Asbestos: Operating System Security for Mobile Devices

by

Martijn Stevenson

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2006

Author . . . .
Department of Electrical Engineering and Computer Science
May 26, 2006

Certified by . . . .
Robert T. Morris
Associate Professor
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Asbestos: Operating System Security for Mobile Devices

by

Martijn Stevenson

Submitted to the Department of Electrical Engineering and Computer Science
on May 26, 2006, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

This thesis presents the design and implementation of a port of the Asbestos operating system to the ARM processor. The port to the ARM allows Asbestos to run on mobile devices such as cell phones and personal digital assistants. These mobile, wireless-enabled devices are at risk for data attacks because they store private data but often roam in public networks. The Asbestos operating system is designed to prevent disclosure of such data. The port includes a file system and a network driver, which together enable future development of Asbestos applications on the ARM platform. This thesis evaluates the port with a performance comparison between Asbestos running on an HP iPAQ hand held computer and the original x86 Asbestos.

Thesis Supervisor: Robert T. Morris
Title: Associate Professor

Acknowledgments

I would like to thank my thesis advisor, Robert T. Morris, for his support and guidance on this thesis. Robert always welcomed and answered technical inquiries and he provided a strong vision for what this thesis should accomplish. Without his insights the ARM Asbestos port would not have reached completion.

The people of the Parallel and Distributed Operating Systems group at MIT helped make my working environment a pleasant one. While Professor Frans Kaashoek appreciated my accomplishments, Micah Brodsky and Max Krohn shared my frustrations and often provided a helpful set of eyes or ears when problems arose. In addition, I thank Asbestos team members Eddie Kohler, Steve VanDeBogart and Petros Efstathopoulos for giving me code, critique, benchmarks and advice. Helpful also were the words of iPAQ wizard Jamey Hicks, and the work of other developers at `www.handhelds.org`.

Finally I want to thank my parents for the support they have given me these past five years at MIT. I have always been able to count on them for words of wisdom and encouragement.

This research was supported by a grant from Taiwan's Industrial Technology Research Institute.

Portions of this thesis are adapted from and/or contain text written in master's theses by Cliff Frey and David Ziegler.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis presents the port of the Asbestos operating system to the ARM processor architecture. ARM Asbestos brings the operating system security guarantees of Asbestos to a mobile computer, supplemented by a file system and a network driver.

1.1 Motivation: Asbestos

On most networked computers, only a suite of user-level software protects private data from remote access. Experience has shown that it is too difficult to ensure that all software handling private data is free of exploitable flaws. We routinely learn about exploits made possible by buffer overflows [11, 10, 9, 12, 14], vulnerabilities in the Java virtual machine [13], and improper allocation and deallocation of memory [15]. Typically, these vulnerabilities result in the complete compromise of the system.

Instead of relying on applications to protect sensitive data, one can contain software flaws at the operating system level, using the *principle of least privilege* [36]: each part of a system should have no more than the capabilities required for it to perform its tasks. For example, an email reader should be able to confine an executable attachment by only giving it access to a display window and restricting its access to the network and file system. A Web application should be able to ensure that one user's private data cannot be sent to another user's browser by faulty Web server application code.

The Asbestos operating system employs the principle of least privilege to limit the impact of exploited software vulnerabilities [28, 19]. It allows programmers to dynamically define data compartments, and only grant access to that data to those system services that require it. An example is the OK Web Server (OKWS) for Asbestos [29]. Using the security primitive of Asbestos labels, OKWS creates a compartment for each user's data. It restricts the compartment such that it can only communicate with a specialized worker process, the trusted local database, and a single connection to the network. The rest of the network connections and the other processes cannot communicate with the worker, thus preventing the unintended disclosure of information.

1.2 Motivation: Mobile Security

The development of Asbestos OKWS for desktop servers was a direct response to the security breaches experienced in recent years by web-based organizations. However, desktop servers are not the only types of computers coping with threats coming from the network. The mobile device space is a prime candidate for security research, because hand held devices are facing increasing threats as their popularity grows. The security threats to mobile devices are similar to the threats facing servers in that these devices can remain on without user attention, and are always connected to a network. There are already billions of cell phone, PDA and smart phone users [34], whose devices often store private data such as phone numbers and addresses. The demand for global connectivity is quickly driving phone service providers to consolidate networks and standardize communication protocols [20], making it easier to create attacks that affect large numbers of users. Network bandwidth and storage are expanding to meet increased demand for data downloads and mobile data, increasing the risk of data leaks and the spread of malicious code. The first mobile worms have already appeared [37], and we should expect the security risks to increase as mobile networks grow and users trust their mobile device software to handle more personal data.

1.3 Motivation: ARM

The Asbestos port to ARM is beneficial because the ARM processor dominates mobile computing. The x86 processor is one of the most common CPUs in desktop computers. Mobile devices, on the other hand, have stringent power requirements: lower power consumption means longer battery life. The dominant chips in the mobile market are ARM processors. ARM manufactures 90 percent of processor cores for cell phones. A host of other mobile devices also run on ARM processors, including many PDAs and Apple Computer's iPod. In 2004, 1.3 billion chips shipped contained ARM cores [35]. Porting Asbestos to ARM will bring operating system security guarantees to a large number of mobile devices.

1.4 Challenges

Several challenges arose in porting Asbestos to the ARM. Not the least of these challenges was that, until now, Asbestos has been a pure x86 operating system. This is the first attempt to port Asbestos to any other architecture. This project had to define a Hardware Abstraction Layer (HAL), a common interface between the kernel and the hardware. This proved difficult because the ARM and x86 processors have different memory management hardware. In particular, x86 Asbestos makes use of unused bits in the hardware page table entries to store extra state. The ARM port had to mimic this behavior, sometimes without extra bit space. This first port of Asbestos also uncovered a number of previously untickled kernel bugs, as also happened frequently during the development of the OK Web Server.

Another challenge in porting Asbestos was choosing a suitable ARM platform for deployment. The ARM architecture has existed for fifteen years and has evolved considerably in that time. There are many generations of ARM processors, and while most support a consistent instruction set, there are small variations between the generations. ARM processors are used in many different configurations of commercial devices, which made it difficult to find ARM simulators that work for a specific hardware model (the HP iPAQ, in this case). In addition, the wide range of ARM devices means open-source drivers are not available for all ARM hardware layouts. Fortunately this project was able to build on code from other ARM ports, such as ARM Linux for the iPAQ, maintained at `www.handhelds.org`.

Finally, there were challenges in designing the system services for ARM Asbestos, namely the file system and the network driver. The Asbestos file system (ASFS) uses little-endian ELF program images. One challenge was maintaining binary compatibility for the file system across architectures, so that program image files may be shared between the ARM and x86 builds. Another challenge was the medium on which the file system is stored. x86 Asbestos can interface with IDE disk drives for persistent storage, or can maintain file system state in RAM. To achieve persistent storage on disk-less ARM architectures, Asbestos requires a Compact Flash driver.

Implementing a new network driver presented yet other challenges. x86 Asbestos supports two wired ethernet network cards: the e1000 and the ne2k. A network interface in a mobile device is guaranteed to be a wireless network card, probably supporting 802.11 Wi-Fi. Adapting to this kind of interface requires a wholly different network driver than the drivers currently in place. Furthermore, the ARM network driver should interface with the LWIP network stack [17], just as x86 Asbestos does. This way, both builds can use `netd`, the user process that implements the network service.

1.5 Approach

The work in this thesis took place in several steps. First, to gain familiarity with the ARM instruction set and hardware, I ported the JOS kernel to ARM. The Asbestos operating system is based on the much simpler JOS operating system used for systems education at MIT, UCLA and NYU [1]. The JOS port involved setting up a simple boot process and interfacing with the serial port to establish a minimal debugging environment. Next I set up a crude memory management system and simple context switching, proving that the basic needs of Asbestos can be met by the ARM processor.

The second step was to merge this early development with the Asbestos mainline. In deciding what system modules the x86 and ARM builds would share, I defined the Asbestos Hardware Abstraction Layer (HAL), a hardware interface that determines what is required of future ports of Asbestos. The merging step ensured that the various versions of Asbestos remain synchronized as the Asbestos security mechanisms and system services develop.

Next I designed the essential services for ARM Asbestos: the file system and network driver. The file system is based largely on x86 ASFS, although file blocks had to be decoupled from x86 pages. In this thesis I discuss the design of a Compact Flash driver and wireless network driver, though their implementation is not finished at the time of writing. The goal of the Compact Flash driver is to utilize permanent storage on mobile devices. The network driver will be based on open-source ARM Linux for iPAQs, and should implement the 802.11 wireless protocol.

Finally I evaluated the port, assessing whether ARM Asbestos' performance is comparable to x86 Asbestos on commodity mobile hardware. I ensure that ARM Asbestos does indeed meet the basic usability requirements of a mobile operating system.

1.6 Outline

The remainder of this thesis is organized as follows. Chapter 2 discusses related work in operating systems and mobile computing. Chapter 3 summarizes the development environment for the ARM Asbestos port. A lot of time on this project was spent gaining understanding of the ARM architecture, and its software and hardware peculiarities. Chapter 4 discusses the design and implementation of the ARM Asbestos kernel. It explores how this port forked from x86 Asbestos and examines the choices made in defining the Asbestos HAL. Chapter 5 considers the design of the file system and network interface. Chapter 6 evaluates the performance of the ARM port by comparing it to x86 Asbestos. Finally, Chapter 7 concludes with a review of the contributions of this thesis, and the work that lies ahead for Asbestos on mobile devices.

Chapter 2

Related Work

The Asbestos operating system was designed for the x86 architecture, without specific regard for future expansion. This thesis aims to run Asbestos on the ARM architecture. Other operating systems have demonstrated the utility of supporting multiple platforms. NetBSD is a Unix-like operating system that also started on the x86, and since 1993 has been ported to seventeen different processors and more than forty platforms [33]. Some operating systems were designed from the start with portability in mind. eCos, a flexible real-time operating system, now supports ten different processor architectures [18]. eCos distances the core OS software from platforms' hardware differences with a Hardware Abstraction Layer (HAL): a programming interface between hardware and software. Porting eCos involves implementing the HAL on a new system, after which the port is guaranteed to work. The Asbestos port achieves platform separation with two sets of architecture-specific code files. In essence the decisions about how to separate source files define a HAL: each new Asbestos port will only have to implement the functions that the ARM port implements.

The ARM port of Asbestos borrows some code from other sources. The volunteers at `http://www.handhelds.org` have made a considerable effort at porting Linux to various ARM iPAQs [22]. The handhelds.org implementations provide examples of how to control iPAQ hardware such as 802.11 wireless cards. The Asbestos port uses the Compaq/HP boot loader to transfer Asbestos images to the iPAQ and boot them. ARM Asbestos takes cues about memory management and context switching from

JOS, the educational operating system on which Asbestos is based. The port uses the same lightweight TCP/IP stack as x86 Asbestos to control the network driver [17].

Asbestos, like JOS, is a microkernel: most of the drivers and system services are pushed out of the kernel into user processes. While often criticized for their lower performance, microkernels can achieve security and reliability goals which monolithic kernels can never hope for. The L4 microkernel [23] separates an operating system from its drivers and applications by running each component in a separate virtual machine. The Minix 3 microkernel [24, 38] exclusively handles process management, scheduling, interrupts and IPC. In user space it has applications layered on top of servers, layered on top of device drivers. The kernel automatically restarts any of these entities in case of a crash or failure. Both of these microkernels achieve reliability by separating critical code from drivers and servers. Asbestos adds strict security guarantees by restricting communications and data sharing between processes.

Many different operating systems support the ARM architecture. Commercial operating systems include Windows CE, Palm OS and Symbian. There have also been efforts to port existing desktop operating systems such as Linux, by companies [31, 30] and groups of volunteers [22]. None of these systems have security as a primary focus, which is cause for some concern. A security evaluation of Windows CE identified all of the traditional security concerns of desktop Windows, as well as new vulnerabilities [16]. The many new interfaces that mobile devices support (wireless networking, Bluetooth connections, infrared ports, and telephony) allow communication with fewer restrictions than traditional Ethernet.

Chapter 3

Development Environment

This chapter will describe the development environment in which the work for this thesis was performed. It will detail the hardware that was available, as well as software that either simulated the hardware or aided in its use. The information may serve as a guide not only to future developers of ARM Asbestos, but also as a reference to anyone developing for the ARM platform.

3.1 Hardware

The port of ARM Asbestos was frequently tested on real hardware, ensuring that the port did not rely on simulator oddities, such as initially zeroed memory. The hardware used was an HP iPAQ pocket computer, model h5555.

3.1.1 The iPAQ h5555

The h5555 is a powerful hand held computer. It contains a 400 MHz PXA255 XScale processor developed by Intel, as well as two 64 MB banks of SDRAM. It also has a 48 MB Compact Flash card inside (not removable) and an SD/MMC card expansion slot. The h5555 boasts an array of wired and wireless interfaces: USB, Bluetooth, Wi-Fi and Infrared. It ships with the Windows Mobile 2003 operating system, the removal of which was obviously the first step in this project. This thesis did not make use of the LCD touchscreen, speaker or microphone. While these are nice features for a production OS, the focus of this thesis was on providing operating system security to a hand held device connected by a wireless interface (Wi-Fi in this case).

3.1.2 Bootldr: A Flash Bootloader

The first step to this project was to remove the production operating system and replace the bootloader with something that could download and boot a custom kernel. I followed the detailed instructions in The Linux iPAQ HOWTO [5] to install a new bootloader named Bootldr. Bootldr was written by developers at Compaq [7] and `www.handhelds.org` [25]. The key functionality of Bootldr is that it allows developers to upload kernel and file system images from a PC to the iPAQ's Compact Flash card. The transfer happens over the serial line via the iPAQ's cradle. Once a recognized image is uploaded (such as a Linux kernel image), Bootldr can copy the OS image into RAM and boot it.

The ARM port takes a small shortcut by booting Asbestos as if it were a Linux kernel. A Linux identifier in the right place in the kernel image will cause Bootldr to start up the Asbestos port. The choice of starting Asbestos as a Linux kernel will ease future deployment onto other Linux-driven ARM devices. Alternate Linux-enabled devices with wireless cards are the Nokia N770 Internet Tablet and several Linux-based smart phones.

There are several annoying limitations to testing Asbestos on iPAQ hardware. First and foremost, Bootldr does not support USB file transfers, though the cradle plugs into the USB port on the iPAQ and has a USB connector for the PC. This means that every test of Asbestos on hardware involves a painstaking transfer over the serial port. The slimmed down version of Asbestos is approximately 2 megabytes large, partly due to the lack of dynamic library linking. At a transfer speed of 115,200 bits per second, uploading such an image to the hardware takes two and a half minutes. This makes for a rather slow development cycle.

Another oddity that slowed the development cycle was the interaction of Bootldr with the h5555's battery charging circuit. The iPAQ's main charge circuit is unfortunately activated in software, for example by Bootldr. When the iPAQ is reset, Bootldr shows a `boot>` prompt and charges. However, the iPAQ does not charge in its cradle when Asbestos runs, or after Bootldr loads an image over the serial port. If the iPAQ remains in one of these states for long, it is possible for the lithium ion battery to get so discharged that after the next reset, it hangs during the boot process and buzzes without stopping. The solution in this case is to leave the battery and iPAQ disconnected for several hours, and then perform a special "hard reset" that wipes all of Flash except for the protected Bootldr partition. To avoid this time-consuming and battery-damaging error state, it is best to always reset the iPAQ after every boot or image load, allowing it to charge in the cradle when not in use.

In spite of these frustrating development quirks, Bootldr is an essential part of iPAQ OS development. Without it, the Asbestos ARM port would hardly have been possible.

3.2 Software

A lot of time in this project was spent trying to discover what software tools were needed. In choosing software tools, I had two goals: speeding up development and maintaining compatibility with x86 Asbestos.

3.2.1 Simulators

In the face of long hardware testing cycles, I searched desperately for good simulators. The first tool tried was Sim-iPAQ, a SimpleScalar-based [4] iPAQ simulator developed in a joint power modeling project between the University of Michigan and the University of Colorado [32]. The project was last updated in 2002, and halfway through the port I discovered that the simulator was laden with bugs and did not support the entire iPAQ architecture. At this point Jamey Hicks, the principal Bootldr developer, advised me to try SkyEye [26], an open-source ARM simulator developed at several universities in China and maintained at `http://www.skyeye.org/`. SkyEye supports several of the latest ARM CPUs and a host of peripherals including timers, UARTs and even LCDs and touch screens. The latest version also allows remote debugging through GDB.

SkyEye is under active development, which is both a benefit and drawback. On the one hand, it is not a finished product: support for the PXA255 processor was just recently added. Before this point I used the SA1110 processor model to test the port. On the other hand, SkyEye has an active developer community that responds to user questions, and new features and bug fixes are constantly being released.

3.2.2 Compilation

At the start of this project, the ARM port was compiled using a prebuilt x86-to-ARM cross-compiler tool chain of GCC 2.95. I soon found it necessary to build my own cross-compiler (arm-elf-gcc 3.4.3) for a variety of reasons. First, the developers of x86 Asbestos began to use advanced compiler directives in their source code. Such directives included nested C structures and unions and the "__builtin_expect" keyword, which allows the compiler to do branch prediction. Second, I needed to specify the architecture I was compiling for. In particular, the PXA255 does not have a floating point unit, so the compiler needs to include software floating point libraries. This is discussed further in Section 3.3.2.

There are many generations of ARM processors with similar instruction sets but wildly varying hardware. The compiler, arm-elf-gcc, lets developers specify the architecture version of the targeted machine. The ARM Asbestos port is compiled for ARM version 4, the oldest and most compatible version of the architecture supported today [3]. More recent ARM processors have instruction sets that are backwards compatible with ARMv4, making it a good target version. The compiler can also generate either big- or little-endian ARM instructions. For compatibility with the little-endian x86, we use the ARM CPU in little-endian mode.

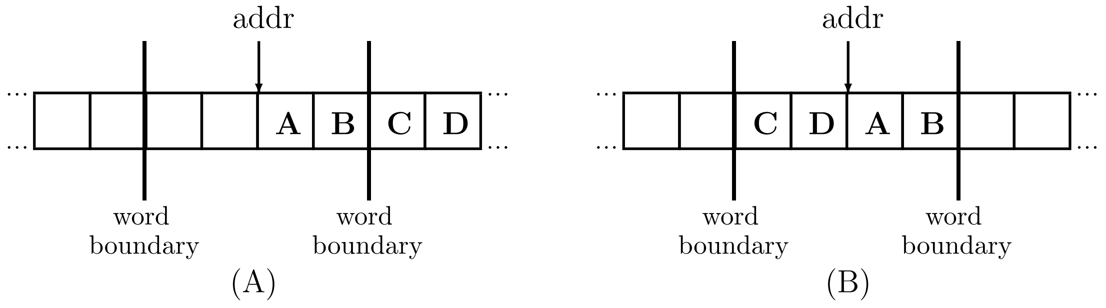

Figure 3-1: Data bytes accessed by a non-aligned address
On x86 (A) bytes are accessed sequentially, whereas on ARM (B) bytes wrap around to the start of an aligned word.

3.3 ARM Platform Peculiarities

In the course of porting Asbestos to ARM, I encountered some distinctive ARM behaviors that cost me many hours to track down. These behaviors often appeared only on hardware, and always caused strange crashes.

3.3.1 Data Alignment

ARM processors have become popular because they are cheap and simple. This is in part because ARM CPUs, like other RISC processors, are optimized to efficiently access aligned data. ARM processors do very well with addresses that are word-aligned (multiples of four). In contrast, the x86 can access data at any byte offset.

ARM CPUs perform an interesting and counterintuitive operation when directed to load or store data at a non-aligned address. For any access that passes a word boundary, ARM processors rotate excess bytes around to the start of the currently accessed word. See Figure 3-1 for an example of a word access performed at a non-word boundary. Unless the compiler is specifically instructed to access data one byte at a time, programs will read wrapped data values when accessing, for example, a 32-bit field at an address that is not word-aligned.

GCC 3.4.3 is relatively clever about aligning data structures so that data words lie on word boundaries. It normally pads structures with extra, unused bits so that

```
struct ok {          struct bad {         struct ok_fix1 {     struct ok_fix2 {
   int32 4byte;         int32 4byte;         int32 4byte;         int32 4byte;
   int16 2byte;         ok 7byte;            int16 2byte;         int16 2byte;
   int8 1byte;          ok fails;            int8 1byte;          int8 1byte;
}                    } __packed           } __packed              int8 padding;
                                                                  }
```

(A) (B) (C) (D)

Figure 3-2: The structure packing problem on ARM
Structure A is automatically padded by the compiler to end on a word-boundary. Structure B is packed, but contains two non-packed instances of A. The second of these, “fails”, will suffer alignment errors. Packing A, which results in Structure C, forces the compiler to generate correct byte-by-byte accesses. Another solution is manually padding A to ensure proper alignment in B, resulting in structure D.

the average developer doesn’t have to worry about alignment problems. When data structures are defined as packed, however, alignment becomes an issue on ARM. The keyword “packed” directs to compiler *not* to insert padding to auto-align fields. Packing a structure on ARM also forces the compiler to generate slow, but correct, byte-by-byte data accesses [2]. In x86 Asbestos, developers packed the File structure to ensure that the compiler would not inflate it beyond its intended 256-byte size. This caused problems for the ARM port, because the data fields of internal C structures ended up unaligned, as Figure 3-2 illustrates. This problem has two solutions. One solution is to declare the internal C structures packed as well, keeping data fields unaligned, but ensuring byte-by-byte accesses. This slows down the accessing code tremendously, requiring four times as many memory accesses for every word-length data field. The alternative, chosen in Asbestos, is to remove the packed attribute from the File structure and its internal structures. Instead, the data fields of all involved structures are padded manually to ensure correct (and speedy) access on ARM.

Modern ARM processors such as the PXA255 contain hardware alignment checking, a useful new development feature that helps prevent alignment errors. By raising a flag to the memory management unit, the programmer can let the hardware throw an alignment fault (including the faulting address) whenever a word-aligned instruc-

Table 3.1: Relative speeds of floating point operations

OS	**FPU?**	**Compile Flag**	**Relative speed**
Gentoo Linux	yes	hard-float	77
ARM Asbestos	no	soft-float	7
ARM Linux	no	hard-float	1

tion is called on a byte-aligned address. This has proved to be extremely useful during the Asbestos port.

3.3.2 Floating Point

Most modern CPUs are bundled with floating point units (FPUs), coprocessors that rapidly perform non-integer arithmetic. ARM developers have several options when it comes to handling floating point math. If the target architecture has an FPU, the compiler can generate special instructions for the hardware unit using the "hard-float" compilation option. Alternatively, developers can use a compiled-in library of optimally coded software floating point routines, included by the compiler flag "soft-float".

Not all ARM ports can use "soft-float", particularly because using this flag changes the ARM calling conventions and thus makes the binaries incompatible with compiled "hard-float" code. The default on ARM Linux has traditionally been to have the compiler use hard-float, even though ARM Linux never ran on any ARM CPU with a hardware FPU. On every FPU operation, this operating system catches an invalid instruction exception in the kernel and emulates it in software. Not only must the kernel perform the FP operation, it also emulates the whole hardware FPU [21].

Table 3.1 shows the relative speeds of the floating point options [27]. Unfortunately, the PXA255 does not come with a floating point coprocessor. However, because it does not depend on a legacy ARM code base, ARM Asbestos can choose the "soft-float" option, which is the optimal solution without an FPU.

3.3.3 Caches & Write Buffers

The ARM processor has a hardware cache and a write buffer which can be separately enabled by raising a flag on the memory management unit. Turning on these hardware features is an optimization, and not essential to the operation of ARM Asbestos. However, principal Bootldr developer Jamey Hicks reported a twofold improvement in performance after enabling ARM hardware caching. Mr. Hicks said that write buffer improvements were less noticeable, so the ARM Asbestos efforts have been focused primarily on enabling the cache. Data caches and instruction caches can be separately enabled on the ARM MMU. On PXA chips the data and instruction caches are separate entities, but on other chips they may be shared. When caching is on and buffering is off, ARM caches act as write-through, meaning data writes immediately pass to memory.

Maintaining cache coherence is a new problem for Asbestos. The x86 features a physically addressed cache: all data is remembered by its physical address. Because all writes to random-access memory are reflected in subsequent reads, it is impossible to get inconsistent cache state on x86. The only caveat is memory-mapped IO, which must always be marked uncacheable. ARM processors feature virtually addressed caches, in which cache data is associated with virtual addresses and cache lookups occur in parallel with virtual address translation. The major problem with caching on ARM is that Asbestos maps all user pages doubly: one virtual address is used in user space, the other is used in kernel space.

Maintaining two cached virtual address lines for one physical memory location can cause cache inconsistencies. One example is the kernel block cache. The block cache fulfills a user write request by copying data to a user page's kernel space address. When a user process (even the one that wrote to the block) performs a read on this cached address, the cache will return stale data. Although the data at the physical address of the block has changed via the kernel virtual address, the user cache line remembers the old data values. In this example, the solution is to invalidate the cache lines at the virtual addresses of the user data, ensuring that all user read requests will be answered by the up-to-date kernel block cache instead of the hardware cache. Cache problems such as these are usually difficult to fix because there is no local solution: if the block cache had access to the user virtual address of the page, then it could perform memory writes to the user address, avoiding the cache consistency problem altogether.

To fix the Asbestos cache problem in general, I make several observations. First, user code only has access to the user space virtual address of a user page: only the kernel (as a whole) has a choice of which address to write to. Second, kernel code is of a restricted size while user code is not: one can fix cache problems in the kernel, but cannot predict what code programmers will run on Asbestos in the future. Finally, it is cleaner to fix cache issues in the operating system than in user applications: application programmers should not be burdened with maintaining cache consistency. With these observations in mind, I decided to handle cache invalidations in the kernel code.

Cache lines only need to be invalidated if they represent user memory; kernel memory only has one virtual address. The kernel has access to all three different addresses representing user data pages. Given a user-space virtual address, it can always discover the kernel-space virtual address as well as the physical address. Because MMU page translation remains on in kernel mode, the kernel will never perform data accesses via a physical address. However, kernel code does use the user-space and kernel-space virtual addresses interchangeably. Access to one physical address through two different (cached) virtual addresses causes cache inconsistencies. To ensure up-to-date cache data, I established the following caching policy. Cache lines representing user data must be invalidated in three cases in the kernel code:

1. Before reading from user memory via a kernel virtual address. After a kernel virtual address first enters the cache, a user program may write to it at its user virtual address. Before a subsequent read, the kernel must *invalidate the cached kernel virtual address* to prevent reading stale user data.

2. After writing to user memory via a kernel virtual address. The user virtual

address representing the physical memory being written may already reside in the cache. In this case it is now stale data, and the kernel must *invalidate the cached user virtual address* to ensure freshness on the next access.

3. On any TLB flush. When a page translation becomes invalid, it means that the cached underlying memory should no longer be readable. The set of TLB invalidations is a subset of the set of cache invalidations.

The implementation of this design in the Asbestos kernel took some effort. Luckily, most user virtual addresses must pass a series of access checks before being dereferenced by Asbestos kernel code. These checks are the perfect place to implement cache invalidations. For details of how enabling the cache affects the performance of ARM Asbestos, see Section 6.2.1.

Chapter 4

Porting the Asbestos Kernel

The goal of this thesis is to make Asbestos usable on hand held devices. While Asbestos was designed for the x86 architecture, most hand held devices run on ARM hardware. The CPU, memory management unit, interrupt controller and timers are all different on ARM than on x86. The operating system had to be adapted to interface with this completely new set of hardware components.

Because the Asbestos kernel must interface with architecture-specific hardware on each platform, it had to be split into sets of architecture-specific and common components. Deciding how to split the kernel was a major challenge. On the one hand, all the new ARM hardware required a lot of new code. On the other hand, it was desirable to keep the split between architectures minimal for two reasons. First, there was pressure from the other members of the Asbestos development team, who continue to work with the x86 code base to develop new applications that take advantage of Asbestos' security features. Second, a minimal architecture fork means that porting Asbestos to other architectures in the future will be simpler. In splitting the kernel into x86, ARM and common components, I defined a Hardware Abstraction Layer (HAL) for Asbestos. The HAL is the boundary between kernel software and architecture hardware. For details of how the source code was split, see Appendix A.

Figure 4-1 shows the Asbestos HAL and the modules it defines. The largest part of the Asbestos HAL is the memory management module. Because memory management is one of the kernel's biggest jobs, there are many references to the

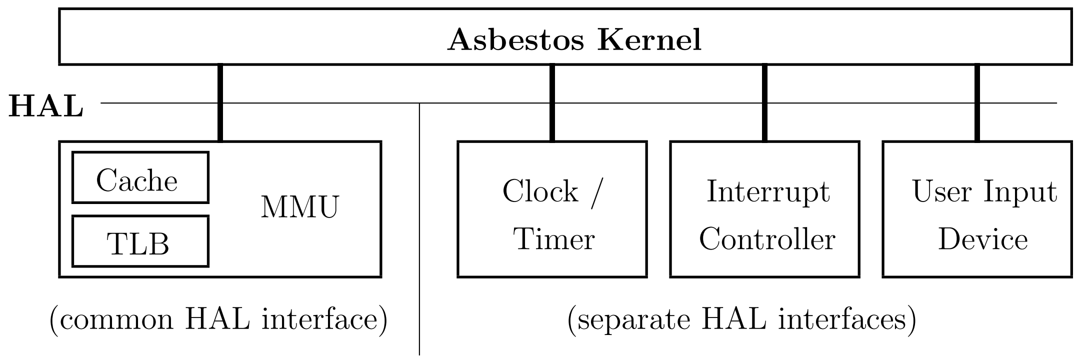

Figure 4-1: The Asbestos HAL

MMU module in both the shared and architecture-independent code. Both x86 and ARM Asbestos use the same, well-defined HAL MMU interface. On the contrary, the other hardware modules are referenced only by architecture-specific trap handling code. The rest of the Asbestos kernel hardly interfaces with the non-MMU hardware modules because Asbestos, as a microkernel, pushes most driver code into user space. As a result, the interfaces to non-MMU modules are not globally defined: x86 code might call `timer_init()`, while ARM code might call `timer_start()`. In this sense, most of the Asbestos HAL is not a list of hardware interfaces, but rather a set of expectations about what functions the hardware should perform. For example, Asbestos expects a timer to raise an interrupt 100 times per second, but there is no common `set_timer_interval()` function. Section 4.4 discusses each hardware module, and how Asbestos expects each module to function.

This chapter describes the design and development of the ARM side of the Asbestos kernel. First it explains how ARM Asbestos boots and performs context switching. Next it describes the memory management design. The chapter ends with a discussion of several smaller hardware modules, and how they were implemented on ARM.

4.1 Booting Asbestos

All the code responsible for booting an operating system is architecture-specific; it must be replaced during a port. One of the tricky designs of the ARM port is the boot code. On x86, segmentation registers map a physical memory address to a linear address, and paging maps a linear address to a virtual address. The Asbestos boot process on x86 uses segmentation registers to fool the kernel startup code into thinking that it is running at a high address in memory. Once the page tables are initialized in the kernel, the segmentation offset is set to 0, and paging alone translates low physical addresses to high virtual addresses.

ARM programmers cannot enjoy the luxury (or rather, confusion) of segmentation. The ARM boot code instead creates a full, but temporary, page table mapping between the physical location of the kernel, PHYSBASE, and its final virtual address, KERNBASE. To ensure that the boot code can continue running in place, there must also be a 1-to-1 mapping from PHYSBASE to PHYSBASE. The boot code then turns on the memory management unit (see Section 4.3), activating page translation. Switching on the MMU does not change the CPU's program counter, so the boot code is still executing at its physical address in memory. The final trick is to jump the program counter to a virtual address above KERNBASE, where the entire kernel will eventually be mapped. This readies the processor for the rest of the C boot code, which will create a new page directory and switch the MMU's page directory from the temporary mapping to the actual Asbestos memory layout.

4.2 Entering and Leaving the Kernel

4.2.1 Context Switches

Asbestos is a message-passing microkernel, which means system services such as the file system and network driver run as their own user processes. Context switching is very important to Asbestos because all interactions between services, as well as process scheduling and system calls, require a jump into and back out of kernel mode.

In x86 Asbestos, the steps below are taken to handle interrupts from user mode. It is assumed here that the example context switch causes a change of environments, but the same steps are taken when the kernel returns control to the same user process that was interrupted.

1. the CPU enters kernel mode
2. process A's entire execution state is saved at the top of the kernel stack
3. the interrupt or system call is handled in the kernel `trap()` function
4. process A's state is copied from the kernel stack to A's environment structure
5. process B's execution state is loaded from B's environment structure
6. the CPU enters user mode, running process B

For consistency and clarity, these steps are duplicated in the ARM port, including the somewhat wasteful double copy of process A's execution state. On ARM, however, there is one added step. The ARM processor understands a set of interrupts and exceptions that does not map 1-to-1 to its set of privileged execution modes (see Table 4.1). For example, both Prefetch and Data interrupts will cause the processor to enter Abort mode. Furthermore, any software interrupt instruction, whether intended as a system call or a breakpoint, triggers Supervisor mode. It is for this reason that ARM Asbestos includes a step 2.5 that pushes an extra argument to the `trap()` function. This argument describes the type of system call (by investigating the last executed instruction in user space) or the type of abort that has occurred (from the type of exception thrown).

Another interesting difference between x86 and ARM is how the kernel knows to get from step 1 to step 2. x86 uses an interrupt descriptor table, initialized by Asbestos on system startup, that maps exceptions and interrupts to kernel entry points. On ARM, this table's address is unfortunately hard-wired into the processor. All interrupts and exceptions cause the program counter to jump to a virtual address in the range 0x0-0x28. At runtime, the ARM port maps this virtual address range

Table 4.1: ARM mapping between exceptions and resulting execution modes

Exception	**Execution Mode**	**Extra Info Needed**
System reset	Supervisor	(none)
Software interrupt (SWI)	Supervisor	System call type
Prefetch Abort	Abort	Prefetch or Data?
Data Abort	Abort	Prefetch or Data?
Interrupt Request	IRQ	(none)
Fast Interrupt Request	FIQ	(none)
Undefined instruction	Undefined	(none)

to a set of instructions that load the program counter with the interrupt handlers described in step 2.

The multiplicity of execution modes on the ARM creates some challenges regarding context switches. On x86, there are just two modes: privileged (kernel) and non-privileged (user). In Asbestos, these execution modes have separate stacks: a kernel stack in kernel memory and a user stack in user memory. On every abort, IRQ and system call from user mode, Asbestos clobbers the stack pointer, setting it to the top of the kernel stack. This resetting of the kernel stack is necessary because sometimes kernel code returns from the trap handler by using the function `env_run()`, which restores user execution state from a trap frame but does not unroll (pop) the stack frames of the trap handler.

On ARM, all the execution modes have separate physical registers for stack pointers. On traps from user mode Asbestos always clobbers the stack pointer for whatever privileged mode it's in to the top of the kernel stack. The problem is that Asbestos also takes two types of interrupts within kernel mode: clock IRQs to configure timers during boot, and page faults to handle copy-on-write pages. In these cases, the trap handler must avoid clobbering existing data on the kernel stack. When x86 Asbestos takes a trap from kernel mode, it layers the trap frame on top of the kernel stack, handles the trap, and then pops the trap frame without switching stack pointers. Unfortunately, on ARM the execution mode switch also forces a switch to the new mode's stack pointer. It is not easy to load the stack pointers from other execution modes, so layering stack frames becomes cumbersome. There are three ways to handle this kernel trap problem:

1. Make all privileged execution modes use the same kernel stack. Just after the trap, quickly enter Supervisor mode. Store away the Supervisor stack pointer to a known location, and return to the previous execution mode. Load the Supervisor stack pointer and proceed as on x86, building a trap frame on top of the existing kernel stack.

2. Always use Supervisor mode to handle exceptions. On traps, save the current execution mode's banked hardware registers to a known location. Enter Supervisor mode and load the saved hardware registers, except for the stack pointer (use the Supervisor stack as the only kernel stack).

3. Use separate exception stacks for each privileged execution mode. On user traps, always clobber the stack pointer to point to the kernel stack, regardless of the execution mode. On kernel traps, always clobber the stack pointer to point to a separate execution stack for each mode (IRQ, Abort, etc).

Each of these solutions has advantages and drawbacks. Solutions 1 and 2 implement a shared kernel stack, requiring that recursive exceptions always clean up their stack state (never use `env_run()` to return). On the other hand, they both allow fully recursive kernel exceptions. Solution 3 does not require exception handlers to clean up their stack state, because separate stacks are used. On the other hand, it allows a layering of exceptions (one for each execution mode), but does not allow fully recursive exceptions. Currently, fully recursive faults are not required in Asbestos, and kernel exceptions (clock IRQs and COW page faults) always clean up their stack state. For the moment, then, the choice is arbitrary. I have chosen solution 3: separate exception stacks.

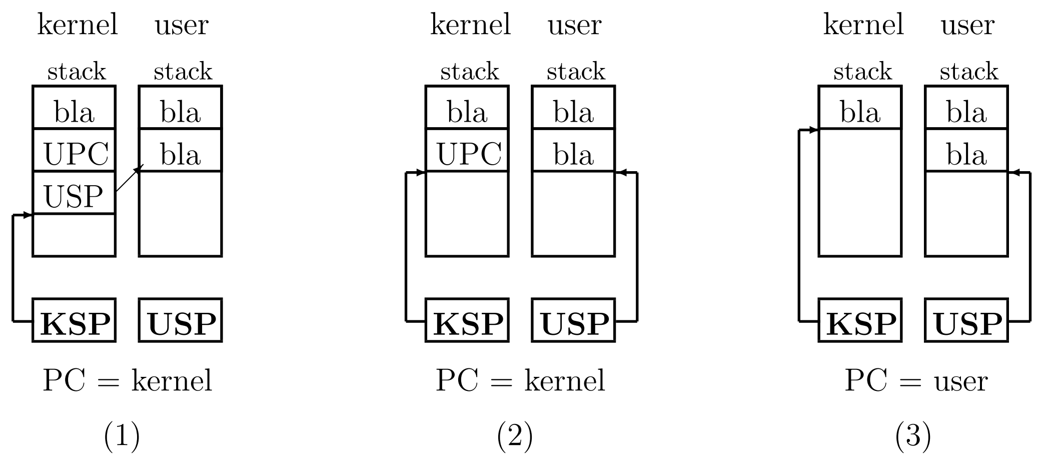

Figure 4-2: An ARM context switch from kernel to user mode
The kernel trap handler (1) restores the user stack pointer into the user mode stack register (2), then jumps the program counter into user mode (3).

4.2.2 User Page Fault Handler

JOS, the educational operating system on which Asbestos is based, allows user programs to handle their own page faults. This lets user programs dynamically grow their own stacks, allocating new stack pages as space is required. Page faults can even occur recursively, from within the page fault handler! Asbestos adopted this same page fault handling method, which demanded some attention during the ARM port.

Handling page faults in user space is more difficult than handling them in the kernel. When a page fault occurs in a user-accessible address range, the kernel sets up a special user-space exception stack and passes control to the user page fault handler. Transferring control to the handler is as easy as transferring control to any user process. However, restoring the user program's trap-time state from the user page fault handler is difficult.

Normally, Asbestos transfers control to a user program from the kernel, which operates in Supervisor mode. Transferring control is doable because ARM provides separate registers in each execution mode for the stack pointer, link register and processor flags. This means one can load the user stack pointer into the user register,

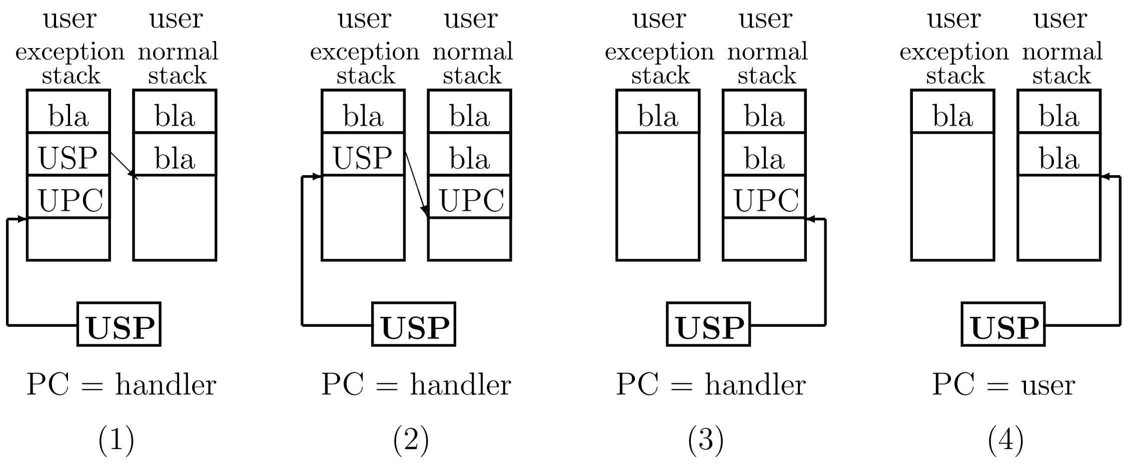

Figure 4-3: An ARM context switch from user to user mode
The user trap handler (1) pushes the trap-time program counter onto the normal user stack, also updating the trap-time user stack pointer (2). Then the trap handler restores the user stack pointer into the user stack register (3), and pops the trap-time program counter, returning to the user program (4).

and then load the user program counter from the unchanged kernel stack pointer (see Figure 4-2). The luxury of operating with separate stack pointers is lost when transferring from user to user mode, yet the trap handler must restore both the trap-time stack pointer and program counter. The solution is to push the trap-time program counter onto the normal (trap-time) user stack. One can then switch the user stack pointer from the user exception stack to the normal user process stack, and finally restore the trap-time program counter from that stack. Figure 4-3 shows the details of this operation, which x86 Asbestos performs in a similar manner.

The user page fault handling procedure requires saving the trap-time program counter and stack pointer onto the user exception stack. As on x86, the caller-saved registers also needed to be saved, because the user page fault handler might clobber these registers. A caveat on the ARM processor is register R12, also called IP. R12 is a scratch register, most commonly used in function prologues to keep track of a function's stack frame. This means R12 is written on function entry, but not restored on function exit! Ipso facto, R12 must be saved on all context switches, or else page faults may corrupt the executing program's state.

4.2.3 Passing Arguments

The ARM C calling convention differs a bit from the x86 calling convention. The most notable difference is that the first four arguments to a C function are held in the caller-saved registers R0 through R3. Any further arguments are saved on the stack. Return values are saved in register R0 (and R1, if the value is 64-bit). Compiled C code automatically adheres to this calling convention, but the differences becomes important when assembly code calls C code. During boot or when a new process is started with a `fork()` or `spawn()` call, the code that sets up the new environment must take care to pass initial arguments in registers.

4.3 Memory Management Module

The memory management module is one of the larger chunks of the Asbestos operating system. Unfortunately, it is also linked deeply to the x86 hardware memory management unit (MMU). It would be best to define a common memory interface as part of a cross-architecture Hardware Abstraction Layer (HAL), and to implement memory management independently in the ARM port. However, the goal of keeping a minimal fork for the ARM port required some sacrifices in the memory module. The memory management code was separated into two parts. First, a shared page tracking module maps pages into environments and keeps reference counts for all pages. Second, a separate page table module defines functions to access the MMU, setting hardware and software flags on the page tables. The page table interface is the most well-defined part of the Asbestos HAL, and it is described in Section 4.3.3.

4.3.1 Page Table Entries

There are considerable differences between ARM and x86 page tables. On both architectures, a virtual address consists of three parts: a page directory index, a page table index, and a page offset. The page directory index and page table index are offsets into first- and second-level tables of addresses. The page offset indexes

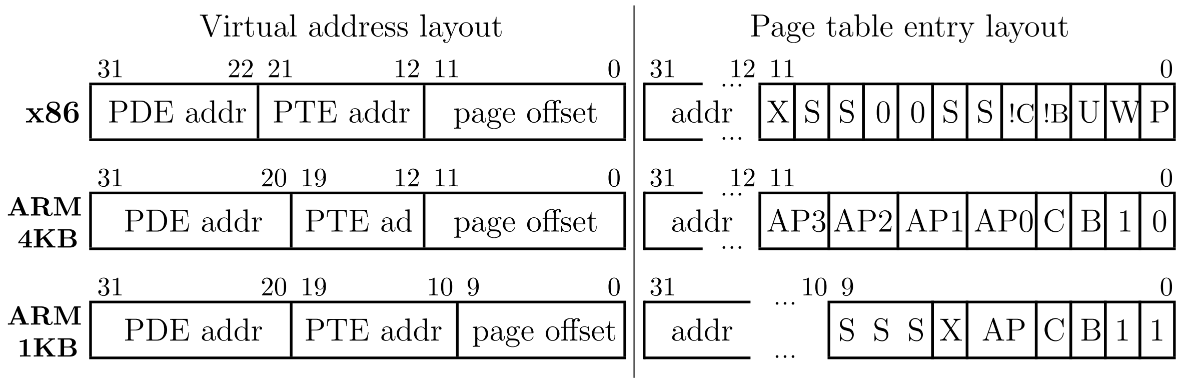

Figure 4-4: Virtual addresses and page table entries
These are the layouts for x86 (top), ARM with 4 KB pages (middle) and ARM with 1 KB pages (bottom). The PTE bits are as follows: P = present, AP = access permissions, W = write, U = user, C = cacheable, B = bufferable, S = free for software use, X = software COW.

into a memory page itself. The respective sizes of these three fields determine a lot about the paging mechanism. On x86, the 32 bits of a virtual address are divided as 10:10:12 (see Figure 4-4). This means that pages are 2^{12} or 4096 bytes in size. Each page directory and each page table stores 4 x 2^{10} or 4096 bytes of data, which fits conveniently within one page. It also means that each page table entry stores 20 bits of a page's memory address, and 12 bits of meta data about a page. Asbestos makes ample use of this meta data space, storing both hardware flags (present, user, write) and software flags (copy-on-write, dirty).

ARM gives programmers a choice for the layout of their page tables. ARM allows for large (64 KB), small (4 KB), or tiny (1 KB) pages. ARM's 4 KB pages are divided as 12:8:12. This ARM division stores more data in page directories than the x86 (4 x 2^{12} or 16384 bytes) and less data in page tables (4 x 2^{8} or 1024 bytes). This means page directories require four aligned, contiguous memory pages, and page tables will only use up one fourth of a page. This is less convenient than the x86 configuration, in which every page directory and page table takes up one page. Furthermore, page table entries in this MMU configuration do not contain any bit space for implementation-specific (software) flags, should they be needed.

The alternative to 4 KB pages are ARM's tiny 1 KB pages. These do allow a few bits of space in the page table entries. Using the tiny layout, the 32 bits of a virtual address are divided as 12:10:10. This means that pages are 2^{10} or 1024 bytes in size. It also means that each page table entry stores 22 bits of a page's memory address, and 10 bits of meta data about a page. Page directories now take up sixteen aligned and contiguous pages, whereas page tables require four tiny pages of space.

x86 Asbestos uses page table entries to store two software flags: copy-on-write and dirty. In the ARM port, the use of software PTE flags can be completely avoided. The dirty bit is no longer used in the latest version of the file system, and the software COW bit can be implemented in hardware, as we'll see later in this section. This means I can decide between 4 KB pages and 1 KB pages in ARM Asbestos, which is a significant design choice. Using 4 KB pages wastes memory, because Asbestos grants memory at the granularity of pages and 4 KB page tables are 1 KB in size. The downside to tiny pages is that they incur more memory management overhead than small pages, for equally sized data structures and files. On the other hand, mobile ARM devices usually have less memory to use than desktop computers, and in this context tiny pages seem reasonable. In Section 6.2.2 I examine both paging configurations and see which is more efficient. If at some future point ARM Asbestos requires the use of software PTE flags, programmers will want to use 1 KB pages. This avoids the use of shadow page tables to keep track of software flags. Such structures are not only cumbersome to program, they require additional memory as well. Switching between the paging layouts is as easy as flipping a flag in the source code.

4.3.2 Page Directory Entries

On x86, the memory management unit checks page permissions twice on every memory access: once in the page directory entry and once in the page table entry. It is thus possible to restrict page permissions for an entire page table by changing the permission bits on a page directory entry. The Asbestos memory module uses this hardware feature to quickly mark an entire page table copy-on-write (not writable), usually when forking a process.

During a normal process fork, the memory pages of the parent process are copied into the child process. This is CPU-intensive and can be wasteful if neither the parent nor the child performs many memory writes. As a fork optimization, Asbestos avoids copying memory. It remaps user memory pages and marks them copy-on-write, meaning they can be read as usual, but must be copied when a write occurs. On x86, this permission is implemented by stripping write permissions at the page directory level and raising a software-defined copy-on-write flag in the PDE. When a write occurs, the MMU generates a fault, and the page fault handler recognizes the COW flag and performs the page copy, giving back write permissions. Note that while removing the write flag from an x86 PDE restricts an entire page table, setting a PDE's write flag does not give write access to all its sub pages (access is checked also at the PTE).

On ARM, the MMU hardware does not perform read/write permission checks at the page directory entry level. There is an analogous entity on ARM called a domain, which can completely restrict a page table from non-kernel access. This is useful for shielding kernel memory from user processes, but Asbestos can't use it to implement the copy-on-write permission (which should still allow reads). The best solution would be to remove Asbestos' reliance on page directory entry permission bits. However, in the interest of a minimal fork, the ARM design solution is as follows. When removing the write permission from a PDE (when setting it COW), the ARM build will remove the write permission from every page table entry covered by that page directory entry. This ARM workaround is unfortunate for performance, because setting a page directory copy-on-write now requires 257 memory writes instead of one (or 1025, for 1 KB ARM pages). Setting a PDE writable requires no extra work, because the ARM's behavior is just like the x86's: it does not restrict the page table's pages.

4.3.3 Defining the MMU HAL

Defining a common HAL for two different sets of paging hardware was not straightforward. x86 and ARM use different mechanisms for restricting access to paged

Table 4.2: Asbestos page table permission mappings

HAL	**User**	**x86 flags**	**ARM flags**	**ARM Kernel**
(kernel page)	none	pres/write	pres/nouser	read/write
PAGE_NONE	none	pres	pres/nouser	read/write
PAGE_READ	read	pres/user	pres/readonly	read/write
PAGE_COW	read	pres/user/cow	pres/defer/cow	read
PAGE_WRITE	read/write	pres/user/write	pres/readwrite	read/write

memory. Table 4.2 shows the hardware flags each architecture uses. I defined a set of common user permission concepts that are understood by both MMUs and sufficient to express the memory permissions needed by the Asbestos memory management module. These permission concepts (PAGE_NONE, PAGE_READ, PAGE_COW and PAGE_WRITE) describe the permissions that kernel and user code have with respect to a page.

In Asbestos, all user pages are mapped twice: once into the kernel address space, and once into user address space. These mappings have different permissions. On x86, the kernel is constrained from writing to user-space virtual addresses in the same way that user code is constrained: it needs the write flag to write. x86 Asbestos uses this fact to evoke copy-on-write faults within the kernel. ARM Asbestos must emulate this behavior.

On version 4 and 5 ARM processors such as the PXA255, kernel code usually enjoys full write permissions on all addresses even though user permissions may be restricted. This will not do for the copy-on-write permission, because Asbestos expects COW faults for user pages to occur from user code as well as kernel code. The COW permission must therefore restrict kernel write access to user-mapped pages. To implement the COW permission, ARM Asbestos uses a hardware workaround that is deprecated for modern ARM processors (version 6 and above). The page table access permission flags are set to defer to system-wide MMU permissions flags (the S and R bits), which can restrict kernel write access. These system-wide permission flags are initialized to allow kernel and user read access. The ARM hardware interface is detailed in Table 4.3. This technique for restricting kernel access stretches the ca-

Table 4.3: ARMv4 and ARMv5 access permission bits

HAL	**AP**	**S**	**R**	**Kernel**	**User**
	0b00	0	0	no access	no access
	0b00	1	0	read	no access
PAGE_COW	0b00	0	1	read	read
	0b00	1	1	(undefined)	(undefined)
PAGE_NONE	0b01	x	x	read/write	no access
PAGE_READ	0b10	x	x	read/write	read
PAGE_WRITE	0b11	x	x	read/write	read/write

pabilities of version 4 and 5 ARM hardware, but luckily the COW permission is the only one that must force the kernel to fault.

In order to standardize the page table permission interface described above, I laid out a set of processor-specific macros that implement the MMU HAL. This part of the HAL is very well defined: it consists of macros and variables that describe the page translation process, and a set of kernel-called macros that manipulate PDEs and PTEs. A port must implement the following macros at the single-page level (PTEs) and one other paging level (PDEs):

`ISP()`	- is entry present?
`ISPU()`	- is entry present and user read-or-writable?
`ISW()`	- is entry writable or COW?
`ISWW()`	- is entry writable?
`ISCOW()`	- is entry COW?
`MKWW()`	- allow write ability for entry (and all sub pages)
`MKCOW()`	- restrict write ability for entry (and all sub pages)

The implementation of these macros is different on x86 and ARM. On ARM, for example, the `MKCOW()` macro for the higher level entry (PDE) must recurse over all pages in its page table to restrict them, whereas on x86 simply stripping the PDE write bit suffices.

4.4 Hardware Modules

This section describes the various hardware modules used by the kernel, and how they were adapted to fit the ARM hardware platform. The ARM platform is only about fifteen years old, and as a result it has fewer legacy hardware oddities than the x86. I have found it easier to understand and program ARM hardware than x86 hardware.

4.4.1 Serial Port

x86 Asbestos sends all output onto the CGA console. While there is support for serial port communication, this form of output is less important because there are many x86 emulators that readily display CGA output [8, 6]. It is difficult to find such emulators for the iPAQ h5555, as this device is most often programmed and tested directly on hardware by uploading binaries via a USB or serial cable. Serial port output has proved to be invaluable in debugging ARM Asbestos. Printing to the serial port is not difficult: it merely requires writing to a memory-mapped register and waiting on an acknowledge signal in a status register. Reading from the serial port is almost as simple, except that Asbestos takes interrupts when data is ready on the serial port. This allows user to interrupt a running program with a keystroke, for example. Setting up ARM serial port interrupts requires configuring some hardware data buffers and unmasking the serial port IRQ on the interrupt controller. This is all well documented, save for the flag that actually connects the UART to the interrupt controller on the PXA255.

4.4.2 Timers

The x86 and ARM architectures have different clock hardware. The x86 configuration has one 1.19 MHz clock that can connect to an IRQ interrupt. In x86 Asbestos, the clock is programmed to interrupt the kernel with an IRQ 100 times per second. ARM hardware provides a real-time clock unit, as well as 4 separate timers which oscillate at 3.6864 MHz. These are all connected to separate interrupt channels, which can be individually unmasked (see Section 4.4.3). The ARM code has been modeled after the x86 code: it unmasks the timer 0 interrupt and triggers it 100 times per second. The ARM port uses the same calibration code as the x86 to determine how many idle code loops it takes to emulate one clock tick.

4.4.3 Interrupts

The x86 interrupt controller is complicated because it is old. There is a set of eight master interrupt channels, which can be masked and unmasked with a series of PCI bus commands. To allow for more interrupts, one of the master channels nowadays serves the eight channels of a slave interrupt controller. All this makes for a confusing configuration. The ARM interrupt controller is much simpler. There are 32 interrupt channels, controlled together by writing 32-bit values to slots of memory-mapped IO. Each interrupt has a mask bit to turn it on or off, and a bit to switch between regular interrupts (IRQ) and fast interrupts (FIQ). To maintain consistency with the x86 build, the ARM port only uses regular interrupts.

4.4.4 Memory-mapped IO

x86 hardware code is rife with `inb()` and `outb()` function calls. These built-in functions read from and write to the PCI bus, enabling communication between the operating system and the hardware. They are noticeably absent on the ARM. ARM uses memory-mapped IO for nearly all hardware communications. Instead of specifying a bus address, the programmer specifies a virtual address (translated to a physical address) and writes to or reads from this address a 32-bit value. ARM Asbestos remaps all physical IO addresses to virtual addresses in kernel space, ensuring that user programs cannot directly access system IO. This maintains the security guarantees of Asbestos, because the kernel can monitor the communications of user programs. When memory-mapped IO is mapped into the kernel, the virtual addresses should not be marked cacheable or bufferable. The hardware registers they represent, unlike memory values, are bound to change independently of the operating system's actions.

There are more than ten generations of ARM processors in circulation. While

ARM processors all share more or less the same instruction set, many ARM architectures have different configurations of memory-mapped IO. Still, it is desirable to support multiple ARM platforms with one build. For example, during development I ran a crude simulator of the StrongArm SA1110 (Assabet) platform, although the target iPAQ h5555 is an XScale PXA255 platform. Of course, the operating system can detect at runtime which processor architecture it's running on. It was necessary to create a separate processor module within the ARM build that discovers the IO addresses for the running platform. On boot up, this module determines what processor it's running on and stores the appropriate IO addresses in a special MAP structure. A developer may then reference this MAP structure to determine, for example, the current processor's serial port data address: MAP→SERIAL_DATA.

Performing processor detection at runtime prevents Asbestos developers from having to recompile the operating system for a new platform. Simply swapping out IO addresses may not be sufficient to achieve compatibility for all future ARM deployments, especially if the hardware should change dramatically. Fortunately, most current ARM chipsets have a consistent hardware interface, and thus runtime processor detection increases the portability of ARM Asbestos code.

Chapter 5

System Services

The previous chapter explored the details of porting the Asbestos core, namely the memory management and context switching mechanisms. These are important elements, but what is an operating system without a file system? And what is a mobile device without a network connection? This chapter details the development of these system services.

5.1 File System

Because Asbestos is a microkernel, the Asbestos file system (ASFS) is a user-level process. ASFS serves requests from other user programs by receiving and replying to request messages. The ASFS user process resides on top of a kernel-space block cache, which calls a driver to fetch blocks from storage. The driver fills pages with file block data, maps them into user space, and returns control to the ASFS server process.

5.1.1 Compact Flash Driver

x86 Asbestos allows two different disk configurations. A bit of detection code at boot up determines what type of storage is available. Most conveniently, there is an IDE hard disk driver that serves blocks from a real or emulated hard disk. The second option is a RAMDISK driver that serves blocks from a file system image embedded within the kernel image. Using the RAMDISK means the kernel image is bloated with user program binaries, an unfortunate side effect considering the iPAQ's long upload times. Furthermore, the RAMDISK is a static binary image, so ASFS does not allow writes to the RAMDISK file system. The RAMDISK offers readable data, but does not allow persistent storage.

The iPAQ does not have an IDE disk; its primary storage device is a Compact Flash card. To enable persistent storage on ARM, a driver for this medium is necessary. Luckily, the Compaq Bootldr uses Flash memory to store the kernel image, and thus it contains driver code. This standalone code can easily be adapted to Asbestos, where it will work in a similar manner to the x86 IDE driver. The only other work that has to be done is to create a file system partition on Flash, using the interface provided by Bootldr.

5.1.2 Separating Pages and Blocks

On x86 Asbestos, pages and disk blocks are conveniently the same size: 4096 bytes. On ARM, I wanted to keep the block size at 4096 bytes, even though ARM memory pages may be 1024 or 4096 bytes. I had this goal for two reasons. First and foremost, it is convenient to maintain file system binary compatibility across ports. File blocks should be completely independent from operating system pages, so that x86 Asbestos can read ARM files, and vice versa. Additional reasons for keeping the block size at 4096 bytes are that 1024-byte blocks would impose limitations on file sizes and ASFS would incur extra overhead when traversing the file system.

One problem during the port was untangling ASFS's dependence on page size. In the original Asbestos, shortcuts were taken because PGSIZE and BLKSIZE were the same. Undoing these shortcuts was one task. For example, for every read from the block cache, ARM Asbestos with 1 KB pages must loop to allocate and load four memory pages instead of one. Another challenge was restructuring the Asbestos file structure. Previously, as a remnant from JOS, Asbestos used a file structure with a single layer of indirect block pointers (see Figure 5-1A). A file contained pointers to

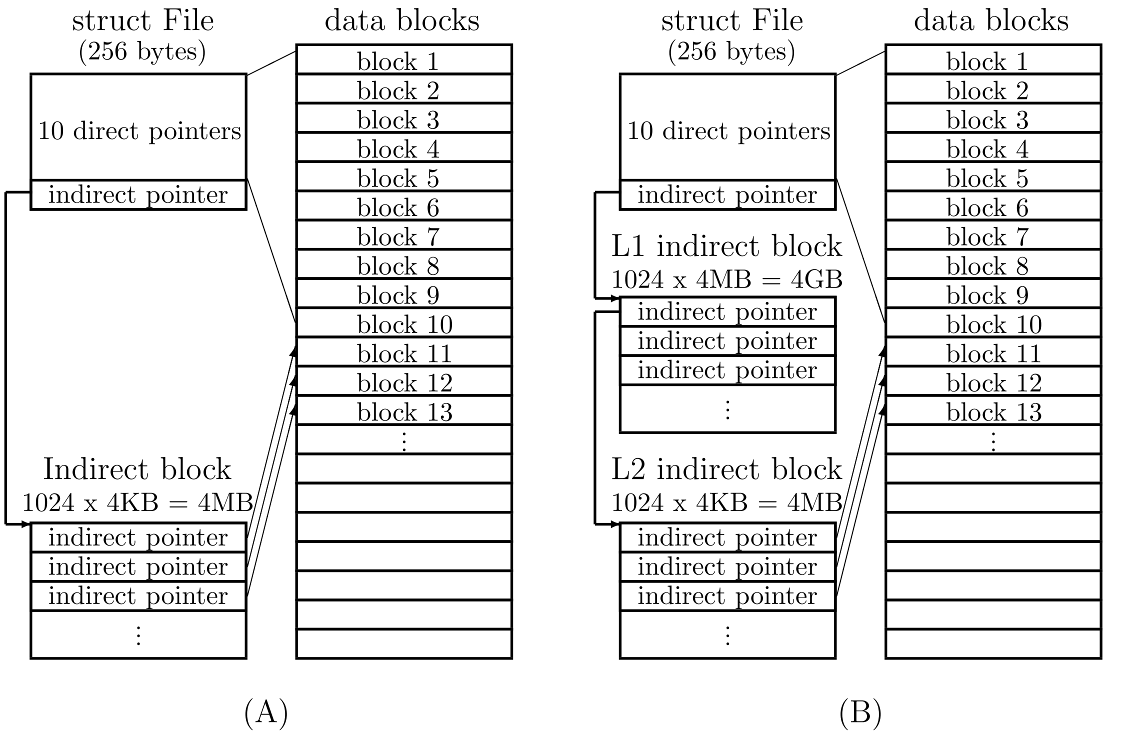

Figure 5-1: Old and new Asbestos file structures
In the old design (A), files greater than 40 KB used a single indirect block to point to 1024 other file blocks. In the new design (B) there are two layers of indirection, allowing a maximum file size of 4 GB rather than 4 MB.

the block numbers of the first ten blocks of the file, called the file's direct blocks. For small files up to 10 x 4 KB = 40 KB in size, this means that the block numbers of all of the file's blocks fit directly within the file structure. For larger files, however, a place was needed to hold the rest of the file's block numbers. For this purpose Asbestos allocated an additional disk block, called the file's indirect block, to hold up to 4096 / 4 = 1024 additional block numbers. This old file structure allowed files to be up to 1024 blocks, or four megabytes, in size [1].

The ARM port was a good occasion to fix the 4-megabyte file size limitation and implement double-indirect blocks. Now, the first 10 blocks are still referenced with direct block pointers, but all subsequent blocks are referenced via two layers of indirection. While ASFS may incur a bit more overhead for files larger than 40KB, the new file layout does allow Asbestos to store files up to a size of 1024*1024 blocks, or 4 Gigabytes (see Figure 5-1B).

5.2 Network

Asbestos is a microkernel, so most of its network stack is handled by a user process. Connections, the TCP/IP stack, and everything else down to and including the packet level resides in user space. A user daemon called `netd` serves as a network interface to other user processes. The code that does reside in the kernel, the network card-specific drivers, is of interest in the ARM port.

x86 Asbestos supports two types of wired network cards: e1000 and ne2k. ARM Asbestos is designed for mobile devices and must support a wireless interface instead. The iPAQ h5555 contains an 802.11-capable wireless network card. Unfortunately, this card connects to the main board via the Atmel AT76C503A Media Access Controller, over USB. While drivers exist for this device, it is a highly nonstandard interface, and any work in porting it to Asbestos would be lost on different devices than the h5xxx iPAQ series. There is an alternative, however. There are PCMCIA sleeves available for any iPAQ. ARM Asbestos will use this interface with an Orinoco-based 802.11 Wi-Fi card.

5.2.1 PCMCIA

The ARM Linux PCMCIA drivers are complicated. They are deeply linked to the Linux kernel and interface with both Compact Flash and Wi-Fi PCMCIA cards. PCMCIA allows a host of features, such as automatic card detection and on-the-fly loading of device drivers. I chose to forego these complicated PCMCIA mechanisms, and focus on the following operations:

1. To discover how the card will interface with PCMCIA, Asbestos probes its meta data, called the Card Information Structure (CIS). The CIS has information about what voltage the card should be given and what interfaces the card uses.

2. Using the CIS data, Asbestos establishes several interrupt channels from the card to the kernel. PCMCIA interrupts go through the General-Purpose IO unit (GPIO), which must be configured to pass on interrupts of the right type from the PCMCIA slot.

3. Finally, Asbestos remaps the physical address of the PCMCIA memory-mapped IO to a virtual address in kernel mode. Using this memory the driver will read incoming data packets and write outgoing packets, as well as control the operation of the wireless card.

5.2.2 The Orinoco Wireless Driver

ARM Asbestos will implement driver code for one specific kind of wireless card. I chose the Orinoco family of wireless cards. These cards are documented as working on handhelds.org's version of ARM Linux. This means the driver code is functional and readily available. Also, the Orinoco driver actually supports a wide range of 802.11b wireless network cards, several of which were available in my lab.

Adapting the Orinoco wireless driver to Asbestos is not trivial. A driver port from Linux to Asbestos involves replacing data types, memory accessors, and locking mechanisms. Linux network drivers also rely on Linux networking tools such as `ethtool`, Linux sockets, and the `net_device` structure. Fortunately x86 developers already adapted these mechanisms to Asbestos. Some other parts of the driver are not yet present on Asbestos. The Orinoco wireless driver uses the 802.11 IEEE wireless standard, of course, and a Linux tool called Wireless Extensions. Source code for these tools will be incorporated into the Asbestos framework, which should then more easily support the addition of other wireless drivers.

Chapter 6

Evaluation

This chapter evaluates the performance and usability of the ARM Asbestos port. A variety of operating system benchmarks have been measured on ARM and x86. I attempt to compare the data to evaluate the benefits of caching, to contrast memory page sizes on ARM, and to ensure that the ARM build demonstrates acceptable performance. Finally I abstractly evaluate the usability of the ARM port in terms of code cleanliness and future portability.

6.1 Measuring Performance

Performance monitoring is a feature of the x86 processor. Each x86 CPU has a 64-bit time stamp counter that counts processor cycles that have elapsed since system startup. A single instruction, `rdtsc`, returns the value of this counter. The ARM, being a RISC processor, does not have a standard cycle counter. One might achieve the same result on ARM CPUs by reading the OS timers, dividing by clock frequency to get elapsed time, and then multiplying by CPU clock speed. This technique gives benchmarks that are further from instruction granularity, because the standard ARM timer only pulses once every 100 or so cycles. One must also watch for looping timers and take extra interrupts when they overflow. Luckily, the PXA architecture has a performance monitoring coprocessor (PMC), coprocessor 14. Not only does the PMC track processor cycles, it can also track a variety of instruction, cache and TLB statistics, such as cache miss rates and a count of executed instructions.

One downside of the ARM PMC is that it is only accessible from a privileged execution mode, unlike the x86 time stamp counter. To allow user benchmarking programs to read the CPU cycle count, the ARM build handles a special software interrupt. This interrupt's handler accesses the PMC in kernel mode and returns the current 64-bit value to the user program. The implication is that user-mode profiling data may be off by several hundred cycles, but simple repetition of benchmarks can make this overhead negligible.

Although processor cycle counts were measured on the ARM and the x86, it turns out these metrics are incomparable. First of all, the processors have completely different instruction sets. ARM's instruction set is much smaller, and thus it may have to execute more instructions to perform the same operations as x86. For example, when accessing an unaligned data word, ARM may perform four byte read instructions while x86 just needs one read instruction. Furthermore, each processor is very differently optimized to perform instructions. Pipelining, branch prediction and even multithreading may contribute to each CPU's attainable cycles per instruction (CPI). These intricacies are much too complex to measure or simulate. I included x86 cycle counts in the results for reference, using them also to compare performance across benchmarks. The cycle counts help ensure that no one test suffers unacceptable overhead compared to other tests, but the absolute cycle counts mean very little.

6.2 Results

This section examines the actual performance data gathered from Asbestos benchmarks run on x86 and ARM hardware. First I evaluate the effect of caching on ARM Asbestos. Next I benchmark a set of standard OS operations, including system calls and memory accesses. As a final sanity check, I examine the measurements from the ARM performance monitoring coprocessor. The ARM hardware was the same HP h5555 iPAQ used for development. The x86 hardware was a Dell Optiplex GX280 with a 2.8GHz Pentium4 and 1 GB of PC3200 DDR SDRAM.

Table 6.1: Kernel label operation benchmarks in thousands of cycles.

Operation	**x86**	**ARM**		
		Cache	**No cache**	**Cache factor**
Create	0.434	5.763	10.929	1.896
Less than	5.186	99.572	220.982	2.219
Add	1.072	17.850	29.688	1.663
Minimum	16.918	297.879	663.986	2.229
Maximum	18.965	345.543	712.116	2.060

6.2.1 Caching

The Asbestos operating system uses primitives called labels to enforce its security policies. Each label can have hundreds of Asbestos handles: unique identifiers tagged at different permission levels. Asbestos' most basic security operations are building up and comparing process labels. Table 6.1 contains benchmarks of the Asbestos label operations, taken in the kernel. These results represent operations on labels with several hundred handles, averaged over 100 repetitions. Results from this test can provide a sanity check for Asbestos' security primitives. These benchmarks are not context-switch intensive, and thus also provide a good estimate of the effect of caching on ARM Asbestos.

The cycle counts on x86 and ARM are consistently scaled across the label tests. The ARM cycle counts for each benchmark are around 18 times higher than those on the x86. This is good news: the ARM does not add unbearable overhead to any one operation. On the other hand, these cycle count ratios are higher than the average ratio achieved in the general OS benchmarks in Section 6.2.2. I can offer one good reason why label benchmarks have high cycle counts: label operations involve lots of memory accesses. In Section 6.2.3 we will see that memory accesses require many cycles on the PXA255 platform.

The results of the label benchmarks also show that the work put into enabling the ARM's virtually addressed cache has paid off. As Bootldr developer Jamey Hicks had predicted, there is approximately a twofold speed increase with the cache enabled. The cache behavior will be examined more closely in Section 6.2.3. It remains to be

Table 6.2: Operating system benchmarks in thousands of cycles.

Benchmark	**x86**	**ARM 4KB**	**ARM 1KB**
Null system call	1.420	13.946	14.006
Memory alloc/read/write	0.035	0.160	0.442
Ping pong message	33.378	618.343	641.757
File create, R/W, delete	1901.403	21001.248	50640.165
Spawn process	19247.662	83772.160	368180.836

seen what kind of improvements can be gained by enabling the write buffer.

6.2.2 Operating System Benchmarks

How fast an operating system can work is determined by the speed of its primitive operations. One important primitive for Asbestos is the context switching overhead. Microkernels rely on rapid context switching, so performing null system calls is the first benchmark. The second benchmark consists of reads and writes to memory from a user process. This test includes the allocation of memory: one page per 1024 read and writes when the page size is 4 KB. The third benchmark tests the overhead incurred by message passing, which is the core operation by which user processes in Asbestos communicate. Message passing involves remapping and sometimes copying virtual memory pages between user and kernel space. The ping pong test involves two processes synchronously replying to each other's messages. The fourth benchmark tests the performance of the Asbestos file system (ASFS). This test includes copious message passing, as well as label checks and ramdisk accesses. The last benchmark measures spawning a new process, an all-around test that should give a sense of the responsiveness of the system. The results are shown in Table 6.2.

ARM Asbestos performed an average null system call (two context switches plus trap identification code) in thirteen thousand CPU cycles. On the PXA255, this amounts to one system call (and return) every 30 μs. Notice that the system call overhead is not much different between ARM Asbestos with 4 KB and 1 KB pages, because page size is unrelated to context switching. The next benchmark, allocating and using memory, does vary with page size. The memory test with 1 KB pages is much slower, because 1KB ARM requires four times as many pages and performs four times as many system calls. Separate profiling data shows that half of the 4KB ARM test is spent allocating and deallocating the pages required for writing, so the observed 5/2 slowdown indeed matches expectations. Similar slowdowns can be seen for the other memory-intensive benchmarks: using the file system and process creation. It seems a little unreasonable that Asbestos should spend as much time allocating a new page as overwriting it. It appears that the large memory management overhead is not a feature of the ARM port, but rather a pervasive problem in Asbestos, and one that future work should address.

Asbestos relies heavily on its message passing machinery. Passing messages is the only way for user processes and system services to communicate in Asbestos. The numbers show that passing two messages (one round trip) takes around 618 thousand cycles, which includes everything from label operations to remapping data pages. This means the overhead for sending one message is about 309,000 cycles. Profiling data shows that in this test, passing a message includes one send and five receive system calls. Subtracting the context switch overhead for these operations leaves 219,000 cycles, or 500 μs of message passing overhead. This time encompasses the security checks that Asbestos performs. It seems acceptable for the port given that, both on ARM and x86, the messaging system calls comprise about 70% of this benchmark's runtime.

Finally there is the benchmark of spawning a simple process from the file system. This file system-intensive task took much longer on 1KB ARM than on 4KB ARM, mainly because the smaller pages required more messages between the file system and the invoking process. On ARM with 4 KB pages, spawning a simple "hello world!" program took 84 million cycles, or about 210 ms. Loading a program from the ramdisk is CPU- and memory-intensive, but crucial to the workability of ARM Asbestos as a mobile operating system. Typing `ls` at the shell essentially runs this benchmark, which therefore serves as a crude user evaluation. The ability to launch almost five new processes every second on the iPAQ h5555 implies that the shell feels responsive to users, a desirable feature for a mobile operating system.

Now I compare the x86 cycle counts to those of the ARM processor. Remember that absolute cycle counts are not important; the relative values for each benchmark are more informative. The results are varied. The ping pong test takes over 18 times more cycles on ARM, whereas the memory test runs in 4.5 times more cycles. In general, the context switch intensive benchmarks (system calls, ping pong) require more cycles than the other tests. This seems to be evidence that interrupts are inefficient either in the ARM port or on the ARM processor in general. Still, this observation doesn't tell us *why* ARM context switches are so inefficient relative to the x86. If the fault lies with the ARM port, one can expect a high number of instructions in system call intensive tests. If the fault lies with the processor, one should expect a normal instruction count with a high cycle count. Further performance data shows that a full system call and return takes about 300 instructions. This is not excessive, so the cause must be sought in the ARM hardware. Section 6.2.3 examines the operating efficiency of the PXA255 platform.

6.2.3 ARM Efficiency

The performance monitoring coprocessor on the PXA255 provided helpful measurements for the evaluation of the ARM Asbestos port, shown in Table 6.3. This data was taken with 4 KB pages, and it was virtually identical for 1 KB pages. The cache and TLB results match expectations. The TLB miss rates are small for all tests, and highest for the ping pong test. This is as expected, because Asbestos flushes all caches and TLBs on a context switch between user processes. The ping pong test constantly switches between two processes, and thus it is not surprising that TLB misses peak during this test. The instruction cache demonstrates the same behavior across context switches. When there is only one user process repeatedly executing the same words of user and kernel code (as in the system call test), cache efficiency is great. When more user processes are added (ASFS or a forked process), context switches ensue and a greater percentage of instruction accesses are cache misses.

Finally there is the data cache miss rate. Because the ARM Asbestos caches

Table 6.3: ARM efficiency for each OS benchmark

Benchmark	**Instr. per cycle**	**Icache miss rate**	**Dcache miss rate**	**ITLB miss rate**	**DTLB miss rate**
Null system call	0.0256	0.0010	0.5358	0.0000	0.0003
Ping pong message	0.0229	0.0767	0.6102	0.0045	0.0249
File create, R/W, delete	0.0345	0.0251	0.4129	0.0015	0.0080
Spawn process	0.0399	0.0258	0.3646	0.0016	0.0089

are configured as write-through caches, all memory writes are in fact cache misses. This results in a miss rate of around 50% for the system call benchmark. During each system call, the kernel writes a trap frame to memory and reads it from the same place not much later. Almost no work is done in user space, and therefore half of cache accesses are misses (the writes) and half are hits (the subsequent reads). The ping pong test is similar, except that the entire cache is invalidated after each environment switch. This leads to a higher cache read miss rate, because nearly all user-space operations related to constructing messages are cache misses. As a result, the total cache miss rate is also higher than 50%. During general Asbestos operation, the cache miss rate is not expected to be as high as 50%. It is generally assumed that the ratio of memory reads to memory writes during normal CPU operation is about two to one. If all reads hit in the cache, the resulting miss rate is 33%. Of course not all memory reads are cache hits, so we can expect the total miss rate to be a little bit higher than 33%. The more complex system benchmarks (file system access, spawning a process) support this, with a cache miss rate close to 40%. All in all, the sensible cache and TLB performance results provide a good sanity check for the ARM implementation.

The first measurement in Table 6.3 is the inverse of the ARM's executed cycles per instruction (CPI). The CPI of each benchmark was between 25 and 44. This result is worrisome. RISC processors such as the ARM are designed to achieve cycle-to-instruction rates of nearly 1, and sometimes even below 1. Why, then, are the benchmark CPIs so high? One explanation is the high rate of context switching.

```
     mov       r0, #(addr)       // R0 = address of allocated memory
     mov       r1, #42           // R1 = value to store in memory
     add       r2, r0, #409600   // R2 = memory address to stop at
1:                               // (code label)
     str       r1, [r0], #4      // Store R1 at R0's value, R0 += 4
     teq       r0, r2            // Test R0 == R2
     bne       1b                // If not equal, go back to label 1
```

Figure 6-1: Assembly code to test memory write performance

Table 6.4: Random-access memory benchmarks.

Benchmark	**Instr. per cycle**	**Dcache miss rate**
Writes, 4 byte increments	0.3269	0.9963
Reads, 4 byte increments	0.1319	0.2570
Reads, 8 byte increments	0.0703	0.5031

Software interrupts and IRQs take many processor cycles, and a microkernel like Asbestos can expect a lot of these operations. Still, the majority of operations are not context switches, so an average CPI of 30 is unreasonable.

My hypothesis is that the bottleneck in this iPAQ deployment of ARM Asbestos is the random-access memory. The memory test performance data is noticeably absent from Table 6.3, because it mixed memory reads, writes and system calls. In a revised benchmark, I attempt to separate memory reads and writes from all other operations, getting a definitive cycle count for each of them. Figure 6-1 shows the assembly code used to test memory writes. The memory read test is very similar. In both tests, the main loop consists of three instructions. One of these is a memory access, and the other two are simple register operations. It is assumed that these operations, a test and a branch, each take one processor cycle. Each loop ran 102,400 times, minimizing the effects of the setup instructions and any interrupting timer IRQs (one for each benchmark). While the benchmarks ran, the PXA performance coprocessor recorded data cache and CPI statistics. The results are shown in Table 6.4.

The results of these benchmarks are critical to explaining the CPI of all previous

Table 6.5: Possible caching scenarios and resulting CPI.

memory ops / instr.	**writes / memory ops**	**reads / memory ops**	**read miss rate**	**Dcache miss rate**	**CPI**
0.5	0.33	0.66	0.1	0.396	4.559
0.5	0.33	0.66	0.3	0.528	9.707
0.5	0.33	0.66	0.5	0.660	14.855
0.5	0.33	0.66	1.0	1.000	27.725

benchmarks. Nearly all of the write test's memory accesses were cache misses, as expected. The write CPI is about three, meaning that every three-instruction loop took nine cycles. Two of these cycles represent single-cycle register instructions. The remainder, *7 cycles*, represents the write to memory. For the first read test, only one in four reads caused a cache miss, which means one memory read occurred every four loops. The read CPI is about 7.5, meaning that every four loops (or twelve instructions) took 90 cycles. Eleven of these cycles were single-cycle instructions. The remainder, *79 cycles*, is the cost of a memory read. It is hard to explain why memory writes appear to happen faster than reads. Most likely the memory system performs some optimization that causes the CPU not to wait for writes to complete. In any case, 7 and 79 cycles are the cycle counts experienced by normal memory-accessing code. These results help to explain our high CPI results (25 to 44) from previous benchmarks!

Table 6.5 gives an idea of possible caching scenarios built on the memory benchmark data. I assume that two of every three memory accesses are reads, and that half of all instructions access memory. The projections also assume that all non-memory accesses take one cycle, but this is an underestimation. In reality the expected CPIs are increased by context switching and other hardware delays. Future work should examine exactly what other delays raise the observed ARM CPI to identify what other system bottlenecks exist. It would also be interesting to measure CPI on an ARM platform with a different (faster) memory interface.

The memory benchmark data in Table 6.4 shows one peculiarity which prompted a second memory read benchmark. The PXA255 data cache is supposed to have a cache line size of 32 bytes, or eight words. This suggests that for sequential data reading, one cache miss loads eight words into the cache and results in seven subsequent cache hits. The data shows that this was not the case! Instead, one cache miss resulted in three cache hits (a 25% miss rate). In the second memory test, the read address was incremented by eight bytes on every loop, so half the memory was skipped. The resulting data is consistent with the first read test, showing a doubled cache miss rate and nearly doubled CPI. It appears that the effective PXA255 data cache line size is not 32 bytes, but 16 bytes.

6.3 Cleanliness & Portability

A parallel result to the performance of ARM Asbestos is the positive change in the Asbestos code base. This first port has readied the operating system for future ports by separating code into architecture-specific sub folders for each implementation. The major differences between x86 and ARM Asbestos versions are in the kernel's memory management, trap handling, and hardware interfaces. These code files all reside in architecture-specific sub folders. Another contribution was the definition of a Hardware Abstraction Layer, a common interface for all future Asbestos ports. The MMU HAL had already proved to be extremely useful. I first defined the HAL for x86 and ARM with 1 KB pages. When I added 4 KB pages to the ARM port, all I had to do was implement the HAL macros. This was a fantastic time saver, tremendously cutting down implementation and debugging effort. Finally, the ARM port cleaned up quite a few miscellaneous x86-specific code dependencies. For example, it decoupled file system block size from page size, thus maintaining file system binary compatibility across ports.

Chapter 7

Conclusion

In this thesis, I have presented the design and implementation of ARM Asbestos, the first port of the Asbestos operating system to a processor other than the x86. I have split the kernel into ARM, x86 and shared components. In doing so, I have defined a HAL for Asbestos. Furthermore, I have ported several system services that make future Asbestos development viable on the ARM. The platform is now ready for Asbestos applications such as the OK Web Server. Finally, I demonstrated the quality and usability of the port with a performance comparison between Asbestos on the ARM and x86 processors.

While completely functional, the port of Asbestos to the ARM is not finished. The results chapter identified a number of areas that could use optimization. One way to achieve better performance for memory-intensive applications is to turn on the virtually addressed ARM write buffer. Another area of work is exploiting the features that newer ARM processors have to offer. ARM Asbestos was compiled for the version 4 ARM architecture. However, newer architectures offer more flexible memory management units and generally faster hardware.

The impact of the ARM port of Asbestos will only be as great as the applications it enables. The next step is to find a "killer application" for mobile devices that uniquely demonstrates the salient features of Asbestos: dynamic control of compartments, decentralized data declassification, OS-limited communication, and fine-grained information flow control. One possibility is a script-sharing application that accepts programs from anonymous users, but needs security guarantees about the data these scripts can access. Another idea is a calendar program that shares sensitive data about meeting times and project notes without worry of information leaks. Asbestos enables a new degree of information protection, an operating system feature that can be especially useful for servers and mobile applications.

Appendix A

Code Split

The x86 and ARM builds share all library and user application code. Below are tables of the kernel code they share, the kernel code that's separate, and the kernel code that is mostly shared (these files contain `#ifdef`s to control the code path).

Table A.1: Separate kernel code

File	**Description**
kclock	system clocks / performance unit
kconsole	serial port and screen output
kentry	kernel boot (assembly)
kinit	memory initialization
machine	CPU functions
memcpy	optimized memory copy (assembly)
memlayout.h	memory layout, MMU functions
memset	optimized memory set (assembly)
mmu.h	MMU layout
pfentry	page fault entry to user mode
picirq	interrupt controller
trap	interrupt and exception handling
trapentry	trapframe construction (assembly)
uentry	entry to user mode (assembly)

Table A.2: Mostly shared kernel code

File	**Description**
env	environment creation, context loading
syscall	system calls (separate register manipulation)
timer	timer calibration (separate delay loop, assembly)
pmap	page tracking module (could be made fully common)

Table A.3: Shared kernel code

File	**Description**
asbestos	Asbestos labels, vnodes, etc
asbestos_request	Asbestos messages
bcache_dev	File system block cache
control_dev	Flow control
dbug_dev	Debug functionality
fork_dev	Copy memory on fork
idedisk	IDE disk driver (not used on ARM)
init	Start all services
labels	Asbestos label functions
lcache	Asbestos label cache
micro	Microbenchmarks
monitor	Kernel debugger/monitor
net	Driver-calling network code
null_dev	Null device
printf	Basic print functions
proc_dev	Process labels + messaging
profile	Profiling module
ramdisk	Ramdisk driver
sched	Process scheduler
select_dev	Select device
slab	Memory slab tracker / allocator
stabs	Debugging symbols

Appendix B

ARM Asbestos HOWTO

B.1 Running Asbestos on the Skyeye Simulator

1. Download and install the latest version of the Skyeye simulator. At the time of writing, this is version 1.2 RC7_3. Earlier versions do not support the PXA processor or GDB remote debugging.
   Information: `http://www.skyeye.org/index.shtml`
   Download here: `http://gro.clinux.org/frs/?group_id=327`

2. Create a Skyeye configuration file, named skyeye.conf. I'm using the XScale PXA25x hardware model, because I have a PXA255 as hardware. You must specify processor, architecture, and memory banks:
   For the PXA255:

   ```
   cpu: pxa25x
   mach: pxa_lubbock
   mem_bank: map=M, type=RW, addr=0xF0000000, size=0x03FFFFFF
   mem_bank: map=I, type=RW, addr=0x40000000, size=0x3BFFFFFF
   ```

   For the SA1110:

   ```
   cpu: sa1110
   mach: sa1100
   ```

```
mem_bank: map=M, type=RW, addr=0xF0000000, size=0x03FFFFFF
mem_bank: map=I, type=RW, addr=0x80000000, size=0x3BFFFFFF
```

The simulator loads the kernel at 0xF0008000 (image link address), so you must specify the M memory bank there. The hardware memory banks for these machines are not at 0xF0000000! I worked around this in arm/proc.c, defining a memory map that includes this address as the DRAM0 bank. The I memory bank is the hardware-specific memory-mapped IO bank.

The skyeye.conf file can be used to specify other simulator specifics as well: network interface, serial port redirection, logging, and LCD on/off. See `www.skyeye.org` or do a web search for "skyeye.conf" for more details.

3. Compile Asbestos for ARM:
   `make ARCH=arm`
   For now, compile with the RAMDISK (until flash memory driver works). This can be done with a file conf/local.mk with contents RAMDISK=1.

4. Run Skyeye
   STANDALONE mode:
   ```
     skyeye -e obj/kern/kernel
   ```
   GDB mode (2 windows):
   ```
     1. skyeye -d -e obj/kern/kernel
     2. arm-elf-gdb obj/kern/kernel
          (gdb) target remote localhost:12345
   ```
   If you compiled in STABS information (-gstabs flag, but bigger kernel image), GDB should recognize it and let you step over functions, etc.

B.2 Running Asbestos on the iPAQ

1. Read Jamey Hicks' Linux iPAQ HOWTO. Carefully follow the steps up to Chapter 5: Bootstrapping Familiar GNU/Linux. The steps for loading/booting ARM will be similar to Chapter 5, but the details vary as follows.

2. Set up partitions in the bootloader:

   ```
   boot> partition reset
   boot> partition delete root
   boot> partition define params 0x40000 0x40000 0x0
   boot> partition define kernel 0x80000 0x1f800000 0x0
   boot> partition save
   ```

   This should save your partitions across resets. Partitions are lost on hard resets, and you will have to redo this step.

   Note: if you use flash-based ASFS later, you'll have to define a partition for it.

3. Load the Asbestos kernel image:

   ```
     boot> load kernel
   ```

   Start a ymodem transfer of zImage.bin. In minicom, this works as follows. Press Ctrl-A and S. A small popup window will appear, allowing you to choose the transfer protocol. Using the arrow down key, move down to the "ymodem" entry and press Enter. Select your file using space bar to navigate directories and enter to pick the file. Pick zImage.bin. Depending on kernel size this will take a while. It's VERY slow. 115200 bps, or 14.4 KB/s. Use the simulator when you can.

4. Boot!

   ```
     boot> boot flash
   ```

   Voila.

B.3 Hardware Notes and Resources

The iPAQ is a tricky beast. Some pointers:

- You can do a soft reset of the iPAQ by using the stylus to press the recessed reset button on the bottom.

- You can do a hard Bootldr reset by pressing and holding the directional pad and pressing the reset button. This resets all partitions.
- The h5555 will not charge when running as charging is software-controlled. The Compaq bootloader will charge the iPAQ at the `boot>` prompt, but ONLY just after a reset (it appears). Make sure to reset immediately after you run your kernel, or you will find yourself with a low battery (see next).
- When the battery gets very low, the iPAQ can start buzzing unstoppably (it gets stuck in the bootup process). In this case, take out the battery and let the iPAQ sit for about 2 to 3 hours. YES, it takes a long time. Then try putting the battery back in. If it still buzzes, wait longer. When it no longer buzzes, do a HARD reset (press + hold directional pad, press reset button), and then immediately put it in the cradle to charge.

`http;//www.handhelds.org`
This website is an excellent resource for drivers for iPAQs and other kinds of mobile devices. The site has a great deal of information about hand held hardware (especially iPAQs). The site is somewhat disorganized, but the Wiki and sources have most of the information you need.
Hardware info: `/moin/moin.cgi/IpaqHardwareCompatibility`
Supported handhelds: `/moin/moin.cgi/SupportedHandheldSummary`
Sources: `/sources.html`

Bibliography

[1] 6.828: Operating Systems Engineering. `http://www.pdos.lcs.mit.edu/6.828/2005/`, December 2005.

[2] Advanced RISC Machines Ltd. Accessing unaligned data from c. `http://www.arm.com/support/faqdev/1469.html`, April 2006.

[3] Advanced RISC Machines Ltd. The arm instruction set architecture. `http://www.arm.com/products/CPUs/architecture.html`, April 2006.

[4] Todd M. Austin, Eric Larson, and Dan Ernst. Simplescalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2):59–67, 2002.

[5] Laszlo Bacsi, Michel Stempin, and Jamey Hicks. The Linux iPAQ HOWTO. `http://math.bme.hu/~lackac/ipaq/linux-ipaq/Linux-iPAQ-HOWTO-1.1.html`, August 2002.

[6] Fabrice Bellard. QEMU. `http://fabrice.bellard.free.fr/qemu/`, 2005.

[7] John Biggs. iPAQ on Linux. *j-LINUX-J*, 92, December 2001. Web only.

[8] Bochs: The Open Source IA-32 Emulation Project. `http://bochs.sourceforge.net/`, March 2002.

[9] Computer Emergency Readiness Team. Microsoft Internet Information Server (IIS) 4.0, 5.0, and 5.1 buffer overflow in chunked encoding transfer mechanism for ASP. `http://www.kb.cert.org/vuls/id/669779`, 2002.

[10] Computer Emergency Readiness Team. Microsoft Internet Information Server (IIS) vulnerable to buffer overflow via inaccurate checking of delimiters in HTTP header fields. `http://www.kb.cert.org/vuls/id/454091`, 2002.

[11] Computer Emergency Readiness Team. Apache HTTP Server contains a buffer overflow in the mod_proxy module. `http://www.kb.cert.org/vuls/id/541310`, 2003.

[12] Computer Emergency Readiness Team. Microsoft SQL Server vulnerable to buffer overflow. `http://www.kb.cert.org/vuls/id/584868`, 2003.

[13] Computer Emergency Readiness Team. Sun Java Runtime Environment allows untrusted applets to access information within trusted applets. `http://www.kb.cert.org/vuls/id/393292`, 2003.

[14] Computer Emergency Readiness Team. Apache vulnerable to buffer overflow when expanding environment variables. `http://www.kb.cert.org/vuls/id/481998`, 2004.

[15] Computer Emergency Readiness Team. MIT Kerberos krb524d insecurely deallocates memory (double-free). `http://www.kb.cert.org/vuls/id/340792`, 2004.

[16] Josh Daymont. Windows CE Hardening. `http://www.blackhat.com/presentations/bh-usa-03/bh-us-03-daymont.pdf`, 2003. Black Hat 2003.

[17] Adam Dunkels. lwIP - A Lightweight TCP/IP stack. `http://savannah.nongnu.org/projects/lwip/`, 2002.

[18] eCos. `http://ecos.sourceware.org/`, 2000.

[19] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the asbestos operating system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP'03)*, pages 17–30, Brighton, UK, October 2005. ACM.

[20] Sony Ericsson. Evolution towards converged services and networks. `http://www.ericsson.com/products/white_papers_pdf/convergence_b.pdf`, 2005.

[21] Martin Guy. Deciding new arm EABI port name. `http://lists.debian.org/debian-arm/2006/04/msg00003.html`, March 2006.

[22] Handhelds.org. `http://www.handhelds.org/`, 2000.

[23] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of μ-Kernel-based systems. In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP-97)*, volume 31,5 of *Operating Systems Review*, pages 66–77, New York, October 1997. ACM Press.

[24] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Modular system programming in MINIX 3. *j-LOGIN*, 31(2):19–28, April 2006.

[25] Jamey Hicks. How to Run Linux on iPAQ Handhelds. `http://www.handhelds.org/handhelds-faq/`, 2002.

[26] Shuo Kang, Huayong Wang, Yu Chen, Xiaoge Wang, and Yiqi Dai. Skyeye: An instruction simulator with energy awareness. In Zhaohui Wu, Chun Chen, Minyi Guo, and Jiajun Bu, editors, *ICESS*, volume 3605 of *Lecture Notes in Computer Science*, pages 456–461. Springer, 2004.

[27] Russell King. ARM Linux Mailing Lists FAQ. `http://www.arm.linux.org.uk/mailinglists/faq.php`, January 2004.

[28] Maxwell Krohn, Petros Efstathopoulos, Cliff Frey, Frans Kaashoek, Eddie Kohler, David Mazières, Robert Morris, Michelle Osborne, Steve VanDeBogart, and David Ziegler. Make least privilege a right (not a privilege). In *Proceedings of HotOS X: The 10th Workshop on Hot Topics in Operating Systems*. USENIX, June 2005.

[29] Maxwell N. Krohn. Building secure high-performance web services with OKWS. In *USENIX Annual Technical Conference, General Track*, pages 185–198. USENIX, 2004.

[30] LinuxDevices.com. `http://www.linuxdevices.com/`, 1999.

[31] MontaVista Software. `http://www.mvista.com/products/`, 1999.

[32] Trevor Mudge, Todd Austin, and Dirk Grunwald. The SimpleScalar-Arm Power Modeling Project. `http://www.eecs.umich.edu/~panalyzer/`, 2002.

[33] NetBSD. `http://www.netbsd.org/`, 1993.

[34] News.com. Nokia: 2 billion cell phone users by 2006. `http://news.com.com/Nokia+2+billion+cell+phone+users+by+2006/2100-1039_3-5485543.html`, 2004.

[35] Red Herring. ARM Unveils Mobile Processor. `http://www.redherring.com/Article.aspx?a=13859`, October 2005.

[36] J. Saltzer and M. Schroeder. The protection of information in computer systems. *Proceeding of the IEEE*, 63(9):1278–1308, September 1975.

[37] Slate. The Perfect Worm. `http://www.slate.com/id/2115118/`, 2005.

[38] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating systems: design and implementation*. Pearson Prentice Hall, third edition, 2006.