 256 entries each mapping a 4KB region. We had to be aware of these details because the physical pages were allocated to the domain using the frame table and are each 4KB in size. Thus, we needed to use the L2 pagetables to access memory at the correct granularity.

Other things the x86 version does such as storing the code segment selector for the event handler in the domain's task_struct are not needed. Finally, in the x86 version the guestOS image needed to be mapped into the hypervisor's address space using the map cache before the verification of the guest OS image could take place. In our version, the guest OS is linked into the hypervisor image and is already mapped as part of the monitor region at boot time, thus we don't have to do any extra mapping via the map cache to gain access to it.

Currently, additional domains are not supported. Further porting is required before loading additional domains is possible. First, the user space domain builder would need to be built into the guest OS in domain 0. Also, the final_setup_guestos() function needs to be ported. It is responsible for loading domains built by the domain builder into their own paravirtualized environment.

The code for domain loading resides in `xen/common/domain.c` where we modified 34 LOC to build pagetables specific to the ARM processor. Roughly 30 LOC were removed that had to do with initializint virtual block devices for the newly created domain.

### 4.1.13 Hypervisor entry code

The hypervisor entry code deals with low-level details that occur when moving between the hypervisor and a domain. This includes dispatching event callbacks and handling hypercalls. In Xen/x86 it is written in roughly 700 lines of x86 assembly language and needed to be ported to it's ARM equivalent. The following is a pseudo code summary of the hypervisor entry code.

```
continue_nonidle_task:
    tsk = get_current_task();
    goto test_all_events;

/* execute a list of 'nr_calls' hypercalls pointed at by 'call_list' */
do_multicall:
    for (i=0; i < nr_calls; i++) {
        hypervisor_call_table[ call_list[i].op ]( call_list[i].args[0],
            call_list[i].args[1],
            ...  );
    }
    goto ret_from_hypervisor_call;

restore_all_guest:
    /* pop the task's frozen state off the hypervisor stack.
    ** restore program counter and flags in one instruction
    ** to jump back to user mode.
    */
    movs pc, lr /* similar to iret in x86 assembly */

test_all_events:
    /* test for pending softirqs for this CPU */
    cpuid = tsk.processor;
    if ( softirq_pending( irq_stat[cpuid] ) ) {
        do_softirqs();
        goto test_all_events;
    }

    /* test for pending hypervisor events for this task */
    if ( hypevent_pending( tsk.hyp_events ) ) {
        do_hyp_events();
        goto test_all_events;
    }
    /* test for pending guest events for this task */
    if ( !  guestevent_pending( tsk.shared_info.events ) )
        goto restore_all_guest;

    /* process the guest events */
    guest_trap_bounce[cpuid].pc = tsk.event_callback_addr;
    create_bounce_frame();
    goto restore_all_guest;

create_bounce_frame:
    /* construct a complete stack frame that is a copy of the
    ** most recent saved task frame and call it new_stack_frame.
    */
    hyp_stack[sp] = new_stack_frame;
    hyp_stack[pc] = guest_trap_bounce[cpuid].pc;
    return;

hypervisor_call:
    /* get the hypercall number */
    hcno = get_hypercall_no();

    /* get the hypercall arguments 'arg1', 'arg2', etc.
    ** then call the function through the table of function pointers.
    */
    hypervisor_call_table[hcno]( arg1, arg2, ...  );


ret_from_hypervisor_call:
    goto test_all_events;
```

There are different ways of entering the hypervisor entry code. The first is through the hypervisor scheduler. Whenever a task is scheduled to be executed the Xen scheduler calls the *schedule_tail* function discussed above. This function drops into the top of the entry code at *continue_nonidle_task*.

The second way to gain entry is through a guest OS making a hypervisor call. Hypervisor calls are made through software interrupts initiated via the SWI instruction. The portion of the hypervisor that handles software interrupts does architecture specific processing such as saving the caller's state on the stack and zeroing out the frame pointer register before jumping to the *hypervisor_call* label. Returning from the hypercall happens at the *restore_all_guest* label where the saved state of the guest OS is popped off of the hypervisor's stack and user mode execution continues.

The mechanics of delivering an event are unique as well. The idea behind events is that they are the method by which interrupts are paravirtualized. Thus, an event can interrupt the execution of the guest OS at any time. In order to achieve this, the hypervisor is clever about how it constructs exception frames before delivering the event. First, a clone of the guest OSes saved state is created in the *create_bounce_frame* function. This cloned frame contains the state of the guest OS at the time it was interrupted, including the working registers, stack pointer, and program counter. Next, the hypervisor ties the new frame to the current stack frame by modifying the current frame's stack pointer to point to the new frame. In addition, the hypervisor modifies the current frame's program counter to be the address of the event callback handler in the guest OS. Once this is done, the event can be delivered and execution can be transferred to the guest OS via *restore_all_guest* which pops the current frame off of the stack. When this occurs, execution is transferred to the guest OS when the program counter is loaded with the address of its event handler. In addition, the stack pointer is loaded with the address of the cloned frame that was created in *create_bounce_frame*. Next, the event handler executes. When execution is complete, the event handler pops the cloned frame off of the stack and jumps directly to the spot where the guest OS was interrupted at the time the event occurred.

The main benefit of using the bounce frame to deliver events is that it cuts down on context switches between user and supervisor modes. Using the bounce frame, execution switches from the hypervisor to the guest OS event handling code and then directly to the spot where the guest OS was interrupted when the event occurred. An alternative approach would be to have the hypervisor create an event handler frame but not tie it to the current frame. Then, after the event is handled the guest OS would have to issue a software interrupt (maybe in the form of a special hypercall) to return to hypervisor which would then pop the current frame off the stack to return to the place where the guestOS was interrupted.

The *do_multicall* entry is actually the body of the hypervisor_multicall hypercall. It provides a way for a guest OS to execute a batch of hypercalls to reduce the overhead associated with entering and leaving the hypervisor on each call.

The hypervisor entry code resides in `xen/arch/i386/entry.S` for x86 and is 703 LOC in size. The entry code for the arm hypervisor is in `xen/arch/arm/entry-common.S` and is 753 LOC large.

## 4.2 Differences in paravirtualizing the architecture

There are a few major differences in the paravirtualized interface presented by the ARM compared to that of the x86 which we discussed in Chapter 3. In this section we give more details on the implementation.

### 4.2.1 Protecting the hypervisor from guest OSes

Guest OSes run in user mode while the hypervisor runs in supervisor mode. In order to make moving between the guest OS and the hypervisor possible without switching pagetables and flushing the TLB the guest OS pagetables map the hypervisor into the top 64MB of the address space. This is important to make operations that are executed frequently such as hypercalls and event delivery efficient. However, the hypervisor mappings must not be accessible from the guest OS, so we protect them with page-level protection as shown in Figure 4.7. Note that the hypervisor mappings are only accessible in supervisor mode while the guest OS mappings are accessible in both user and supervisor mode. Hypercalls use software interrupts to enter supervisor mode before jumping into the hypervisor. No page table switch is necessary since the hypervisor's memory mappings are included in the guestOS's pagetables.

### 4.2.2 Protecting the guest OSes from applications

This functionality is not currently implemented as we do not support applications. However, we outline a possible implementation here because of the importance of application support.

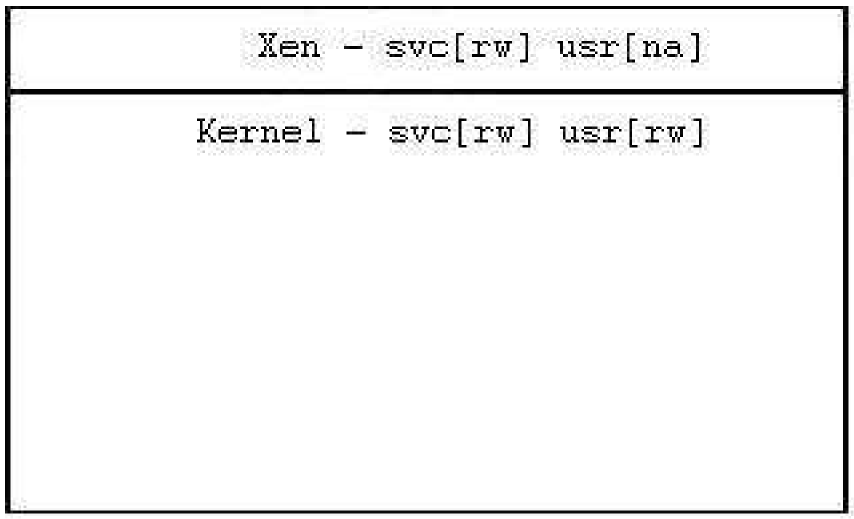


Figure 4.7: Hypervisor protects itself from guest OSes using page level protection.

As stated in Chapter 3, both guest OSes and applications run in user mode. For this reason we cannot use the access permission bits in their page mappings to protect a guest OS from its applications. Thus, each must have their own set of page tables and a pagetable switch along with accompanying TLB and cache flushes must occur when switching context between the two. The guest OS pagetables will contain mappings for itself and its applications as shown in Figure 4.8. Application pagetables hosted by guest OSes contain mappings for themselves and not for guest OSes as shown in Figure 4.9. This prevents applications from accessing guest OS memory. As shown above, all pagetables contain mappings for the hypervisor that are only accessible through supervisor mode.

**System calls**

To make a system call, program control must be transferred from the application to the guest OS. Along with the transfer of control, the addressing context will have to

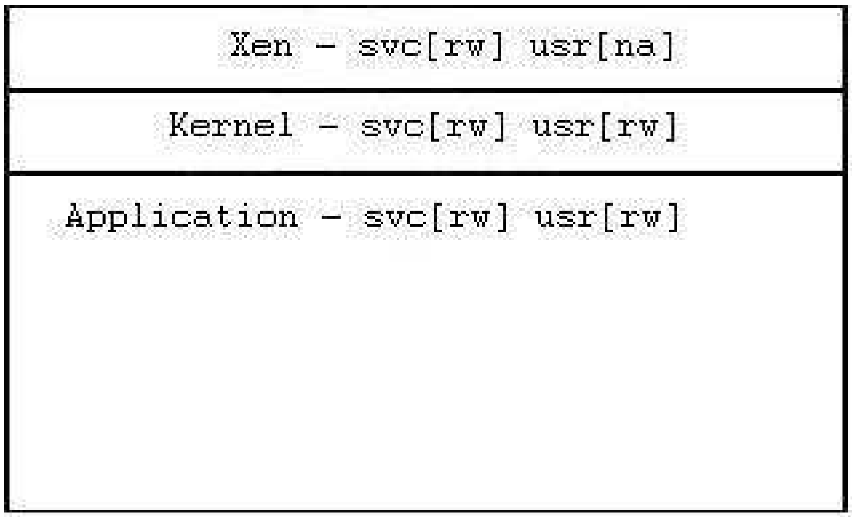


Figure 4.8: GuestOSes have access to application memory as well as their own.

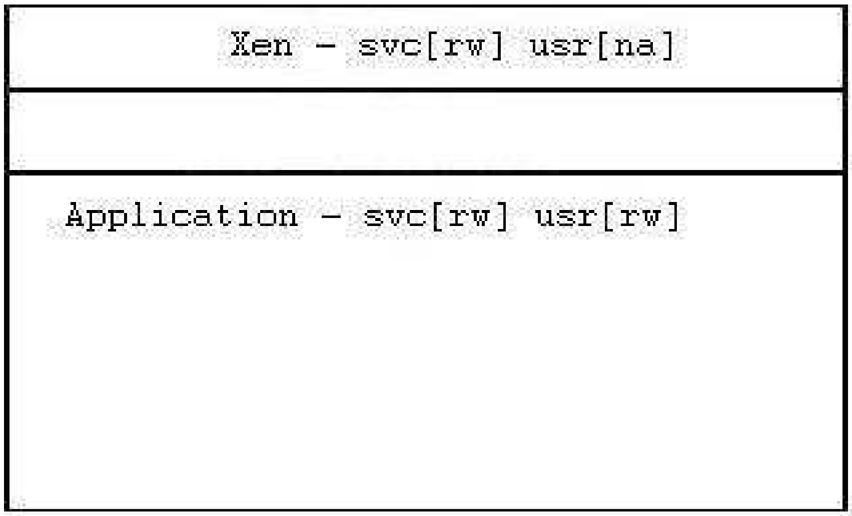


Figure 4.9: Applications have access to their own memory.

change from the application's context to the guest OS's context. Thus, the system call must enter the guest OS indirectly through the hypervisor where the page table mappings can be switched.

One possible implementation could be similar to that used in the x86/64 implementation of Xen. Two extra hypercalls could be added to the hypervisor named *syscall* and *sysreturn*. The *syscall* hypercall could take a call number and an array of arguments as parameters. Applications would use this hypercall to specify what system call they would like to execute. When handling *syscall*, the hypervisor could switch context from the application's pagetables to the guest OS's pagetables before bouncing off of a trampoline into the guest OS at the specified system call entry point with the specified arguments. The trampoline could be built at guest OS initialization time with the domain registering the addresses of system call handlers with the hypervisor. Once the execution of the system call is complete the guest OS has to return control to the hypervisor. This could be done using another hypercall, *sysreturn*. When handling *sysreturn*, the hypervisor would switch context from the guest OS's pagetables back to the application's pagetables before returning control back to the application.

The problem with the above implementation is that applications will have to be modified to use the new hypercall before they could run on the hypervisor. Another solution is that the hypervisor could keep track of what is executing in the domain, OS or application. Then an application would be able to make system calls using the regular system call numbers and the hypervisor could check upon receiving a software interrupt whether it is a system call or a hypercall by first determining what issued it. If the software interrupt was issued by an application then the hypervisor could translate it into the appropriate system call and deliver it to the guest OS via a trampoline as discussed earlier. If the software interrupt was issued by a guest OS then the hypervisor could execute the corresponding hypercall as it normally would.

All of the context switch overhead that occurs in the above schemes during a system call is costly. As we mentioned in Chapter 3, the cost of context switching could be reduced from a pagetable switch and TLB flush to a simple reload of the Domain Access Control Register if we use the techniques for Fast Address Space Switching on the ARM. Another way to reduce the overhead would be to allow applications to batch system calls. The hypervisor already provides the functionality to batch hypercalls using the *hypervisor_multicall* hypercall. Applications could execute *hypervisor_multicall* with a list of *syscall* operations as arguments. This is a simple way of amortizing the cost of system calls, but the applications would have to be modified to make use of the multicall functionality.

## 4.3 Summary of Implemented Functionality

In this section we present a brief summary of the features that are implemented in the hypervisor, as well as a listing of the features are not supported.

### Supported Features

The features that the hypervisor supports are:

- hypervisor boot on the DNARD.
- output via the serial console.
- real time clock hardware initialization.

- software interrupts and accurate timers.
- notion of virtual and wallclock time.
- hypervisor events used to asynchronously pass messages to the hypervisor.
- guest events used to virtualize interrupts.
- hosting a single guest OS.
- isolation of the hypervisor from the hosted guest OS using page level protection in conjunction with the two physical execution modes on the ARM cpu.

**Unsupported Features**

The features that the hypervisor does not support are:

- frame table management of discontiguous physical memory.
- hypervisor heap allocator management of discontiguous physical memory..
- isolation of guest OSes from their hosted applications using page level protection in conjuction with the two physical execution modes on the ARM cpu plus a virtual execution mode in the hypervisor.
- hosting more than one guest OS.
- ethernet hardware initialization.
- virtual block devices.
- virtual network interfaces.
- transfering data between the hypervisor and guest OSes using Xen's asynchronous I/O rings.

# Chapter 5

# Performance

This chapter presents performance measurements of our port of the Xen hypervisor for the ARM platform. We evaluate the performance of our system by microbenchmarking four of its operations: handling hypercalls, handling batched hypercalls, delivering events, and loading a domain. We also compare the memory consumption of our prototype to the memory consumption of the Xen 1.2 hypervisor for x86.

## 5.1 Minimal OS

In order to measure the performance of our hypervisor we needed a guest operating system to exercise the operations it supports. The Xen 1.2 codebase contains such a test operating system named Mini-OS. We ported Mini-OS to our prototype hypervisor for the ARM; it demonstrates how a guest OS performs the following actions.

- parsing the start_info struct at boot time.
- registering virtual interrupt handlers for timer interrupts.

- handling asynchronous events.
- enabling and disabling asynchronous events.
- it also includes a simple page and memory allocator as well as minimal libc support.

The port was straightforward. For the most part, once the code compiled it ran correctly. The biggest challenge was converting the architecture dependent boot and entry code for the Mini-OS from x86 assembly to ARM assembly. The entry code was the most complicated as it handles asynchronous event callbacks from the hypervisor.

## 5.2 Experimental Setup

The experiments were run on a single DNARD Shark device with a 233MHz SA-110 StrongARM processor and 32MB of memory [Dig97]. For each of the micro-benchmarks the Mini-OS was loaded as Domain 0 on top of the hypervisor.

### 5.2.1 Micro-benchmarks

For each of the operations below we modified the hypervisor so that the cost of entering the scheduler would not affect our results. To achieve this, we modified the RTC interrupt handler in the hypervisor to only update its notion of the system time and the timing information in the shared_info struct. The RTC interrupt handler does not post the softirq that triggers the scheduler.

Once the modifications were made, we used the hypervisor's notion of the system time to measure how long a particular operation would take[1]. However, because the RTC interrupt only fires at a frequency of 128Hz as dictated by the real time clock on the DNARD, we could only measure time at a 7ms granularity. Thus we could not measure the time it took to perform a single operation accurately. Instead, we measured the time it took to perform $10^6$ repeated operations and calculated the average time per operation.

### Hypercalls

In order to measure the time consumed while making a hypercall we needed to issue hypercalls that did not spend time doing any hypercall-specific processing. Fortunately there is already such a hypercall dubbed the no-op syscall, its body is shown below:

```
asmlinkage long sys_ni_syscall( void ) {
      return -ENOSYS;
}
```

We modified the Mini-OS code to call the no-op syscall repeatedly to calculate the average time it took to perform a single hypercall. The code to do this is shown below:

```
static const int nr_calls = 1000000;
int i = 0; HYPERVISOR_print_systime();
```

[1]The hypervisor stores system time as an unsigned 64-bit integer that contains the number of nano-seconds that have passed since the hypervisor was booted.

```
for (i=0; i < nr_calls; i++) {
      HYPERVISOR_noop();
}
HYPERVISOR_print_systime();
```

We added the *HYPERVISOR_print_systime* hypercall to the hypervisor and use it to display the system time before and after a batch of operations takes place. Its body is shown below:

```
asmlinkage long do_print_systime( void ) {
      s_time_t t = get_s_time();
      printk("!!!  system_time=%llu", t);
      return 0;
}
```

The *get_s_time* function is only accessible from inside the hypervisor. It returns the hypervisors notion of the system time.

### Batched Hypercalls

We took a similar approach to measuring the performance of hypercall batching. The code to perform a large number of hypercalls in a batch and measure the time it took is shown below:

```
static multicall_entry_t calls[1];

calls[0].op = __HYPERVISOR_noop;

HYPERVISOR_print_systime();

HYPERVISOR_multicall(calls, nr_calls);

HYPERVISOR_print_systime();
```

It should be noted that in order to perform such a large number of batched hypercalls we needed to hack the way *HYPERVISOR_multicall* works. Notice that the number of multicall_entry structs declared is 1, where usually it would usually equal the number of hypercalls to be batched. On the DNARD, it is not possible for us to declare an array of $10^6$ multicall structs because there is not enough physical memory[2]. As a result, the implementation of *do_multicall* in the hypervisor was modified to repeat the first operation in the multicall_entry array the specified number of times for the purpose of evaluation.

**Delivering Events**

We took a similar approach to measuring the performance of event delivery and processing. We added a hypercall to the hypervisor that caused the notification of an event to the Mini-OS. We then used this new hypercall to repeatedly send events to the Mini-OS. The code to perform a large number of event deliveries and measure the time it took is shown below:

```
add_ev_action(EV_TIMER, &timer_handler);
```

[2]Each multicall_entry_t requires 32 bytes. An array of $10^6$ multicall_entry structs would therefore require 31250KB (just over 30.5MB) of memory.

```
enable_ev_action(EV_TIMER);

enable_hypervisor_event(EV_TIMER);

HYPERVISOR_print_systime();

for (i=0; i < nr_calls; i++) {

       HYPERVISOR_set_timer_event();

}

HYPERVISOR_print_systime();
```

The *add_ev_action* function associates an event handler with an event number in the Mini-OS. The *enable_ev_action* function sets the 'enable' bit in the local mask associated with the timer event. This tells the Mini-OS to allow the timer callback function to be called when an event occurs. The *enable_hypervisor_event* function sets the bit corresponding to the timer event in the 'events_mask' field of the Mini-OS's shared_info struct. This effectively tells the hypervisor to enable timer event delivery for the Mini-OS.

The timer callback function that we used for our measurements does not perform any event-specific processing. Its body is shown below:

```
static void timer_handler(int ev, ...)  {

       return;

}
```

Finally, we needed to add functionality to the hypervisor that caused the notification of timer events. The required functionality took the form of the *HYPERVI-*

*SOR_set_timer_event* hypercall[3]. Its body is shown below:

```
asmlinkage long do_set_timer_event(void) {
        unsigned long flags;
        struct task_struct *p;
        /* send virtual timer interrupt */
        read_lock_irqsave(&tasklist_lock, flags);
        p = &idle0_task_union.task;
        do {
                if ( is_idle_task(p) ) continue;
                test_and_set_bit(_EVENT_TIMER, &p->shared_info->events);
        }
        while ( (p = p->next_task) != &idle0_task_union.task );
        read_unlock_irqrestore(&tasklist_lock, flags);
        return 0;
}
```

This function iterates through the task list and sets the bit corresponding to the timer event for each domain. As soon as the hypercall's processing is complete the hypervisor tests for events to that need to be delivered (as shown in Chapter 4's discussion of the 'Hypervisor Entry Code'). When the hypervisor discovers that there is a timer event to deliver and that timer events are enabled in the Mini-OS's events_mask it creates a bounce frame on the Mini-OS's stack and delivers the event to the Mini-OS. The Mini-OS performs event specific processing by calling

[3]It should be noted that our measurement of the time taken to deliver an event includes the time to execute the hypercall which triggers the event

the timer event callback function. Once the event has been delivered the Mini-OS pops the bounce frame off of its stack to return back to where the *HYPERVISOR_set_timer_event* function was called.

### Domain Loading

Since we only support loading a single domain we can only measure loading domain 0. There are two routines in the hypervisor that build and load domain 0; we simply measure the time it takes to execute them using the following code:

```
before = get_s_time();

new_dom = do_createdomain( 0, 0 );

setup_guestos( new_dom, ...  );

after = get_s_time();
```

Table 5.1 shows the results of our micro-benchmarks.

| Operation | Time per $10^6$ operations (ms) | Average Time per operation ($\mu$s) |
|---|---|---|
| Hypercall | 504 | 0.504 |
| Batched Hypercall | 98 | 0.098 |
| Event Delivery | 1960 | 1.960 |
| Domain Loading | n/a | 49.0 |

Table 5.1: Micro-benchmark results for the ARM hypervisor prototype.

Note that loading a domain requires the most time at 49$\mu$s. This is mostly due to the fact that the domain loader must switch to the guest OSes addressing context before copying the guest OS image to the correct location in memory, and swith back to its own addressing context after the copy is complete. Switching addressing contexts is expensive because it reloads the pagetables to be used, requiring TLB and cache flushes. Copying memory from one location to another is inherently expensive.

Delivering events is the next most expensive operation and takes almost 2$\mu$s per event. This is due to the creation and use of the bounce frame and to the extra processing done in the event handling code in the Mini-OS. Creating the bounce frame is not bad on its own. It only requires that 72 bytes be pushed onto the guest OSes stack. Popping an exception frame is potentially expensive as it requires jumping to a completely different region of code, possibly thwarting cache locality. The extra processing done in the event handling code is a result of demultiplexing the event to the correct handler.

Hyercalls are the next most expensive operation as they take just over 0.5$\mu$s. This is due to issuing the software interrupt and returning to the guest OS by popping the exception frame off of the stack. With our measurements it is possible to see how issuing hypercalls in batches saves execution time since we do not have to enter and leave the hypervisor repeatedly. A hypercall executed in a batch is over 0.4$\mu$s faster than a standalone hypercall on average.

## 5.3 Soft Evaluation of Memory Consumption

It is interesting to compare the memory consumption of our prototype to the hypervisor in Xen 1.2 for x86. Table 5.2 shows this comparison with a break down of the direct mapped hypervisor regions.

| Region (MB) | ARM | x86 |
|---|---|---|
| Monitor | 8 | 16 |
| Machine-to-physical Mapping Table | 4 | 4 |
| Frame Table | 4 | 20 |
| Total Memory Consumed | 16 | 40 |

Table 5.2: Comparison of hypervisor memory consumption on ARM and x86.

The memory consumption of the monitor region was reduced by one-half. It contains code, static data, and space for dynamic memory allocation for the hypervisor. Since the implementation of our hypervisor is very stripped down in terms of features and we currently only support a single domain without applications we could reduce the monitor region safely. However, if our implementation was feature complete and could host the same number of operating systems that is possible on the x86 implementation then it would need to use 16MB of memory. The reason for this increased memory usage is not because of increased code size, but because of the space that the data structures internal to the hypervisor require when supporting multiple domains.

The memory consumption of the machine-to-physical mapping table was not reduced at all because we do not change the implementation of the mapping table from its x86 counterpart. The mapping table is used by the hypervisor to produce the illusion of contiguous physical memory to guest operating systems. It maps a machine page address to a physical page number. A page is the same size on both the x86 and ARM implementations at 4KB. Given that the maximum amount of physical memory on an ARM device is 4GB we use the following calculation to determine the size of the mapping table:

```
mpt_size = ( max_phys_addr / page_size ) * entry_size
```

Given that an entry in the table is an integer (the page number) the entry size is 4 bytes and the machine to physical mapping table must be 4MB in size. This calculation assumes that a simple linear mapping from machine page address to physical page number is used. We could have modified the machine to physical mapping table implementation to take the physical memory present on the machine into account instead of using the maximum theoretical physical address. There was no need to change the implementation since we can afford to use the required 4MB easily. It is also worth noting that the current implementation provides for efficient page number lookups. Retrieving a physical page number given a machine page address is done in the following way.

```
phys_page_no = mpt[ phys_page_addr >> PAGE_SHIFT ]
```

Where PAGE_SHIFT is 12; this gives us the entry in the mapping table for a 4KB page (NB: $1 << 12 = 4096$). It is important that the above lookup operation can be performed quickly since it becomes an extra level of address translation that is performed by the hypervisor when presenting the illusion of contiguous physical memory.

The memory consumption of the frame table region was reduced by a factor of six from 24MB to 4MB. This was possible because the frame table is used to track the allocation and use of physical pages by domains in the system. As such, its size could be reduced given the amount of physical memory available on a device. The frame table uses the pfn_info structure to maintain information about a physical page. This structure requires 20bytes of storage. In our implementation, we modified the frame table to maintain allocation information for physical memory bank #2 which is 8MB in size. Therefore, the size of the frame table needed to maintain information can be calculated as:

```
ft_size = ( phys_mem_size / page_size ) * pfn_info_size
```

Where phys_mem_size is 8MB, page_size is 4KB, and pfn_info_size is 20 bytes. As a result we find that a frame table of 40KB in size is sufficient to track the use of 8MB of memory[4]. If we were to modify the frame table to track the allocation of physical memory from both bank #2 and bank #3 then we would still only require 80KB. Clearly the 4MB of space allocated to the region is more than enough and could be optimized further should other parts of the system require more memory as it evolves. It should also be noted that by changing the size of the frame table region we do not affect the efficiency of lookups in the table since the underlying linked list implementation is left untouched.

[4]The x86 frame table size of 20MB is enough to track the use of 4GB of physical memory.

## 5.4 Summary

Overall the results of our measurements are encouraging. The micro-benchmarks of the operations supported by the hypervisor were held back by the accuracy of the timing hardware on our test devices. However, we were able to repeatedly execute operations a high enough number of times to obtain a reasonably accurate measure of the average time per operation. In addition, the evaluation of the memory consumed by the hypervisor confirms the lightweight implementation used by the Xen team and that Xen could be used on a handheld device supporting a limited amount of memory.

# Chapter 6

# Conclusion and Future Work

## 6.1 Conclusion

As a result of this thesis, we have an implementation of the Xen 1.2 hypervisor ported to the first generation StrongARM processor, the SA-110. The hypervisor is capable of loading guest operating systems, servicing hypercalls, and delivering events. The hypervisor uses page-level protection to isolate itself from guest operating systems but does not support applications. The hypervisor paravirtualizes the CPU and memory but does not support device paravirtualization

Porting the hypervisor to a different architecture was difficult for a few reasons. As a hardware hosted virtual machine monitor, the hypervisor sits directly on bare metal. This restricted our options for debugging when things went wrong. In fact, the debugging environment consisted of sending characters to the serial console. Debugging virtual memory problems constituted dumping pagetable entries out to the serial console. Similarly, debugging hypercalls and event delivery required dumping the stack(s) to the serial console.

The final reason that porting the hypervisor was a challenge is because its implementation is x86 specific. The implementation of features for this project followed a three step process.

1. Figure out what the x86 hypervisor is doing.
2. Figure out why the x86 hypervisor is doing what its doing.
3. Write the corresponding feature for the ARM version of the hypervisor.

In some cases step 2 could be more time consuming than the other steps due to the eclectic nature of the x86 architecture. Other times steps 1 and 3 were the most time consuming due to the fact that the code was written in assembly language.

We have supported the implementation of the Xen hypervisor for StrongARM with a port of the Mini-OS found in the Xen 1.2 codebase. In addition, we present some hard performance numbers of execution time as well as a soft evaluation of the hypervisor's memory usage. Of particular interest is that the StrongARM version of the hypervisor consumes less than half of the memory that the x86 version does. This was possible because of the clever way the Xen team designed the hypervisor and the ability to resize the regions of memory managed by the hypervisor depending on the amount of physical memory available. Our results suggest that our hypervisor could perform comfortably on a StrongARM powered handheld device supporting a limited amount of memory and processing power.

This thesis has presented a case for the use of small and fast virtual machine monitors such as Xen on mobile devices powered by the StrongARM architecture. The implementation of a prototype hypervisor and an evaluation of its performance show that the use of such a paravirtualization architecture is attractive and feasible on the StrongARM. Finally, we have shown that while Xen style paravirtualization of the ARM architecture requires guest operating systems to be ported to run on top of the hypervisor, the ARM instruction set architecture is far more accommodating than its x86 counterpart.

## 6.2 Future Work

The prototype implemented in this thesis supports only the base functionality to load a test operating system. There are many features needed to support a fully fledged general purpose OS that would need to be implemented, and as such the opportunities for future work are great.

The frame table can currently only manage a contiguous range of memory. This lead to the final 8MB bank of physical memory on our DNARD test devices being unusable by our prototype. There are two possible solutions to this problem. We could maintain one free frame list in the frame table for each region of physical memory to be managed. Under this scheme clients of the frame table could adjust frame addresses by an offset corresponding to the free list (i.e., region) that the frame belongs to. The other solution is to maintain a single free frame list and store a physical offset as part of each node in the list. In either case, the frame table management code needs to associate a physical offset with each frame being managed instead of assuming that all of memory is contiguous and keeping the offset constant.

Once the hypervisor is ready it would be interesting to try embedding a few simple security services for it such as the secure logging facility used in ReVirt. Once a secure logging service is built it will be worthwhile to think about what security services could be most useful in a mobile environment where there is no hard disk to store log files and the limited amount of physical memory makes certain types of processing difficult. Perhaps a system where mobile devices running secure logging services stream their logs to a central server for processing could be explored. There may be interesting ways that the log files from different devices could be cross referenced in the spirit of intrusion detection systems to backtrack network worm propagation to its source. In a similar vein, capabilities from the Snort network intrusion detection system [sno05] were recently embedded in the Xen hypervisor and demonstrated by XenSource [Xen05].

Other services that could be useful in a mobile environment could be explored as well. The quality-of-service tools in Xen 3.0 could be used as a basis for providing even more control over the resource utilization of hosted virtual machines. Such resource control would be useful in a mobile environment for applications such as file sharing. For example, users might be more inclined to run a file sharing daemon on their handheld device if they could place accurate restrictions on the amount of network bandwidth, CPU time, and, probably most importantly, battery power being consumed.

Moving our implementation into the Xen 3.0 codebase would make such a project easier as this would make the high-level QoS tools available to us. In order to move to the Xen 3.0 codebase a version of the GCC 3.4 or 4.0 cross compiler for ARM would need to become more stable. At the very least, the bug that caused compilation of the DNARD boot code to fail with a segmentation fault would need to be fixed. Once an acceptable version of the compiler is available it is predicted that the biggest changes to our implementation of the hypervisor would be due to the support for multi-threaded guest operating systems that was introduced in Xen 3.0.

A port of a general purpose operating system such as Linux to the ARM port of the hypervisor would also be worthwhile. It would require the implementation

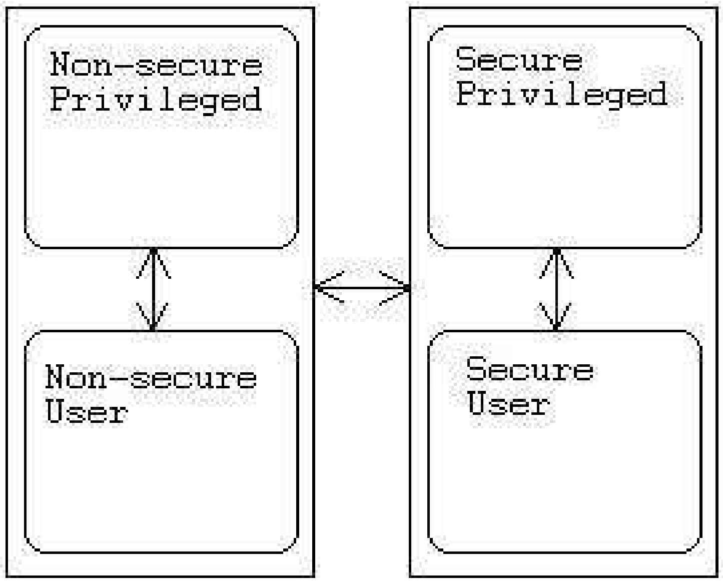


Figure 6.1: Modes of operation for processors supporting the ARM TrustZone hardware extensions.

of features that we could not complete due to time constraints such as support for guest OS applications. Once these features are implemented the performance of switching between the context of guest operating systems and their applications could be measured. In addition, it would be interesting to implement a fast address space switching scheme similar to [WH00] in the hypervisor and investigate the speedup it provides when switching between guest OSes and their applications.

One other direction of future work could be to make use of the TrustZone hardware extensions that were recently added to the ARM architecture [AF04]. Using these extensions, software can be run in one of the four execution modes shown in Figure 6.1: unsecure user, unsecure supervisor, secure user, and secure supervisor. There are a few possibilities for how these extensions could be used. We could use the TrustZone software provided by ARM to have the hypervisor make use of the TrustZone API for building secure services [ARM05a]. Using the TrustZone

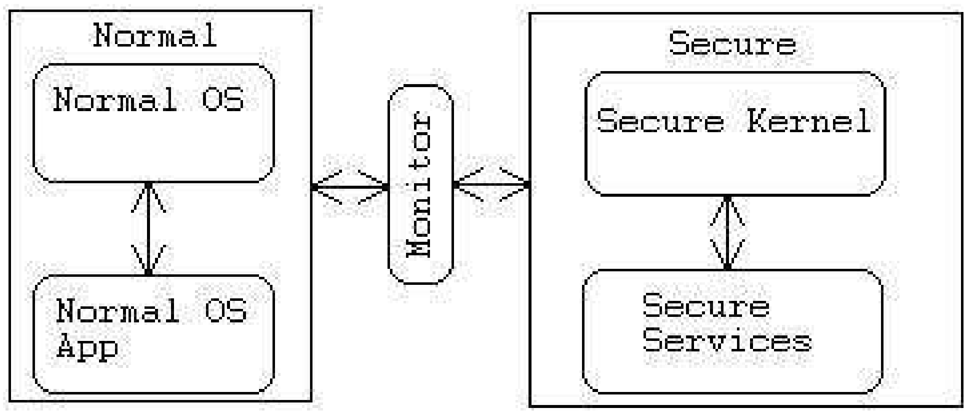


Figure 6.2: Organization of a system using the ARM TrustZone API.

software the system would be organized similar to Figure 6.2. The monitor, secure kernel, and secure services operate in the secure mode with the monitor and secure kernel software supplied by ARM. Using the API the hypervisor would execute in unsecure supervisor mode with applications operating in unsecure user mode; any secure services we write would execute in secure user mode. This would give us flexibility deciding where we would like to place the code for the services. For example, we could embed the service's code inside the hypervisor or place the code inside of a guest OS and the code would still execute in secure mode on the processor.

Alternatively we could eschew the provided TrustZone API altogether and work directly with the hardware extensions themselves. Doing this would make it possible to organize the system in many different ways with different organizations providing parts of the system with varying levels of control and performance. For example, it would be possible to have the hypervisor operate in secure supervisor mode with the guest OSes running in unsecure supervisor mode and their applications running in unsecure user mode. Any security services could then operate in secure user mode. Under this organization, a system call could be issued by a guest

OS executing an SMI (secure mode interrupt) instruction. It would be interesting to see how the performance of the system is affected by the different organizations. For example, switching context to and from secure mode may have adverse affects on performance. Switching to and from secure mode will necessitate TLB and cache flushes unless processors supporting the TrustZone extensions provide tagged TLBs and caches.

# Bibliography

[Adv05] Advanced Micro Devices. *AMD64 Virtualization Codenamed "Pacifica" Technology Secure Virtual Machine Architecture Reference Manual*, May 2005.

[AF04] T. Alves and D. Felton. Trustzone: Integrated hardware and software security, July 2004.

[ARM05a] ARM. *TrustZone Software API Specification*, July 2005.

[arm05b] Arm linux project website, 2005. http://www.arm.linux.org.uk/.

[BDF$^+$03] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, E. Kotsovinos, A. Madhavapeddy, R. Neugebauer, I. Pratt, and A. Warfield. Xen 2002. Technical Report 553, University of Cambridge, Computer Laboratory, January 2003.

[BX06] H. Blanchard and J. Xenidis. Xen on powerpc. In *linux.conf.au*, 2006.

[CDD$^+$04] Bryan Clark, Todd Deshane, Eli Dow, Stephen Evanchik, Matthew Finlayson, Jason Herne, and Jeanna Neefe Matthews. Xen and the art of repeated research. In *USENIX Annual Technical Conference, FREENIX Track*, pages 135–144, 2004.

[che05] checkps project website, 2005. http://sourceforge.net/projects/checkps/.

[DFH$^+$03] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the art of virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles*, October 2003.

[Dig97] Digital Equipment Corporation. *DIGITAL Network Appliance Reference Design: User's Guide*, November 1997.

[DKC$^+$02] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. Revirt: enabling intrusion analysis through virtual-machine logging and replay. *SIGOPS Oper. Syst. Rev.*, 36(SI):211–224, 2002.

[FHN$^+$04] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the xen virtual machine monitor. In *OASIS ASPLOS, Workshop*, 2004.

[Fra03] K. Fraser. Post to linux kernel mailing list, October 2003. http://www.ussg.iu.edu/hypermail/linux/kernel/0310.0/0550.html.

[FSM05] FSMLabs. Rtlinux website, 2005. http://www.fsmlabs.com/.

[Gar04] Larry Garfield. 'metal gear' symbian os trojan disables anti-virus software. *InfosyncWorld*, December 2004. http://www.infosyncworld.com/news/n/5654.html.

[Gol72] R. Goldberg. *Architectural Principles for Virtual Computer Systems.* PhD thesis, Harvard University, Cambridge, MA, 1972.

[GR03] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proc. Network and Distributed Systems Security Symposium*, February 2003.

[gru05] Gnu grub project website, 2005. http://www.gnu.org/software/grub/.

[Int98a] Intel. *SA-110 Microprossor: Technical Reference Manual*, September 1998.

[Int98b] Intel. *SA-1100 Microprocessor: Technical Reference Manual*, September 1998.

[Int03] Intel. *Intel 80200 Processor based on Intel XScale Microarchitecture: Developer's Manual*, March 2003.

[Int05a] Intel. *IA-32 Intel Architecture Software Developer's Manual, Vol 1: Basic Architecture*, September 2005.

[Int05b] Intel. *Intel Virtualization Technology for the Intel Itanium Architecture*, April 2005.

[Int05c] Intel. *Intel Virtualization Technology Specification for the IA-32 Intel Architecture*, April 2005.

[Jal05] Jaluna. Jaluna website, 2005. http://www.jaluna.com/.

[JFMA04] Nick L. Petroni Jr., Timothy Fraser, Jesus Molina, and William A. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *USENIX Security Symposium*, pages 179–194, 2004.

[Kaw05] Dawn Kawamoto. Cell phone virus tries leaping to pcs. *ZDNet Asia*, September 2005. http://www.zdnetasia.com/news/security/0,39044215,39257506,00.htm.

[KDC03] Samuel T. King, George W. Dunlap, and Peter M. Chen. Operating system support for virtual machines. In *USENIX Annual Technical Conference, General Track*, pages 71–84, 2003.

[Keg05] D. Kegel. Crosstool project website, 2005. http://kegel.com/crosstool/.

[KMAC03] Chetana N. Keltcher, Kevin J. McGrath, Ardsher Ahmed, and Pat Conway. The amd opteron processor for multiprocessor servers. *IEEE Micro*, 23(2):66–76, March/April 2003.

[PD80] David A. Patterson and David R. Ditzel. The case for the reduced instruction set computer. *SIGARCH Comput. Archit. News*, 8(6):25–33, 1980.

[PFH$^+$05] I. Pratt, K. Fraser, S. Hand, C. Limpach, A. Warfield, D. Magenheimer, J. Nakajima, and A. Mallick. Xen 3.0 and the art of virtualization, 2005. Ottawa Linux Symposium 2005 presentation.

[RI00] J. Robin and C. Irvine. Analysis of the intel pentium's ability to support a secure virtual machine monitor, 2000.

[s0f05] s0ftpr0ject. Kstat website, 2005. http://www.s0ftpj.org/en/site.html.

[Sea00] David Seal. *ARM Architecture Reference Manual.* Addison-Wesley, 2 edition, December 2000.

[sha05] Shark linux project website, 2005. http://www.shark-linux.de/shark.html.

[Sim05] SimWorks. Simworks anti-virus website, 2005. http://www.simworks.biz/sav/AntiVirus.php?id=home.

[sno05] Snort project website, 2005. http://www.snort.org/.

[SS72] Michael D. Schroeder and Jerome H. Saltzer. A hardware architecture for implementing protection rings. *Commun. ACM*, 15(3):157–170, 1972.

[Sun05] Jorgen Sundgot. First symbian os virus to replicate over mms appears. *InfosyncWorld*, March 2005. http://www.infosyncworld.com/news/n/5835.html.

[SVL01] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 1–14, Berkeley, CA, USA, 2001. USENIX Association.

[Tri05] Tripwire. Tripwire website, 2005. http://www.tripwire.org/.

[VMw05] VMware. Vmware website, 2005. http://www.vmware.com/.

[Wal02] Carl A. Waldspurger. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, 2002.

[WH00] A. Wiggins and G. Heiser. Fast address-space switching on the strongarm sa-1100 processor, 2000.

[Wro05] Jay Wrolstad. Mabir smartphone virus targets symbian-based mobile phones. *Contact Center Today*, April 2005. http://www.contact-center-today.com/ccttechbrief/story.xhtml?story_id=32327.

[WSG02] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Denali: A scalable isolation kernel. In *Proceedings of the Tenth ACM SIGOPS European Workshop*, St. Emilion, France, September 2002.

[WTUH03] A. Wiggins, H. Tuch, V. Uhlig, and G. Heiser. Implementation of fast address-space switching and tlb sharing on the strongarm processor, 2003.

[Xen05] XenSource. Xensource showcases secure xen hypervisor, August 2005. http://www.xensource.com/news/pr082505.html.