The Design and Implementation of the L4 Microkernel on the StrongARM SA-1100

Supervisor: A/Prof. Gernot Heiser

Assessor: Dr. Jerry Vochteloo

A Thesis submitted to the School of Computer Science and Engineering of The University of New South Wales in partial fulfillment of the requirements for the degree of Bachelor of Engineering (Computer Engineering)

Adam Wiggins (2193695)

November 1999

Abstract

The ARM architecture has established itself as one of the leading microprocessors in today's embedded and portable marked. Its simple design founded in RISC principles yet balanced by CISC influence leads to low cost/power cores which offer high performance. In particular the StrongARM SA-1100 has proven that an ARM processor can deliver enough processing power within the constraints of low power, low cost and small size, to handle such demanding tasks as voice recognition in a handheld device.

Recent trends have seen a push towards the need for secure and reliable operating systems in embedded and portable devices for such application areas as Internet appliances and wireless networking for handhelds. This thesis examines the challenges facing the design and implementation of a general purpose microkernel to the StrongARM SA-1100 and evaluates the potential performance of such a system. It is hoped that the work covered will benefit future development of general purpose operating systems on ARM processors as well as provide a framework for microkernel development in general.

Acknowledgements

This work would not have possible without the support and contributions of a countless number of individuals that it has been my pleasure to have met and known during my years at the University of New South Wales. A list (by no means complete) of some of the names that come to mind, is as follows:

I'd firstly like to thank my supervisor Gernot Heiser who's support and faith in me have made this work possible. His guidance and input have been invaluable. I'd also like to thank my DiSy group colleagues, both past and present for the many drinks and discussions we've shared.

I'd also like to thank the Operating Systems Group at the University of Dresden, Germany. In particular Hermann Härtig for his invitation to visit Dresden. I'd like to thank Daniel Potts my travel partner for putting up with me and Sebastian Schönberg for organising our accommodation. I'd especially like to thank both Michael Hohmuth and Jean Wolter for their hospitality and friendship during my visits to Germany.

I'd like to thank the System Architecture Group at the University of Karlsruhe, Germany. In particular Jochen Liedtke for providing me with the opportunity to visit Germany for a second time to work with his group. Also I'd like to express my gratitude to Christian Ceelen and Andrea Engelhart for getting me settled in Karlsruhe and making my stay a pleasant one.

Special thanks also goes to Uwe Dannowski and Espen Skoglund for their friendship and input into the development of the kernel.

I'd also like to thank my friends for always providing an escape from work and the troubles of life, as well as for looking out for me when they felt like it :) They know who they are.

Finally I'd like to thank my family, in particular my parents, Penny and John. Their love, support and guidance has seen me through ups and downs of life.

Contents

Chapter 1

Introduction

1.1 Background and Motivation

This thesis topic stemmed out of the PLEB Project [Wig98b], a project initiated to develop a palmtop computer capable of running the Linux Kernel. Since its instantiation, the goal of the PLEB project has matured to that of developing a flexible system for use in high performance portable and embedded applications. This goal amounts to developing both hardware, and operating system level software suitable for use in embedded and portable applications were a high level of processing power and system performance is required.

Some of the anticipated target areas of the PLEB project include, but are not limited to, the following: Wearable computing, robotics and hard-real-time systems such as scientific measurement and instrumentation. Since PLEB is primarily a research project being undertaken at the University of New South Wales, a side target application area is use in teaching and research.

While Linux provides a mature kernel with a extensive UNIX environment, it has many limitations, some general to its design, others specific to the StrongARM port. In particular while Linux finds much usage in workstation and server environments, it is only in the first stages of exploring its usability in embedded and portable environments. Due to its source code size and structure the Linux kernel does not present an ideal platform for the kinds major internal reorganisation required in research, for example to replace the virtual memory management system. Another notable problem with Linux is its poor address space context switch performance on the StrongARM. While the Linux kernel's limitations may be addressed in the not too distant future, in keeping with building a flexible system it was decided that an alternative to Linux should be examined and developed.

Following the modular design philosophy of the project it was decided that a microkernel [LES+97, BH70] would be the most suitable operating system base, allowing systems from hard-real-time executives to general purpose time sharing systems to be implemented on top of it, with the option for concurrently running other operating system personalities [HHW98]. The biggest concern in using a microkernel base was the poor performance displayed by first generation microkernels [CB93, Lie96d]. The L4 microkernel

[Lie95b, Lie96a, AH98] presented itself as an ideal choice not only due to its high performance implementations, but since the research group behind the PLEB project have a solid experience base with L4. This experience is due to the usage of L4/MIPS [EHL97] in teaching and research [HEV+98] at the university.

As a result, a proposal [Wig98a] was submitted to implement L4 on top of the ARM architecture (L4/ARM), known as Gauntlet.

1.2 Thesis Structure

The structure of this thesis basically covers two main aspects. The coarse (or high level) kernel design, and the kernel implementation (with relevant micro-optimisations). The progression of the thesis chapters is outlined as follows:

Introduction: This chapter, presents the motivation and the goals behind the thesis.

Background: Presents the background material of the thesis (the ARM architecture and StrongARM SA-1100 CPU).

Design: Explores the challenges faced and outlines the coarse high level design of L4 on ARM and the SA-1100.

Fast Address Space Context Switches: Details why address space context switches on the SA-1100 are expensive and outlines a management technique which reduces these costs.

Implementation and Evaluation: Examines the development of a base implementation of L4 on StrongARM SA-1100, as well as any optimisations. It also analyses the potential performance of this implementation.

Future work and Conclusions: The final chapter elaborates on what work this thesis paves the way for. It then finishes up with a discussion of what conclusions about the design can be drawn from presented implementation.

1.3 Thesis Goals

The basic goal of this thesis is to present and evaluate a complete design of L4 on the SA-1100, as well as provide a base framework from which a complete and optimised implementation can be derived.

It is also hoped that this thesis together with the implementation can provide a reference for ports of L4 to other processors and architectures as well as the development of L4's API and ports of other operating systems to the ARM and SA-1100.

1.3.1 Gauntlet's Goals

The goal of the implementation, known as Gauntlet, is to provide a flexible foundation for operating systems research and development. It is hoped that Gauntlet will in the near future exhibit the following characteristics:

- Stability
- Complete, well defined API
- High performance
- Hard-real-time support
- Power management support

Once Gauntlet displays these properties the performance of the implementation (and hence the presented design) can be throughly evaluated as well as further ARM and possibly other architectural ports explored.

Chapter 2

Background

2.1 The ARM Architecture

The ARM architecture [Jag95, Fur96] is one of the most commonly used architecture in embedded systems in the world. Since the first code ran in silicon in 1985 the ARM's unique design, fundamentally RISC while drawing on CISC strengths, has found its niche in the embedded market due to its simple design which leads to low cost/power implementations. For a full history of the ARM see [Fur96].

The basis of this chapter is to introduce the ARM from the operating system point of view; the programming model and the user view of the architecture are not covered in detail. For a full description of the programming model and instruction set see [Jag95]. The rest of this chapter refers to version 4 of the architecture unless otherwise stated.

A quick summary of the ARM architecture is:

Overview

- 32-Bit load/store RISC architecture
- High performance at low cost/power
- 15 visible general purpose registers and *program counter* (PC)
- Privileged mode register banking

Good Code Density

- Conditional execution of all instructions
- Parallel shift & ALU operations
- Multiple register load/store instructions
- 'Thumb' 16 Bit instruction sub-set (not supported by StrongARM)

Virtual Memory Support

- Hardware loaded page table
- Domains, a variation on *address space identifies* (ASIDs) [Hen84]

2.1.1 Privileged Modes and Register Banking

The ARM provides 7 operating modes. Of these one mode is dedicated to the user and the remaining 6 are privileged modes of operation for the support of operating systems. Modes can be switched under software control but only from a privileged mode. Switching from the user mode to a privileged mode of operation can only occur in a controlled manner via exceptions either under the control of the user (ie a SWI, Software Interrupt) or via memory aborts, interrupts, etc.

One of the main features of the privileged modes is that of register banking. The ARM has thirty-seven, visible registers:

- 30 general-purpose registers
- 6 status registers
- a program counter

The registers are arranged in a partially overlapping banks with a different register bank for each processor mode. All privileged modes of operation entered via exceptions (5 of the 6 modes) bank (that is have their own dedicated register for) the *stack pointer* (SP) and *link register* (LR) as well as a *saved program status register* (SPSR), the remaining privileged mode has the same view of the register set as the user mode and can only be entered via software control from another privileged mode. This design is to facilitate fast exception handling via hardware switching of the stack and support of exceptions (which trap into a different privileged mode) within exceptions without saving the LR and SPSR. The FIQ mode (Fast Interrupt) banks an extra 5 registers to further reduce FIQ handling costs.

2.1.2 Exception Handling

Exceptions are generated by internal and external sources and result in a change of the flow of control of the processor to handle events, e.g., externally generated interrupts or the execution of an undefined instruction. During an exception, the processor state must be saved and then restored after the exception is handled. ARM supports some hardware saving of state via the register banking method described above.

On the ARM when an exception is raised the PC is forced to a fixed address vector in an array starting at address 0x0. Each entry is required to contain a branch to the relevant exception handler routine. ARM supports the following exceptions.

Reset: The *reset* exception is raised via a CPU reset condition, this can be signaled from a number of sources, from initial power up to wake up from sleep the mode (see 2.2.3) of operation on the SA-1100.

Undefined Instruction: An *undefined instruction* exception is generated whenever a program tries to execute an unimplemented instruction or when no coprocessor responds to a coprocessor request. It can be used to emulate instructions and/or coprocessors.

Software Interrupt (SWI): The *SWI* exception is the standard method for invoking operating system service requests from the kernel. This exception is generated by the SWI instruction.

Instruction Fetch Memory Abort (Prefetch Abort): When a memory abort is generated during a instruction fetch a bit is flagged. If the instruction reaches the execute phase, a *prefetch abort* exception is raised to signal the memory fault.

Data Access Memory Abort (Data Abort): When a memory abort is generated during a data access a *data abort* exception is raised immediately.

Interrupt Request (IRQ): The *IRQ* exception is raised when the IRQ signal of the CPU core is asserted, it is intended for use with the majority of interrupt signals. IRQs can be masked by setting the I-bit of the CPSR (Current Program Status Register), an obviously privileged operation.

Fast Interrupt Request (FIQ): The *FIQ* exception is raised when the FIQ signal of the CPU core is asserted, it is intended for use by a single or small number of interrupts that require fast handling to implement things like software DMA. FIQs can be masked by setting the F-bit of the CPSR, again this is a privileged operation.

Once an exception is raised the I-bit of the CPSR is set to mask IRQs. If the FIQ exception is raised the F-bit of the CPSR is set as well to mask all interrupt sources.

2.1.3 Memory Management Unit

The ARM architecture defines virtual memory support via a hardware loaded *pagetable* with the option for *translation look-aside buffers* (TLBs). An implementation may provide either a unified TLB or split *instruction TLB* (ITLB) and *data TLB* (DTLB). The features of the ARM memory management architecture can be summarised as follows:

- Hardware loaded TLB(s)
- Support for mixed page sizes (1MB, 64KB, 4KB as well as 1KB on newer processors)
- Coarse grained tagging via ARM's domains
- Support for TLB & cache management

2.1.4 Page Table Format

The ARM TLB(s) are hardware-loaded from a two-level page table. The first level table is referred to as the *page directory* (PD) and the second level tables as *leaf page tables* (LPTs).

The PD is an array of 4096 *section entries* (SEs). Each section entry can be set to one of four types as follows:

Fault: The entry is invalid and will lead to a *translation fault* on access which will raise a prefetch/data abort.

Page table: The entry contains a pointer to a LPT, as well as a domain identifier (see 2.1.5) which tags the whole LPT.

Section: The entry maps a 1MB (aligned) region of virtual address space, as well as a domain identifier and permission bits which apply to the whole 1MB region.

Reserved: The entry is reserved for future expansion and is actually used in newer processors [ARM99] to support 1KB page sizes.

A LPT is an array of 256 *page table entries* (PTEs). Each PTE can be set to one of four types as follows:

Fault: The entry is invalid and will lead to a *translation fault* on access which will raise a prefetch/data abort.

Large page: The entry maps a 64KB (aligned) page. The entry contains four sets of permission bits, each set defining accessibility of one of the four 16KB sub-pages.

Small page: The entry maps a 4KB (aligned) page. The entry contains four sets of permission bits, each set defining accessibility of one of the four 1KB sub-pages.

Reserved: This entry type is reserved for future expansion and must not be used.

Is should be clear from the above that the ARM supports three page sizes (newer chips with the additional 1KB size support four) from 4KB to 1MB. To optimise TLB coverage larger page sizes should be used wherever possible.

2.1.5 Domains and Protection Bits

Protection Bits

The ARM page table protection bits, along with the *System* and *ROM* protection bits of the *system coprocessor control register* (CP15), allow specification of one of the following access rights:

- No Access,
- Kernel R/O, User No Access,
- Kernel R/O, User R/O,
- Kernel R/W, User No Access,
- Kernel R/W, User R/O,
- Kernel R/W, User R/W.

An memory access which violates the permissions of the accessed page/section results in a *permission fault*, raising a prefetch/data abort exception.

Domains

ARM Domains generalise the access permissions bits found in many TLBs. The ARM architecture 16 domains; each domain forms a collection of section and page mappings defined in a page table (or TLBs), tagged by the domain identifier. A coprocessor register, the *domain access control register* (DACR), defines the accessibility of all domains, by specify each domain's state as one of the following:

No Access: Any access to sections or pages belonging to this domain will generate a *domain fault* raising a prefetch/data abort exception.

Client: Accessibility is determined by the setting of the standard permission bits of the section/page entry (with sub-page granularity).

Master: Access is permitted independent of the setting of the standard permissions bits of the section/page entry.

Reserved: This state is reserved for future expansion and must not be used.

2.2 The StrongARM SA-1100 CPU

The StrongARM SA-1100 [SAT98] is a high speed, low power implementation of the ARM architecture. Based around the SA-1 processor core, teamed with peripheral controllers (Memory controller, serial, I/O, etc) and integrated into a single package, its design is targeted to portable and embedded systems.

The SA-1 core was jointly developed by ARM and Digital Semiconductor (now owned by Intel). At the time it was released the SA-1 core was the first ARM processor to implement a modified Harvard architecture [Hen84], that is split caches and TLBs for instruction and data streams.

The SA-1100 offers high performance at low power, delivering 200MHz plus at under 500mW power consumption. Intel's plans for the second generation StrongARMs is to increase that speed to 600MHz at the same level of power consumption.

For the purposes of this thesis some of the features of the SA-1100 will be examined in detail in the following sections.

2.2.1 Cache Architecture

The SA-1100 as mentioned above incorporates a split cache architecture to achieve its high clock speed. The main instruction and data caches are complimented by a standard *write buffer* as well as a data prefetching *read buffer* and a *mini-data* cache.

All caches are fully virtual (that is virtually indexed and tagged).

Instruction Cache

The Instruction cache is made up of 512 lines of 32 bytes (8 words) each to make 16KB in total organised in a 32-way set associative arrangement.

A cache control operation allows *invalidation* of the complete instruction cache.

Data Caches

The data caches are made of a *main-data cache* as well as a *mini-data cache* with the following properties:

Main-data cache: The main-data cached is made up of 256 lines of 32 bytes each to make 8KB in total organised in a 32-way set associative arrangement.

Mini-data cache: The mini-data cache is made up of 16 lines of 32 bytes each to make 512B in total organised in a 2-way set associative arrangement. The purpose of the mini-data cache is to be allocated for large data structures which would otherwise unnecessarily flood the main-data cache.

The *cachable* and *bufferable* PTE/SE attributes define in which (if any), of the two caches a data line may reside with the hardware guaranteeing that no line can reside in both caches at the same time. Stores do not allocate in either cache and both implement a write-back policy.

Cache control operations allow *invalidation* of the whole cache or a single line, as well as a *clean* operation to force write-back of a single line.

Write/Read Buffers

To further enhance the efficiency of the processor a standard write buffer as well as a data pre-fetch buffer are implemented, with the following properties:

Write Buffer: The write buffer can hold up to eight blocks of data, each 1 to 16 bytes in size (up to half a data line). The allocation of stores to the write buffer is controlled by the *cachable* and *bufferable* PTE attributes corresponding to the data.

A *drain* operation is supported to empty the whole write buffer.

Read Buffer: A read buffer allows preallocation of read-only data into one of four 32 byte buffers without stalling the processor. Load operations which hit the read buffer bypass the data caches allowing critical data to be locked down.

Control operations allow fine grained software control of the contents of the read buffer.

Translation Look-Aside Buffers

The SA-1100 provides split TLBs for both instruction and data streams. Each TLB contains 32 entries and is fully associative. The SA-1100 TLBs are untagged like the caches, requiring coherency to be ensured by software whenever the page table is modified or is switched (via an address space context switch).

Control operations allow *invalidation* of the whole ITLB and/or DTLB as well as of a single DTLB entry.

2.2.2 ARM Architectural Enhancements

The SA-1100 added some additional features to the core to enhance the performance of operating systems running on-top of it. These enhancements are now incorporated in newer ARM processors, in particular the ARM920 [ARM99] and ARM720 [ARM98] cores.

Exception Vector Relocation

The exception vector relocation simply allows the operating system to relocate the exception vector from address 0x0 to 0xFFFF0000, facilitating kernels residing in the top of the address space without the cost of long jumps (direct branches have a restricted range)

The PID Register

The *process identifier* (PID) register facilitates relocating the first 32MB of virtual addresses prior to translation into a physical address. If an address less then 32MB is issued that address is bitwise OR-ed with bits 30-25 of the PID register. This effectively re-maps a small 32MB address space into any aligned 32MB region of the lower 2GB of the virtual address space (a total of 64 'slots').

As any addresses outside the lowest 32MB are unaffected by the PID register its use introduces a form of aliasing in the address space. If $0 \leq x < 2^{25}$ then addresses x and $PID|x$ refer to the same memory location (with the same access permissions).

2.2.3 I/O Support

This section examines some of the SA-1100 I/O support relevant to operating systems.

Interrupt Controller

The SA-1100 interrupt controller is a memory mapped controller which gives the software full control over the interrupt sources. Each interrupt source may be individually masked as well as be mapped to either the IRQ or FIQ signal.

System Timers

The SA-1100 provides two system timers with memory mapped interfaces:

Real Time Clock (RTC): A RTC with 1s granularity is provided and maintained even during sleep mode. Since 1s granularity is too coarse to be useful as an operating system timer it is left to the user space and is of no further interest.

Operating System Timer: A operating system timer with four match registers (each with a unique interrupt signal) is supplied with a granularity of the the main oscillator clock (3.6864MHz) from which a 1ms period can be calculated.

Low Power Operation

The SA-1100, being targeted at portable systems, has a wide range of support for low power operation and stand-by. Two low power modes of operation are supplied:

Idle Mode: The Idle mode is entered by a series of privileged operations to disable the main clock and wait on an interrupt. This mode allows low power while retaining fast wake-up response time.

Sleep Mode: The sleep mode of operation is entered by a memory mapped control register. This mode puts the full system into its lowest power stand by mode and then shuts down all operation of the core (besides the updates of the RTC) and offers the lowest power consumption at the cost of a much slower restart time. Sleep mode is exited via a software reset and hence requires support from the firmware (since resets disable the MMU).

The SA-1100 also provides selectable clock speeds to facilitate lower power consumption. This is controlled by memory mapped I/O register.

2.3 The L4 Microkernel

A microkernel can be viewed as a kernel (software running in the privileged mode of operation of the processor) which implements a minimal set of abstractions on which any complete from of operating systems services can be build upon.

L4 is a second generation microkernel developed by Jochen Liedtke. It is build on the principle that only those operations that require the processors privileged mode of operation should be in the kernel, the rest should run in user space.

Implementations currently exist for ix86 [Lie96a], MIPS R4600[1] [EHL97] and Alpha [Sch96], all of which are workstation/server orientated processors.

2.3.1 L4's Basic Abstractions

This is a quick introduction to the L4 model. For full details see [AH98].

Address Spaces: L4 provides the facility to define address spaces recursively from a identically virtual to physical mapped base address space.

Threads: L4 provides an execution abstraction via light-weight threads, where each thread is associated to a task and executes in that task's address space.

Inter Process Communication (IPC): IPC is the basic communications and synchronisation abstraction. L4's IPC performance is the critical factor of is speed and flexibility.

[1] The R4600 series is classed as an 'embedded' processor. However its size and power consumption limit it to a particular subset of console based embedded applications.

2.4 Currently Available Systems

While a large number of kernels and operating system environments are available for the ARM including some StrongARM specific ports, the majority of them are proprietary and closed systems where source code is only available with licensing fees. Linux represents one of the few free kernels available for ARM. Linux is also one of the few general purpose operating system kernels on the ARM as most available ARM kernels focus on running a few embedded applications and/or providing real time support.

A detailed list of available systems can be found on ARM's web site at `http://www.arm.com/DevSupp/OS/`

Chapter 3

Design

A number of challenges face the design of L4 explored in this thesis, most unique to the ARM and/or SA-1100, as well others more general to the targeted application areas.

The general philosophy of L4 can be regarded as minimalist. In minimising the size and complexity of the microkernel, its interference with user programs is minimised [Lie96b, Lie95b]. In light of this, the overall constraint on the design and implementation should be to minimise all data structures and code in terms of their size and complexity.

3.1 Exception Handling

L4 makes use of IPC to communicate all events to user space handlers. The majority of exceptions raised in the user space (as well as some in the kernel) result in this form of an IPC operation to communicate the exception as follows:

- Interrupts are signaled to the interrupt handler via an *interrupt* IPC,
- Page faults are signaled to the thread's registered page fault handler via a *pager* IPC,
- Finally, general user exceptions are signaled to the thread's registered exception handler via an *exception* IPC[1]

The remaining exceptions are used to handle events like timeslice expirations or to implement the fast address space switching mechanisms described in Chapter 4.

3.1.1 General Constraint

Before any specifics regarding exception handling can be examined a general constraint of L4 exception handlers on the ARM must be highlighted: that is *"All ARM exceptions potentially lead to context switches in L4"*.

This constraint can be illustrated as follows:

[1] This is not the case for L4 implementations in general, however it is the case for Gauntlet.

Syscalls: *ipc()* as well as *thread_switch()* syscalls lead to a thread context switch.

Prefetch/Data Aborts: Both potentially result in a either a pager or exception IPC.

Undefined Instruction: Results in an exception IPC.

Interrupts: Interrupts (IRQs and FIQs) fall into two categories: kernel interrupts for system timers resulting in a thread switch; user interrupts which result in interrupt IPCs.

It can be seen that all exceptions (except reset, which is irrelevant in this context) can result in an IPC or other event which leads to a context switch. What this means is that the interrupted processor state (regardless of whether in the a kernel or user mode) must be stacked on entry to an exception handler. This constraint over-shadows the overall exception handler design.

3.1.2 Register Banking

The main challenge the ARM architecture presents is that of how to deal with the register banking across different processor modes. The register banking as mentioned in Section 2.1.1 leads to a kernel stack per exception mode. The reason that this presents a problem is that L4 is based around a kernel stack per thread [Lie93], not per exception mode/type and, the privileged mode banked registers cannot be accessed from other modes, requiring explicit switching between modes which represents pure overhead. The overhead results from having to move banked registers to non-banked registers or by temporarily storing them in well know locations to be reloaded and stacked later. This would add an unacceptable amount of overhead to the critical IPC path.

Two general solutions to the problem the register banking presents, to a kernel stack per thread design, are examined in turn below. Both methods basically ignore the banking of different ARM privileged modes.

Exception Handling with a Single Mode

One simple way of avoiding the problem is to use a single privileged mode of the ARM to handle all exceptions. While this may at first appear not to solve the overhead problems, by using some unconventional tricks the time critical exceptions can be kept fast. This is achieved by using the abort mode to handle all exceptions in the following way:

Syscalls: Syscalls can be generated by attempting to execute from kernel reserved memory. The attempted address fetch can then be used as a syscall number which is loaded by the hardware into the LR (PC of aborted instruction)

Prefetch/Data Aborts: These exceptions trap into the abort mode already.

Undefined Instructions: The SWI and undefined instruction exceptions result in exception IPCs to be handled by user space (to emulate the instruction or otherwise). This allows binary compatibility with other operating systems by emulation of the SWI syscall trap.

While user exception IPC generation is less time critical, the handler would have to store the LR and SPSR to a well know memory location, switch to the abort ARM mode then reload them into a scratch register (the abort LR) and stack them one at a time. This procedure requires a pure overhead of at least 2 stores, 2 loads and a mode switch, requiring 9 cycles or more in instruction issues and stalls. Extra cost from TLB and/or data cache misses would be minimal (if any) so this could be regarded as an acceptable cost.

Interrupts: IRQs and FIQs would experience the same overhead as the undefined instruction traps. This would not only result in overhead on the interrupt IPC generation path but also in thread dispatching events resulting from timeslice expiration, and thread wake-ups. Another cost is that any advantage of the FIQ modes extra 5 banked registers is lost.

One of the benefits of this method is that it allows the banked abort SP to be preloaded with the current thread's kernel stack pointer on a thread dispatch so it doesn't need to be loaded on initial exceptions within the user space. However, the overhead it adds to the interrupt path, and the complications it generates in the implementation, justify the exploration for an alternative approach. Furthermore the preloaded SP does not really save on the exception handling path, since it just moves the loading from the exception entry point to its exit point.

Using an Unbanked Stack Pointer

The second alternative is to use an unbanked register as the current thread's kernel stack pointer. This requires an initial loading of the stack pointer when the kernel is first trapped into via an exception from the user space, but further exceptions within the kernel (even if they change privileged modes) can rely on the kernel stack pointer being valid.

The advantage of this method is that the cost of exceptions becomes evenly balanced. Also, the banked SP can now be utilised to store info relevant to the particular exception and the FIQ exception handler is able to make use of the extra 5 banked registers at its disposal.

The added cost is that when exceptions, other then a syscall, occur within the user mode, the unbanked register used for the kernel stack must be temporarily stored, until the kernel stack is set up and it can be stacked. The only trashable register available is the banked SP. The cost is then simply a single register move operation only taking a single cycle.

While this method presents no problem to an assembler implementation (Gauntlet) it violates the ARM C calling convention and would require hacking of the compiler if a high level language is used to implement the kernel.

Because of its simplicity and balanced cost across different exceptions the unbanked stack pointer method was chosen as the best approach for the Gauntlet implementation.

Multiple Kernel Stacks per Thread

An alternative to the the above general methods is to actually have a Kernel stack per thread per privileged mode. While this leans towards the ARMs hardware design the relevant exception mode stack would have to be loaded for each type of exception separately resulting in higher cost for nested exceptions then using a single unbanked kernel stack pointer. For this reason this option was not examined in detail.

3.1.3 Exceptions within the Kernel

Exceptions within the kernel are raised in a relatively controlled manner. Firstly, only a subset of exceptions are raised by the kernel, and secondly, the kernel stack is already set up, so the processor state can be stacked on entry to the exception.

Despite the controlled manner in which exceptions within the kernel are raised, a few issues still arise which require careful attention.

Interrupts in Kernel mode

Asynchronous exceptions raised due to interrupts require some attention owing to the exception model of the ARM. Generally we can classify the sections of the kernel code as one of the following:

- preemptable sections, and
- non-preemptable sections.

Preemptable sections do not present a problem, we simply unmask both the IRQ and FIQ signals and the kernel can be interrupted without a problem. The problem arises in the non-preemptable sections. For example, if exception handlers have the precondition that the kernel stack pointer is valid when the exception occurs within the kernel, this precondition will be violated the exception occurs between trapping into the kernel from user mode and setting up the kernel stack pointer, since it is not an atomic operation. To avoid this interrupts must be masked or the precondition of a valid kernel stack relaxed.

The IRQ exception presents no problem since the IRQ signal is masked on entry to any exception (it can then be unmasked once the kernel stack is loaded if and when required). The FIQ exception however is only masked by an FIQ exception. This means that an interrupt routed to the FIQ signal can preempt the kernel before its stack pointer is loaded violating the kernel stack precondition.

On the SA-1100 this problem can be bypassed by simply routing all interrupt sources to the IRQ signal via the processors interrupt controller. This approach, however, suffers from the following problems:

- The distinction between IRQ and FIQ can be used to simplify the handlers (hence making them faster), for example by routing all kernel interrupts (system timers) to one signal and all user interrupts (which result in interrupt IPCs) to the other.

- Once again the extra FIQ mode banked registers remain unutilised.
- It is also not sufficiently general, since not all ARM systems offer the ability to route all interrupt sources to either signal, leaving some sources hardwired to the FIQ signal.

A more general and elegant solution to this problem is to simply check the values of the I and F bits in the SPSR (IRQ and FIQ masks, respectively) at the start of the exception handler. The possible states and required behaviour FIQ handler are then:

F=0/I=0: Kernel is in a *"safe"* state, ie preemptable and the kernel stack is valid. Go about handling FIQ.

F=0/I=1: Kernel is in a *"intermediate"* state and the kernel stack has not yet been loaded. In this case the FIQ should load the kernel stack and stack the user state before handling the FIQ. It should also set the F-bit in the SPSR before storing it and adjust the return address so that when control is returned to the thread it skips over the masking of the F-bit as well as the loading of the kernel stack and stacking of the user state.

F=1/I=X: Not relevant since by definition the FIQ exception cannot be raised.

The nice thing about this solution is that with a simple comparison operation and branch (only taking 2 cycles if the compare fails) the FIQ handler can check if it interrupted the kernel in an *"intermediate"* state. If so it takes the branch and stacks the processor state on behalf of the preempted exception handler adjusting its return address to skip over the stacking.

Valid Kernel Exceptions

Assuming a bug-free implementation, only a subset of the exceptions can occur within the kernel:

Data Abort: During IPC operations (either during long IPC or when touching non-existent Thread Control Blocks).

IRQ/FIQ Interrupts: During preemptable kernel operations.

As mentioned above these exceptions (excluding the *"intermediate"* state case presented above) are relatively controlled. The kernel can guaranty in critical areas that they won't be raised, by masking all interrupt sources and by not touching potentially unmapped data memory locations. Because it has control over what sections of code these exception can be generated in, steps can be taken to ensure exception handler preconditions when raised in a privileged mode.

This way, the state of some of the register set can be well defined on re-entry into the kernel resulting from an exception. In particular it can be guaranteed not only that the kernel stack pointer is valid, but that other important values like a pointer to the current thread's control block (TCB) and the kernel miscellaneous data are loaded into defined registers.

3.1.4 Exception Entry/Exit

All exceptions (apart from syscalls) must stack the whole processor state (in preparation for a context switch) before handling the exception. Once the exception is handled and ready to exit it may take one of two paths (depending on thread priorities):

Return to Current Thread: The exception will return to the preempted thread if no context switch is required (pager/exception/interrupt IPC receiver is not ready to receive) or the context switch would result in dispatching to a lower priority thread (pager/exception/interrupt IPC receiver is a lower priority then the current thread). In this case the current thread's state is simply restored.

Switch to Another Thread: The exception results in a context switch since the thread to be dispatched is ready and at a higher priority then the current thread. In this case remaining information required to restore the currently running thread, should be stacked and the new thread dispatched.

Both cases present no problem to exceptions generated within the user mode. In the second case though, exceptions generated within the kernel require some attention once again. This results from the fact that the banked registers of the preempted kernel mode cannot be directly accessed if the exception changes processor modes. If no context switch results (the first case) then the banked registers are unchanged. However, if a context switch occurs (the second case) the dispatched thread may trap into the same mode the previous thread was in, trashing the banked registers. Two solutions exist to the problem:

Allow trashing of the banked registers: Since the kernel can control when exceptions within the kernel can be generated, the banked registers can be assumed to be trashed at any time during the preemptable sections of the kernel. They can either not be used (extreme) or they can be made invariant, that is, if preempted, a known value (like the current TCB) is reloaded into them by the unstacking procedure of the preempted thread.

Stacking of the banked registers: Alternatively the exception handler can switch back to the preempted kernel mode and stack the banked registers just before dispatching to the new thread when its next dispatched.

Which method is used depends on the requirements of the kernel in its preemptable states. If registers are scarce the stacking method can be used, alternatively where handling of nested exceptions is time critical the trashing method can be used.

3.1.5 Reset

The reset exception is generally irrelevant to the kernel since it results in the MMU being disabled and the PC being forced to address 0x0, which is a firmware address. However, since the sleep mode of operation raises a reset exception when exited, it must be addressed for sleep mode support. This requires a protocol to be established between the kernel, firmware and user space. The reset vector could possibly be used to facilitate such support. However, this is beyond the scope of this thesis.

3.2 Syscall Entry/Exit

Syscalls are a special case of exception used to request services from the microkernel. Syscalls handling differs from other exceptions in that only the user stack pointer is preserved across syscalls, the remaining general purpose registers contain either arguments or are scratch registers. This means that when a syscall is trapped, only the user SP, SPSR and LR (the users CPSR and PC respectively) need to be stacked.

3.3 Scheduling

Thread switching events can occur for one of three reasons:

Yield: Threads can yield the processor either by donating their remaining timeslice to a specific ready thread, or have the scheduler select the next ready thread.

Timeslice expiration: The current thread's timeslice expires and the scheduler selects the next ready thread.

Thread wake-up: A thread of higher priority then the currently running one reaches an IPC timeout and is dispatched.

Two forms of dispatching events are possible to handle the thread switch:

Specific Destination: This can result from a wake-up or by yielding to a runnable thread. The thread is dispatched directly in this case.

Schedule Next: This case results when the specified thread targeted by a yield is not runnable, or when the processor is yielded without donation (via timeslice expiration or otherwise). The scheduler selects the next runnable thread and then dispatch it.

3.3.1 Schedule Next

The L4 API specifies hard priorities with threads at the same priority being scheduled in a round robin fashion. The schedule-next operation must first find the highest priority runnable thread, then dispatch it. The simple way of doing this is to start at the highest priority ready queue (see Section 3.4.5) and check if its empty or not. If it is empty then the next priority ready queue down is checked and so on until a non-empty queue is found. Once a non-empty queue is found the head is dispatched and the queue pointer updated to point to the next thread in the queue.

A simple design optimisation is to keep track of the maximum runnable priority to avoid searching through the ready queues for a non-empty one. Though the extra few cycles it potentially adds to the IPC path may result in this optimisation being left out of the implementation.

3.3.2 Yielding

There are two situations where L4 dispatches threads without scheduler invocation:

- Every successful IPC operation (L4 IPC is blocking) forces a context switch between the two communicating threads. The thread which was blocked waiting for the IPC to take place gets dispatched with the remainder of the partner's time slice. On top of this IPC to a partner who is ready to run (not blocked waiting for IPC) results in the same form of dispatching.

- A thread can explicitly donate the remainder of its time slice to an arbitrary runnable thread. This feature is designed to be used for user-level scheduling. In an extreme case there would be a single highest-priority thread (the user-level scheduler) which schedules other threads by donating its time slices. That scheduler would be run once per time slice, about as often as all the other runnable threads combined.

3.3.3 Thread Dispatching

Thread dispatching involves switching the thread context (as well as address space context if the dispatched thread is in a different task). The operation simply involves storing the thread's state (via stacking or storing values in the current TCB) and then restoring the dispatched thread's state. One issue that arises with this is how to restore the thread's user register set from the stack. Different processor yielding events result in different sub-sets of the register set being stacked. Syscalls, for example, only restore the user's SP, CPSR and PC, while time-slice expiration restores the whole register set for the user. The way this can be dealt with is to stack the address of the register set restoring procedure. By doing so, when a thread is dispatched all context is switched, except the register set, at which stage the top of the kernel stack is popped into the PC, which effectively branches to the unstacking procedure.

3.3.4 System Timers

The flexibility of the SA-1100 OS timer and alarms allows two forms of system timer to be implemented for the kernel:

Fixed Interval Timer: A regular interval timer can be generated by using a single OS timer alarm register (with associated interrupt) to generate a clock with 1ms accuracy. Finer accuracy for fixed interval timers is impractical [Lie96c].

Variable Interval Timer: Variable interval timers can be implemented to give higher accuracy. Variable timers are only a problem when reprogramming the timer interval is expensive. This is not an issue with the SA-1100.

The variable timer method is most appropriate for the SA-1100 since in not only allows more accurate timing, but allows utilisation of multiple alarm registers.

The multiple alarm registers allow simplification of the alarm interrupt handlers. One alarm can be set on every schedule-next invocation (to the selected thread's time-slice) and another can be set to the earliest wake-up event. This way the time-slice alarm interrupt handler just calls the schedule next procedure while the wake-up event alarm register can jump directly to a wake-up handler. This leaves two alarm registers available, these could be used to efficiently implement multiple wake-up queues (see Section 3.4.5) or possibly to maintain a global clock for the user space.

3.3.5 Idle Thread

If no user threads in the system are runnable, the scheduler dispatches to the *idle thread*. This simply spins until the next exception is raised. It should be clear that the idle thread is a prime candidate switching to the idle power mode the SA-1100 has to offer, since only the kernel can invoke it. The idle mode offers reduced power consumption while maintaining low wakeup latency, making it a suitable replacement for a simple spin loop implementation of the idle thread.

3.3.6 Hard-Real-Time Support

Support for hard-real-time systems is of particular importance to an ARM implementation, since many application areas of the ARM involve various forms of real time constraints. While hard-real-time support by itself is not particularly difficult or new (a large number of hard-real-time kernels exist for ARM in both research and commercial communities) mixed systems that support general purpose time sharing and hard real time components are of growing interest [Bar97, BBH$^+$98]. Real time support introduces the following design/implementation considerations:

Bound Kernel Blocking Time: This can be achieved by either guaranteeing kernel operations are time bound, or by providing preemption points in unbound operations, maintaining an upper bound on the time the kernel cannot be preempted.

Controlled Interference: Interference is one of the more difficult aspects of real time systems. The typical push behind modern processor and operating system development is to increase average performance which often blows out the worst case. Caching is a prime example of this [LHH97]. While interference is a complex matter, measures can be taken to at least reduce the interference the kernel has on the system (by keeping cache footprints small, etc).

3.4 Data Structures

The number of data structures required by the kernel are small but they must be designed in such a way as to minimize not only their size but more importantly the cost of accessing the information contained in them or the structure they form. This minimises not only the resources they consume but the interference with the user space that they generate.

3.4.1 Thread Control Blocks

The *thread control blocks* (TCBs) contain all the relevant information associated with a particular thread as well as the thread's kernel stack. A large virtual array is used to allow fast lookup of a TCB address from the a *thread identifier* (TID). TCBs are allocated on demand via page faults. A read fault on a TCB causes a shared read only zero page (the NULL TCB) to be mapped in. This facilitates the fast checking of a threads existence in the IPC path. Write faults map in a fresh zeroed 4KB frame since they signify thread creation (via lthead_ex_regs).

3.4.2 Kernel Miscellaneous Data

The Kernel Miscellaneous Data contains miscellaneous global information concerning the current kernel state, required by the kernel, which doesn't fit in any other data structure. Information like the heads of the ready queues, memory allocator data, as well as the current thread's kernel stack pointer (from which the current TCB address can be generated from quickly) are stored here.

3.4.3 Task Table

The task table is a small array of words where each word represents a task state. The main motivation for a task table is to store a small amount of information regarding the state of inactive tasks (since inactive tasks have no TCBs). It can also be used for fast access to a small amount of task specific data for the more critical routines that need avoid a TCB access. At the least the task table is used to store whether a task is active or not and its chief (owner task).

3.4.4 Task Page Tables

The task page tables are simply the hardware traversed page table specified by the ARM architecture. A pointer to the page directory for each task is stored in the TCB of each thread in the given task.

Since the kernel mappings are a reserved part of the virtual address space all task page tables share a copy of the kernel mappings. Consistency is maintained by adding new kernel mappings to sigma0's page table and then lazily adding them to the page tables of other tasks on page faults within the kernel space.

A problem with this system is that when kernel mappings are flushed they must be flushed from all task address spaces, a slow operation. This problem was not examined in detail since the work presented in Chapter 4 provides an alternate form of consistency for kernel mappings.

3.4.5 Thread Lists/Queues

A number of lists and queues form a critical frame work of the kernel, responsible for maintaining thread states. Each list/queue is structured to keep the speed critical operations fast. Thread list/queue pointers are stored in TCBs.

Present List

The preset list contains all the active threads in the system, it is required to keep track of threads for the task deletion. The present list structure was adopted from Fiasco [Hoh99], a L4/ix86 compatible kernel. It is made up of a doubly linked circular list, this supports quick entry and removal of TCBs to/from the list. The partial ordering that is maintained is that for any task, if we start at lthread0:

To one side: All threads of the task are located, ending when a thread from a different task is reached.

To the other side: All threads of child tasks are located, ending when a task with a depth equal to or less then the given task is reached.

This partial ordering allows all threads of a task to be located in a sweep of one direction while all child task threads can be located in a single sweep in the opposite direction.

The ordering supports a fast preemptable task deletion operation. Ordering is maintained by adding new threads (via lthread_ex_regs, *a la* thread creation) to the right of the caller thread and adding new tasks (lthread0's via task_new) to the left of the caller task's lthread0. No pointer to the present list is required since all operations on it are relative to the syscall invoker.

Ready Queues

The ready queues (one for each priority) are required to maintain round-robin scheduling within each priority. Pointers within the kernel miscellaneous data keep track of the head of each priority queue with newly ready threads being inserted at the end of their priorities queue. Once a thread is scheduled, the relevant ready queue pointer is updated to point to the next thread in the queue.

Send Queue

A send queue is required per thread to keep track of threads polling to send to it to support the *open receive* IPC operation. The send queue is a double linked list sorted on priority and FIFO with equal priorities. Open receives then select the head of the send queue. This results in higher priority threads being sending to open receivers first.

Wake-Up Queue(s)

Support of real IPC timeouts (ie non-zero/-infinite) requires a wake-up queue, this queue contains a list of all threads waiting on a real timeout. With a fixed interval system timer at each tick the queue is examined for any threads with a timeout expiration. Current implementations like L4/MIPS and L4/ix86 make use of multiple unordered wake-up queues [Lie93].

While a detailed design was not examined, a possible approach for the SA-1100 port is to use 3 ordered wake-up queues each with their own OS timer alarm. The different queues could represent different wake-up events to distinguish between send/receive IPC timeouts and/or Long IPC page fault timeouts. An alternative is to use the queues in a cascade of time intervals so that the timer signals progression into the next next queue (with a smaller time interval). This allows the sorted insertions to be faster as the individual sorted queues are smaller. The design needs to be revisited and examined in further as future work.

3.4.6 Memory Management

Since the kernel contains some dynamic data structures some mechanism must exist inside the kernel for dynamically allocating and freeing memory. This dynamic memory allocation is required for the following reasons:

TCB Management: Requires allocation/freeing of 4KB frames for TCBs.

Page Table Management: Requires allocation/freeing of 16KB frames for PDs and 1KB frames for LPTs.

Mapping Database: Requires allocation/freeing 4KB frame which the mapping database then manages to provide more fine grain buffers for its own use.

The kernels memory management design should optimize for the *allocating* cases so thread/task creation as well as IPC mapping operations are kept fast. For this reason a separate freelist is maintained for each of the frame sizes the kernel must manage (1KB, 4KB and 16KB). To keep allocation fast, the design must either ensure that each freelist is kept non-empty, or that frames can quickly migrate between freelists. Since the freeing (deallocation) mechanism isn't time critical, most management of the freelist migration can be handled in this operation.

Migration of frames between the different freelists avoids the situation where the freelist of the requested allocation is empty even though a frame of that size might be available. This arises when either a larger frame size freelist is non-empty or enough contiguous frames are located in a smaller frame size freelist. Moving from a larger to a smaller frame size freelist is simple operation, however locating contiguous frames to form a larger frame to move to a larger frame size presents more of a challenge.

One possible solution is to search through the whole frame list in order to find the consecutive frames, this however is too inefficient. Another solution is to keep the freelists fully sorted. This will however put a lot of overhead of the freeing process and while the freeing operations are not so critical, making the operation preemptable proves difficult to implement.

Kernel Memory Manager

The solution presented (developed in collaboration with Espen Skoglund [Sko99]) is to keep parts of the frame list ordered. Frames are clustered in groups of four consecutive and properly aligned frames at the end of the freelists. A *cluster pointer* points to the first frame in the freelist, which belongs to a cluster, for each freelist. When, say a 16KB frame needs to be allocated from frames in the 4KB freelist, the frames in the first cluster of the 4KB freelist are removed, and the cluster pointer for that freelist is updated to the next cluster (if any) of the freelist. The invariant that must be maintained is that all frames in the freelist following the cluster pointer are organized in clusters. Keeping clusters in the end of the freelist also ensures that clusters are not unnecessarily split up when unclustered frames are available.

The freeing operation must then maintain the organisation of clusters within the freelist. This is achieved by checking the status of the frames which make up the freed frames cluster each time a frame is freed. If all three partner frames are found to be unallocated they together with the freed frame are concatenated and moved to the end of the freelist. If not all partner frames are unallocated the frame is simple added before the first cluster.

Kernel Memory Pool Size

While the simplest way of handling the kernel memory pool size is to use a compile/boot time parameter, it may not prove the most flexible method. Further examination of a kernel/sigma0 protocol to allow 4KB frames to flow between the two is worthwhile since it is a more flexible model. This allows dynamic balancing of memory allocation between the kernel and user spaces, avoiding hand tuning of compile/boot time parameterization. If required, parameterisation can simply be implemented by a sigma0 which reserves a fixed amount of memory for the kernel.

3.4.7 Mapping Database

To support the flushing of memory mappings a record of the hierarchical mappings/granting of memory between address spaces must be maintained. This is record keeping structure is knows as the *mapping database*. For an ARM implementation the following specific design issues must be addressed:

1. How to keep the mapping database as small as possible (Since typical ARM platforms and applications require lean memory consumption).

2. How to maintain support for real time systems via time bound or preemptable operations (as mentioned earlier, support of real time systems is a design goal).

3. How can the mixed page sizes of the ARM (or any other CPU for that matter) be supported.

While the first two design problems are general constraints of the whole design of the kernel the last provides an interesting problem which is not only applicable to the ARM implementation but to any L4 implementation on top of a processor supporting mixed page sizes, MIPS, Alpha, etc.

Mixed page sizes can not only increase system performance via higher TLB utility but can also improve memory consumption via reducing page table memory allocation. For example on the ARM 1MB sections only require a first level PD entry.

Work is currently underway by Espen Skoglund [Sko99] to provide a mapping database design and implementation within the above constraints. Once ready this design will be integrated into the Gauntlet implementation.

3.4.8 Mini-Data Cache and Read Buffer

These hardware enhancements only effect performance and potentially reduce interference with user data cache lines.

Mini-Data Cache

The mini-data cache is designed to reduce the impact of large data structures on the data caches working set which would otherwise flood the main-data cache. While the pagetable *cachable* and *bufferable* bits, specifying which of the data caches a data lines will be loaded into, could be controlled by the user space (via arguments to the *mapping IPC* operation), kernel data structures are typically large, require less associativity and make up a small proportion of the total system data. For this reason the initial design forces all user data into the main-data cache while all kernel data occupies the mini-data cache. In the future it would be worthwhile to properly access what benefits (or costs) this method generates.

Read Buffer

The read buffer again can be visible to the user space (it can be enabled or disabled for non-privileged access). It is debatable however, whether the user space could make suitable use of it. For this reason, like the mini-data cache is its reserved for kernel usage. While with a single cycle it can be used to pre-fetch data, as soon as the data changes another cycle must be consumed to invalidate the entry since the read buffer maintains no coherency with caches or main memory and overrides the data caches on loads.

It is expected that the read buffer could make use of SA-1100 load delay slots that cannot otherwise be filled.

Chapter 4

Fast Address Space Context Switches

This chapter details and expands on the work done in collaboration with Gernot Heiser [WH00] on achieving improved address space switching performance on ARM processors, in particular the StrongARM. The approach is based on exploratory work performed in October 1998 [Wig98a] and is similar to Liedtke's work on fast address-space switches on the Pentium processor [Lie95a].

4.1 The Problem of Fully Virtual Caches

The biggest challenge facing an L4 port to the StrongARM is the fact that its caches are fully virtual, yet have no form of tagging. This is compounded by the lack of fine grained invalidation control of the caches, which leads not only to heavy indirect costs (to reestablish the caches working set) but also heavy direct costs to invalidate/clean the caches, which blow out context switch times. These problems are what lead to the poor address space switching times of the Linux kernel. In the microkernel context, address space switches are more frequent [Lie93], so it is critical that the address space switching issue is addressed to result in any practical L4 implementation.

The heavy context switch costs arise from the fact that fully virtual caches are tied to the address space context. When an address space switch occurs (by switching to a thread in a different task in L4), the change in virtual to physical mappings requires all cache lines to be invalidated, after the dirty lines are written back to main memory. The only way to avoid this purging is to tag cache lines with an address space identifier (ASID) which increases the cache size and its associated costs.

This problem, while highlighted in the StrongARM (due to its high clock speed and large caches), is not unique to it. All ARM cores to date have fully virtual caches without any form of tagging. While earlier cores used write-through caches (hence avoiding the direct costs of cleaning dirty cache lines), the general trend of ARM processors is towards fully virtual write-back caches. This trend can be seen in other processors like the Alpha [Dig92] which in the new 21264 [Com99] line of processors uses fully virtual caches for both the data and instruction streams. This architecture however uses ASIDs to tag TLBs and caches to provide addressing context, and also offers such fine grained control mechanisms such as *invalidation* of all TLB/Cache entries matching an particular ASID. This approach leads to fast address space context switches at the expense of more complex silicon implementations.

4.2 Overview of a Solution

ARM Domains (see Section 2.1.5) mitigate, to a degree, the problem of TLB entries not being tagged with an address-space identifier. They allow the kernel to map in or invalidate large and non-contiguous regions of virtual memory efficiently. This supports fast address-space switches, *as long as the kernel can guarantee that there is no overlap in mapped address spaces.* If this guarantee can be made, then an address-space switch can be performed simply by reloading the DACR with an appropriate mask.

A similar observation can be made about caches. The StrongARM architecture observes domains and protection bits even for cached data and instructions, not only on cache misses. Hence, as long as there is no overlap in mapped address space between threads, it is guaranteed that a context switch does not leave stale data in the caches. Caches need not be flushed in that case.

In can therefore be concluded that fast address-space switches are possible on the StrongARM as long as the kernel ensures the above condition of no overlap between mapped address spaces. The core idea of the fast address space switching mechanism is a technique which ensures that this condition always holds.

4.3 General idea

Normally an address-space switch is performed by replacing the page table. In the case of the StrongARM, where the hardware reloads the TLB from a two-level page table, this is handled by reloading the *translation table base* register with a pointer to the new page directory. This, by itself, is fast. However, the two address spaces would overlap in general and TLBs and caches need to be flushed to insure coherence and correct operation.

For this reason the translation table base pointer is never changed. Instead the register points to a *caching page directory* (CPD) which contains entries from a number of different address spaces, each defined by its own page table.

Figure 4.1 shows the idea: On a TLB miss the hardware reloads the TLB by indexing into the CPD, which contains pointers to LPTs of various address spaces. Each of these address spaces is associated with a unique domain, and CPD entries are tagged with that domain. The DACR is set up to give *client* access rights to the domain associated with the address space of the presently executing thread, and *no access* to any other domains.

On a TLB reload, one of the following can happen when the hardware accesses a CPD entry:

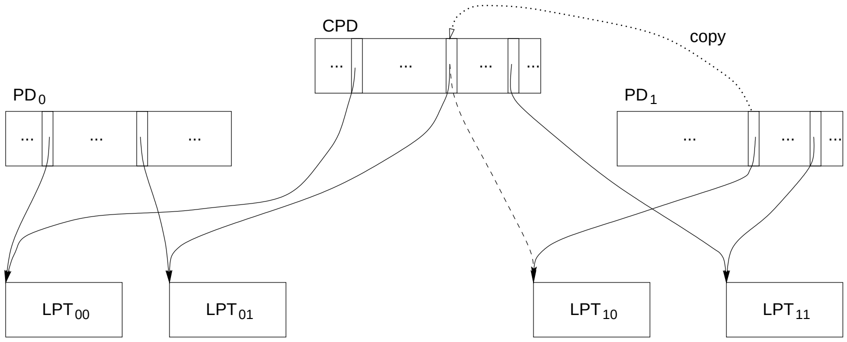

Figure 4.1: Caching page directory (CPD) and per-address-space page tables.

Translation fault: The kernel's fault handler indexes the current thread's PD and, if it finds a valid entry, copies that into the CPD and restarts the thread. The hardware will again attempt to reload the TLB and this time find a *valid section* or *valid page table* entry. Otherwise the kernel treats the access as a page fault.

Valid section entry: The entry is a *section* entry and the DACR indicates a valid access. If the permission bits allow the attempted access, the hardware loads the entry into the TLB and proceeds. If there is a permission mismatch, the hardware raises a *permission fault* which is interpreted as an invalid access by the kernel.

Domain fault: The kernel handles this as in the case of a *translation fault*: The entry is reloaded from the current thread's PD, replacing the existing entry in the CPD. In addition, the cache must be flushed of any remaining entries corresponding to the replaced CPD entry. If the current thread's PD does not contain a valid entry, the kernel treats the access as a page fault.

Valid page table entry: The entry is a *page table* entry and the DACR indicates a valid access. The hardware follows the pointer to some address-space's LPT which it indexes to find a PTE. If this PTE contains a *large* or *small page* entry, the hardware loads the entry or raises an *permission fault* as in the case of a *valid section* entry in the CPD.

Remembering that domains and permission bits are checked on every access, not only on a TLB load. Hence most of these faults can also occur when accessing cached data. This complicates handling somewhat, as in some circumstances the TLB must also be flushed, which wouldn't be necessary if the fault arose as the result of a TLB miss.

The CPD therefore caches PD entries from several different address spaces, similar to a direct-mapped software TLB [BKW94]. As long as there are enough domains for all threads wishing to execute, no flushes are required. Caches and TLBs must be flushed whenever a valid CPD entry is replaced.

However, the ARM only supports 16 domains, which is much less than the number of concurrently running tasks a general-purpose operating system is expected to support.

Domains therefore need to be recycled.

Domain recycling is required when a thread whose address space is not yet associated with a domain becomes runnable, and there are no free domains left. A domain must be recycled by preempting an address space. Before allocating a recycled domain to the new address space, the following must be done:

- TLBs must be flushed of all entries belonging to the recycled domain;
- the caches must be flushed of all lines corresponding to any of the just invalidated TLB entries;
- all of the domain's entries in the CPD must be invalidated.

This is obviously costly. A detailed analysis of cost is required to determine whether the TLBs and caches should be flushed completely or selectively.

4.4 Domain preemption

In order to implement domain preemption a strategy is required for selecting a victim. There are two rather different scenarios in which domain preemption becomes necessary:

1. the number of available domains is greater than or equal to the working set of address spaces but less than the number of active address spaces. In this case preemption is an infrequent event triggered by changes in the address-space working set;
2. the number of available domains is less than the working set of address spaces. In this case the system experiences domain thrashing and preemption occurs frequently.

The third possibility, the number of available domains being greater than or equal to the number of active address spaces, is irrelevant as no domain preemption occurs in that regime.

If it is assumed that address-space switching only occurs as a result of preemptive scheduling, and that address spaces are scheduled round-robin,[1] it is clear that the optimal domain-preemption strategies are:

- *least-recently used* (LRU) for case 1 above, and
- *most-recently used* (MRU) for the case of domain thrashing.

To find the least or most recently used address space it would be required, in theory, keep time stamps on domains. Under the above assumptions preemption can occur optimally if it can be determine in which regime (thrashing or non-thrashing) the system is in.

Note that in the non-thrashing case the preemption strategy is of relatively minor importance, as domain recycling is rare. If the "wrong" address space is preempted, this will

[1]The entity that gets scheduled is, of course, a thread, not an address-space. For simplicity the terminology of "scheduling an address space" is used, which is understood to mean "scheduling a thread running in that address space".

lead to some extra preemptions, until the working set is established. In the thrashing case, the wrong preemption strategy will lead to frequent costly preemptions. In particular LRU is the worst possible strategy in the thrashing case, given the above assumption on how address spaces are scheduled.

However, the above scheduling assumption is a gross oversimplification, particularly for L4. In L4, scheduling (as mentioned in Section 3.3.1) uses hard priorities with threads being scheduled round-robin scheduling within each priority. If several of the runnable threads belong to the same address space, address spaces will not be scheduled round-robin, even if threads are. Furthermore, the two circumstances described in Section 3.3.2 can result in thread dispatches without scheduler invocation.

Without prior knowledge of the communication and time-slice donation patterns of the client threads, no optimal strategy can be devised under these circumstances. The best that can be said is that an address space which has been used frequently (like the one of the scheduler thread, or a server which communicates with many clients) is most likely a bad choice for preemption, as it will most likely be used again in the immediate future. This argues in favour of a *least-frequently-used* (LFU) strategy. Note that least-recently used is no longer the worst possible algorithm in this case.

Note also that MRU is a dangerous algorithm to use, as it may fail to recycle domains which completely drop out of the working set, leading to thrashing of a small subset of domains.

What can be implemented efficiently? To answer that question it must be understood that two costs are involved:

1. the cost of the algorithm for selecting the victim domain. This cost is to be paid whenever domain preemption becomes necessary. As domain preemption is inherently expensive (due to the need to flush caches and TLBs) this cost is of minor importance;

2. the cost of maintaining the data structures required for victim selection. This cost must be paid at potentially each domain switch. As domain switches occur frequently, this cost is critical. In particular, a domain switch occurs on each cross-address-space IPC and a general goal is to keep the critical IPC path as short as possible.

In keeping with the general L4 philosophy, the fast cases should be kept fast. In the StrongARM's target application areas of embedded and portable systems, 16 domains should in most cases be sufficient to hold the address-space working set. In this case the overhead for domain recycling should be as close to zero as possible. In particular, any slowdown of basic IPC operations should, as long as the number of domains suffices, be kept to a minimum.

4.5 Domain preemption strategies

The obvious approach to implementing a domain preemption algorithm is to use a reference counting scheme on address spaces associated with domains, and then use an LRU approximation algorithm such as the well-known clock scheme for selecting a victim domain.

The above requirement of minimising IPC overheads strongly impacts on the question whether it is reasonable to keep accurate reference counts. A discussion of the reference counting approach, as well as an alternative approach based on *approximate reference bit maintenance*, follows.

4.5.1 Using proper reference counts

Maintenance of accurate reference counts requires that each domain be associated with a reference bit, and that this reference bit is set each time a switch to this domain is performed (that is, potentially at each IPC).

The kernel stores the reference bit in the *task table*, an array of one or two 32-bit words for each of the 1024 tasks (address spaces) supported. That word has to be loaded on each address-space switch. Maintaining the reference bit requires two extra cycles: for setting the bit and for storing the word back.

The advantage of this approach is that, at no extra cost, the reference bit can be changed to a reference count. Up to 6 bits can be allocated to the reference count in the task table, and adding to that field (modulo 2^6) is not slower than setting it to one. The cycles are then: for incrementing the reference and for storing the word back. Placing the reference in the *most significant bits* (MSBs) of the word insures overflow does not corrupt any other data in the word and allows the possibility of using the *carry out* (C-bit) flag of the CPSR to detect the wrap around case.

An implementor then has the freedom to implement either a clock algorithm (which would only check the reference count for being non-zero), or an LFU algorithm, which appears particularly attractive in the L4 context. Which one is used in the end is a small implementation detail, and the final decision will be made once the system can be benchmarked.

In either case a busy domain is occasionally preempted because its reference count has wrapped around to zero. This will only occur infrequently, so the cost of occasionally picking the wrong domain is irrelevant. Alternatively, the addition of a cycle (by utilising the conditional execution feature of the ARM instruction set) avoids the wrap around case. What effect if any this has on the performance of the kernel is left to benchmarking to determine.

The advantage of accurate reference counts is that it provides, at no extra cost, a good measure of whether or not the system is in a domain-thrashing regime. The total number of address-space switches can be obtained by summing all domain's reference counts. Monitoring the number of address-space switches per domain preemption seems to be the best indicator of thrashing. That figure, normalised to the number of domains, would be one in the case of maximum thrashing (e.g. n domains, $n + 1$ address spaces scheduled round-robin) and would grow very high if the working set fits into the available domains. This *thrashing criterion* would be very useful when taking special measures to reduce thrashing, such as using the PID register (see Section 4.6).

4.5.2 Maintaining approximate reference bits

There is an alternative way of approximately maintaining a reference bit which has zero overhead as long as the number of domains is sufficient, at the expense of increased domain-switching costs once preemption becomes necessary. The idea is based on the method used to implement page replacement algorithms on systems without a hardware-maintained reference bit in the TLB.

The method for maintaining the bit is tightly integrated with its use in the clock algorithm for selecting a victim domain. In addition to the reference bit the *DACR value* is also stored in the task table. On each domain switch, the DACR is loaded from that task table entry.

When a domain is initially assigned to an address space, the reference bit is turned on and the DACR value is set to giving *client* rights to the domain.

When a domain is needed but no free one is available, the clock algorithm is applied to the reference bits in the usual fashion: All reference bits are examined round-robin, and if one is found unset, that domain is preempted. Whenever a reference bit is encountered which is set, it is then cleared, and at the same time the DACR value of that domain is set to *no access* (for all domains).

Turning off access rights for unreferenced domains will force a *domain fault* next time that domain's associated address space is activated. The domain fault handler can recognise this special case as the fault will occur on a CPD entry which is tagged with the correct domain (and should therefore allow access). The handler turns on the reference bit, changes the domain's DACR value back to the original *client* setting, and loads that value into the DACR.

This algorithm introduces domain management overheads only when domain preemption becomes necessary (and for a short time after, until all the domains in the working set have been accessed.)

In addition, by keeping track on how many of entries from an address space's PD are presently cached in the CPD we can, before running the clock algorithm check whether there is any domain associated with an address space with currently no CPD entries. This would be the first candidate for preemption.

4.5.3 Discussion

The latter approach saves two cycles on each address-space switch (including each intra-task IPC operation). The costs are:

- Increased frequency, and thus overhead, of domain fault handling. However, this overhead only occurs once domain preemption becomes necessary, and is small compared to the cost of preempting a domain.
- Less flexibility in the choice of preemption strategy, essentially only the basic clock algorithm is available. As there could be performance advantages from using LFU, this cost could be significant.

- No real reference counts, and therefore no accurate criterion for the onset of domain thrashing. (The best that can be done is to count the number of domain preemptions per unit of time, which is probably not unreasonable.) Whether or not this cost is relevant will depend on how effectively the PID register can be used to reduce domain thrashing, see next section.

In the end, the decision will have to be made on the basis of measured performance. Therefore both variants should be implemented and benchmarked.

In addition to either method, by keeping track of how many of the number of PD entries presently cached in the CPD for each domain, before running one of the above algorithms it can be checked whether there is any domain with currently no entries in the CPD. This would be the first candidate for preemption since recycling it would require no flushes.

4.6 Using the PID register

The above strategy will work on any ARM processor with a TLB. On the SA-1100 (or any other ARM processor with the PID enhancement) the PID register can be used to improve the performance of the above strategy. The PID register can either be used to effectively increase the number of available domains (at some not insignificant cost) or to reduce the conflicts in the CPD. Either method if successful would reduce the amount of flushing required.

4.6.1 Sharing a domain for small address spaces

The PID register allows transparent allocation of small address spaces at non-overlapping virtual memory regions. However, such a re-mapping only affects the caches and MMUs view of addresses issued in the lowest 32MB. Any larger address will be translated normally, without use of the PID register. This means that small address spaces cannot easily share a domain.

However, the PID register can be used to implement a less expensive version of domain preemption. By keeping a *small bit* for each address space, indicating whether or not that address space has mappings above the 32MB limit. If a domain needs to be preempted and there are at least two small address spaces currently associated with a domain, these address spaces can share a domain.

In order to share a domain between several small address spaces, we need to allocate a 32MB slot (and a corresponding PID) for each in the lower 2GB of the address space (cores like the ARM920 allow allocation anywhere in the full 4GB address space). This slot can, in principle, be shared with any of the large address spaces, as the domain tag prevents invalid data accesses or cache incoherence. However, this would mean that two (or more) address spaces compete for the same CPD entries, which would defeat the purpose of the exercise. Hence a slot should be chosen which is not presently in use by any active address space.

If a small address space, different from the last active small one, is to be scheduled, all mappings left from the previous address space using the same domain need to be invalidated. This is done by invalidating all CPD entries corresponding to the previous small address space's slot. Both TLBs are then flushed. Caches, however, do not need to be flushed, as the management of the domains ensures that no aliases exist.

Cleaning the data cache has been identified as as being significantly more expensive than flushing the TLBs. Domain sharing of small address spaces should therefore incur less overhead than full domain preemption, although it is certainly not cheap. A thorough performance evaluation of the implementation will be required to determine whether it is worthwhile. In any case it should only be used if the system is definitively in domain thrashing mode.

Whether the system is thrashing domains can, in principle, be decided by counting the *number of address-space switches per domain preemption event*, Section 4.5.1 discussed how this figure can be determined if reference counts are being kept. Should the strategy presented in Section 4.5.2 be chosen in the end, domain thrashing must be identified (somewhat less reliably) by monitoring the domain preemption rate.

Note that this optimisation will only be beneficial if the operating system personality running on top of the microkernel cooperates by allocating user stacks within the first 32MB of virtual address space whenever possible.

4.6.2 Removing conflicts in the CPD

The PID re-mapping facility could alternately be used to minimise contention for the 32 CPD entries covering the first 32MB of the virtual address space. When replacing one of these first 32 CPD entries it is first check if the new entry can be relocated (using the PID register) to a CPD entry in an empty 32MB slot.

By turning a CPD replacement into a PID register load flushing of the TLBs and caches can be avoided. The first 32MB of that address space is then relocated using the allocated PID until either the Domain or the PID is recycled. While this method could be used on any user address space, the aliasing problem mentioned in Section 4.6 complicates matters. To keep the design simple the use of the PID is restricted to address spaces which don't contain any mappings in the relocation slots. This is a different style of *small address space* in which mappings are excluded from the region between the first 32MB and the upper 2GB. The operating system personality running on top of the microkernel can enhance the performance of the scheme by allocating user stacks at non-overlapping regions in the upper 2GB of the virtual address space.

4.6.3 Discussion

The above methods of utilising the PID register are both restricted to benefiting a form a small address space. Both also potentially complicate the original ARM generic design, which limits the potential benefits. Hence a full evaluation of both methods requires an implementation to explore their utility in a number of different application environments.

Both methods still require more thought experimentation with an implementation before their impact on system performance can be accessed. The shared domain requires some method of reloading the CPD entries of an inactive small address spaces when a cache line of that address space is written-back. To further complicate matters, the PID registers on data aborts is difference to that on prefetch aborts (due to one using the PC while the other uses the *fault address register*).

The utility of the PID optimisations however still appears attractive on the ARM920 core. This cores does not suffer problems such as TLB faults during write-back of inactive small address spaces, since the ARM920 tags data caches lines with their physical translations for write-back.

4.7 Impact on Kernel Data Structures

The fast address space switching mechanisms mentioned require some modifications to the data structures presented in Section 3.4.

4.7.1 Task Table

The task tags introduced are stored in the task table. This allows a single register to contain all the relevant task information for manipulation during context switches. The allocated domain and PID numbers as well as the reference count and small bits are stored here.

4.7.2 Domain Table

The *domain table* is a new data structure required to keep track information for domain management. What task the domain is allocated to (if any) as well as a reference count, of CPD entries belonging to the domain, is stored here. The domain table is small enough to be integrated into the kernel miscellaneous data.

4.7.3 Virtual Page Directory Array

One of the side effects of the CPD is the fact that the PDs of each task are no longer accessed via the hardware. Since the task PDs are now only accessed by the kernel they can be paged (like TCBs) using 4KB frames to save on memory consumption. This removes the need for a 16KB frame freelist. The entry storing the location of the tasks PD can also be removed from the TCB and loads replace with a macro to calculate the location of the PD.

Another side effect of the CPD being the only page directory visible to the hardware is the fact that it solves all coherency issues regarding kernel mappings. Dynamic kernel mappings no longer need to be made coherence across multiple task PDs, they can simply be added/removed directly to/from the CPD.

4.8 Sharing LPTs Across Address Spaces

A specific case of sharing could be optimised in the kernel which would avoid the flushing required when switching between address spaces, which would otherwise result. This specific case is where the mappings of two (or more) address spaces are identical for a particular 1MB section. In this case an LPT can be shared between them. By allocating the shared regions their own domain for each unique set of sharing, fast address space switches can then be supported by activating an address spaces exclusive domain as well as its shared mapping domains.

This takes advantage of the full power of domains. Unlike ASIDs, multiple domains can be active at once. To attempt the same kind of sharing on a platform providing ASIDs would require trapping faults when a thread moves from a shared to an exclusive mapping (and visa-versa) as well as when changing to a different shared mapping set.

The type of sharing that domains support perfectly match the model of *protection domains* within a *single address space operating system* [CLFL94, HEV$^+$98]. However the limitations of the ARMs design, in particular the fact that domains do not tag the native page size and the small number of domains provided, make it questionable whether such support on the ARM would improve performance, if not penalise it.

4.9 Aliases Within an Address Space

Since the caches are fully virtual, *aliases* (frames mapped into the same address space more then once) present a consistency problem. Consistency must therefore be maintained by the software. The recommended method for overcoming the aliasing problem of doubly mapped spaces is to set the aliased mappings as non-cachable/-bufferable. This slows down accesses of the aliased pages but insures consistency.

During the review of [WH00] it was suggested that the CPD mechanisms be expanded to cover the aliasing problem and be used to maintain cache coherency within address spaces. This however can only be achieved if the aliased pages are covered by different CPD entries, that is they do not share the same 1MB section. The method also leads to extra cache flushes which not only leads to interference within the aliased address spaces but other address spaces, penalising the performance of the system as a whole.

As a result it is questionable whether such a method would actually be practical in light of the costs. The user can maintain consistency by marking aliased pages as uncached, or if required the microkernel can force aliases as uncached. For these reasons exploration of the applicability of these ideas is left as future work.

4.10 Conclusions

The techniques presented in this chapter are functionally transparent above the microkernel interface, the resulting reduction in context switching overhead will never lead to incorrect execution. However, microkernel clients can minimise overheads by reducing address-space conflicts as much as possible.

Reduction of overhead by smart address-space allocation is especially important for the PID-specific optimisations for small address spaces (should they turn out to be beneficial). Standard address-space layout in Unix systems maps code and data segments at the low end of the address space, and the stack at the high end, thus making all address spaces "large".

To make efficient use of the hardware, an operating system personality running on top of the microkernel should allocate the stack at the top of the 32MB "small" address space whenever possible.

While the algorithms discussed in this chapter are geared specifically towards L4, they should be mostly applicable to other systems, such as monolithic Linux. The main difference is that address-space switching behaviour will be much better approximated by round-robin. This should make an approximate LRU preemption algorithm, like clock, perform better in the non-thrashing case (where it is somewhat less important) but possibly perform worse in the thrashing case. Other algorithms could be also be implemented and assessed, although their cost would almost certainly be higher.

Work is currently underway by a third party to adopt and test the applicability of the work outlined in this chapter into the Linux kernel. Since Linux uses the Unix address space model which would lead to high contention for CPD entries, it should provide a good worst case assessment of the performance of the presented mechanisms.

Chapter 5

Implementation and Evaluation

The goal of this chapter is to present an implementation which assesses the critical aspects of the kernel design. While a complete kernel implementation was beyond the time-frame of the thesis an initial base implementation was completed. By providing a base implementation some performance estimates can be made regarding the critical paths of the kernel. The implementation also provides a foundation from which a complete and optimised kernel can be built.

Unfortunately within the time constraints the kernel implementation could not be tested and benchmarked for a complete performance evaluation. However the analysis presented should represent a ballpark figure, with the calculated cycle times presented derived using the instruction timing application note for the StrongARM [Int98]. Hence they should reflect procedure timings given primed caches and TLBs, that is when no cache or TLB misses stall the processor during the procedure.

5.1 Overview

It was decided that kernel (everything except the initialisation) would be coded completely in assembly code. While this does not follow 'good software engineering' practices, if the kernel is sufficiently small, well commented and documented it should be maintainable by a small number of developers.

Assembler provides the complete control of the kernel required to implement most efficiently, not only in terms of performance, but in data structure size as well. An example of this is the fact that the kernel stack makes up the majority of the TCB size; without extreme care, stack size is difficult to keep small. Also keeping track of stack size when using higher level languages like C/C++ proves difficult. This is compounded by the need for assembly entry/exit stubs or special compiler extensions to avoid the ARM C calling convention during syscalls. The results is the same level of care and attention being required for a C implementation as an assembly one, without all the benefits. Implementing the kernel completely in assembly however must be justified with heavily optimised critical paths and lean stack usage.

5.2 Build Tools

Gauntlet is built using the *arm-linux* build of the latest GCC and GNU Binutils. This build provides ELF format is are maintained by the Linux/ARM development community. GCC is used simply to provide a preprocessor giving the kernel source code more structure and readability. The GNU tools were chosen not only for their good level of support from the Linux community but also for the free software licensing covering them (the GPL), to make Gauntlet easily accessible to all interested parties requires not only suitable licensing (Gauntlet is covered by the GPL), but suitably accessible tools to build it as well.

5.3 Executable Layout

The kernel executable is linked as an ELF binary. It consists of 3 program segments loaded at different *loading* (physical) and *execution* (virtual) addresses. This is done to move the complexity of extracting the different kernel segments to different physical addresses out of the kernel initialisation code and into an ELF loader (presumably the systems firmware). It is for this reason the executable is linked as an ELF binary (as well as ELF being the main development target maintained). The 3 kernel segments are their purposes are defined in the following sections.

5.3.1 Initialisation Segment (INIT)

Gauntlet's initialisation code is responsible for setting up and booting L4 and its initial server, Sigma0. Once the initialisation is complete, the code unmaps itself, freeing its resources, and switches to sigma0's entry point. The initialisation code is written in C to assist its maintainability (performance is not an initialisation issue) and also to assist kernel debugging via GDB stubs as described in Section 6.1.2.

The INIT segment must be loaded with identical virtual to physical mappings in a 1MB area of free RAM. The one to one mapping allows easy initialisation and enabling of the MMU.

5.3.2 Exception Vector Page Segment (EXCP)

The exception vector page segment is the result of ARM having a fixed exception vector at address 0x0 while the kernel resides in the top of the virtual address space. The EXCP segment consists of a single small 4KB page in which the exception vector table and a few speed critical procedures (all exception dispatchers) reside. Exception code must jump to the kernel TEXT segment via *long jumps* to run any code that does not fit into the EXCP segment 4KB page. What goes in the EXCP segment must be carefully selected to get the best utility from the full 4KB page.

By enabling the exception vector relocation on the SA-1100, the EXCP vector can be moved to address 0xFFFF0000 which precedes the TEXT segment. This allows the EXCP and TEXT code segments as well as some kernel data structures to be located in the top

64KB of the address space being covered by a single *large page* entry in each TLB. The *long jumps* could also be replaced with cheaper *direct branches*. Its physical loading address should be the start of the RAM area. For this reason vector relocation should be enabled on any ARM processor which supports it.

5.3.3 Text Segment (TEXT)

The remaining segment, the kernel TEXT segment contains all the kernel procedures that do not fit into the EXCP segment. The TEXT segment should be loaded physically in RAM after the EXCP segment.

5.3.4 Sigma0 and Initial Servers

It is expected that the Sigma0 initial server as well as any payload (other initial servers and/or data) will be linked into the kernel executable in a special sigma0 segment. However further work is required before something is settled on.

5.4 Kernel Code/Data Area

As mentioned above, the last 64KB of the address space are used to store the kernel code as well as the main static data structures, allowing it to be covered by a single 64KB large page entry in each TLB. The following shows the allocation of the 64KB page[1]:

```
/* From: Gauntlet/kernel/kernel.h */

/********************************
* Kernel Code/Data Area Offsets *
********************************/

#define KCDA_EXCP 0x0000 /* 4KB, Kernel EXCP Segment */
#define KCDA_TEXT 0x1000 /* 36KB, Kernel TEXT Segment */
#define KCDA_DATA 0xA000 /* 4KB, Kernel Misc Data */
#define KCDA_TT   0xB000 /* 4KB, Kernel Task Table */
#define KCDA_CPD  0xC000 /* 16KB, Caching Page Directory */
```

It can be seen that the total kernel code area is limited to 40KB, this should be plenty for the total kernel, but if required more space can be allocated in another part of the kernel reserved area of the address space (the top 256MBs).

[1] the offsets are relative to by the physical load address (platform dependent) and the virtual execution address of 0xFFFF0000.

5.5 Exception Dispatching

As mentioned in Section 2.1.2 exceptions trap into the kernel and force the PC to the relevant exception vector. Since the vector table forms a word array only a single instruction can be placed in each vector location. The last vector however, the FIQ vector can flow straight into the FIQ handler:

```
/* From: Gauntlet/kernel/exception.S */

/* Exception Vectors Table */
VECT(vector_table)
        b       reset_entry             @ Reset entry point
        b       undefined_dispatch      @ Undefined Instruction dispatcher
        b       soft_int_dispatch       @ Software Interrupt dispatcher
        b       prefetch_abort_dispatch@ Instruction fetch abort dispatcher
        b       data_abort_dispatch     @ Data abort dispatcher
        nop                             @ Reserved
        b       irq_dispatch            @ IRQ dispatcher
                                        @ FIQ dispatcher follows
/* end vector_table */
```

Above is Gauntlet's vector table residing at either relocated address of 0xFFFF0000 or 0x0. Once the exception is trapped (timing depends on the particular exception), it takes 2 instructions to issue the relevant branch. As mentioned, the FIQ dispatcher follows directly thus avoiding the branch cost.

5.5.1 Exception Entry

Exception dispatchers must first determine which mode the exception was raised in with something like the following:

```
/* example_exception: An example exception entry point.
 * Pre:                 Exception Raised.
 * Post:                Exception dispatched to the relevant handler,
 *                      r13 = SPSR.
 * Registers:           r0 = Current Thread's Kernel Stack Pointer. */
EXCP(example_exception)
        mrs     r13, spsr               @ Load SPSR
        tst     r13, #MODE_MASK         @ Test whether user or kernel mode
        bne     user_example_exception  @ if kernel, go on
/* Exception raised in kernel mode */
        EXCEPTION_STACK_KERNEL()        @ Stack the aborted kernel state
        ...
```

```
        <Rest of kernel example exception handler>
        ...
/* Exception raised in user mode */
user_example_exception:
        EXCEPTION_STACK_USER()          @ Stack the aborted user state
        ...
        <Rest of user example exception handler>
        ...
/* end example_exception */
```

The cycle times of the two cases are as follows:

Kernel: Each instruction takes a single cycle resulting in 3 cycles to get to the stacking macro. This totals to 5 cycles plus the exception trapping time before stacking starts.

User: The branch requires an extra cycle if taken. A total of 6 cycles plus the exception trapping time is required before stacking starts.

Within a Kernel Mode

As detailed in section 3.1.4, exceptions resulting in re-entry to the kernel have a valid kernel stack pointer so the common unbanked registers can be stacked immediately:

```
/* From: Gauntlet/kernel/macros.h */

/* EXCEPTION_STACK_KERNEL(): Stack the kernel state for an exception.
 * Pre:                      r0 = Current Thread's Kernel Stack,
 *                           r13 = SPSR.
 * Post:                     Kernel exception stack frame set up on current
 *                           threads kernel stack.
 * Registers:                r14 = Temp. */
#define EXCEPTION_STACK_KERNEL()
        PUSH(r0, r14)          /* Stack p_lr */
        stmfd   r0!, {r1 - r12} /* Stack r1 to r12 */
        PUSH(r0, r13)          /* Stack SPSR */
/* end EXCEPTION_STACK_KERNEL() */
```

Each PUSH() macro costs a single cycle while the *store multiple* (stmfd) takes 12 cycles (one for each register) giving resulting in 14 cycles for stacking. The r13 precondition results from the fact that exceptions must first check whether they were raised in the kernel or user mode. As a result the r13 is loaded with the SPSR prior to the macro.

The total time to stack the register set for exceptions within the kernel (once the exception is trapped and loads the PC) is 19 cycles if all unbanked registers are stacked, if the exception returns back to the preempted kernel context it may save a cycle per register untouched.

Within the User Mode

Exceptions within the user mode require more work since the full register set must be restored when the thread is next run (either by returning to the thread after the exception, or when its next dispatched if a thread switch occurs). The EXCEPTION_STACK_USER() macro sets up the current threads kernel stack pointer and then stacks the threads full user state:

```
/* From: Gauntlet/kernel/macros.h */

/* EXCEPTION_STACK_USER(): Stack the user state for an exception.
 * Post:                   User exception stack frame set up on current
 *                         thread's kernel stack.
 * Registers:              r13 = Temp. */
#define EXCEPTION_STACK_USER()
        mov     r0, r13                   /* Move cksp to Temp */
        LOAD_KDATA(r0)                    /* Load kpd into cksp */
        ldr     r0, [r0, #K_CURRENT_STACK] /* Load Thread's SP */
/* S */ PUSH(r0, r14)                     /* Stack p_lr */
        stmfd   r0, {r1 - r12, sp, lr}^   /* Stack User registers */
        PUSH(r0, r13)                     /* Stack r13 (was r0) */
        mrs     r14, spsr                 /* Load SPSR into r14 */
        str     r14, [r0, #-60]!          /* Stack SPSR */
/* end EXCEPTION_STACK_USER() */
```

Examining the cost of each operation:

1. Moving the contents of r0, 1 cycle.
2. The LOAD_KDATA() macro loads the address of the kernel data in 2 cycles:

```
/* From: Gauntlet/kernel/macros.h */

/* LOAD_KDATA(reg): Load address of Kernel Data into reg.
 * Post:            reg = Kernel Misc Data Pointer. */
#define LOAD_KDATA(reg)
        mov     reg, #0xFFFFFF00
        bic     reg, reg, #0x00005F00
```

```
/* end LOAD_KDATA() */
```

3. Loading the current threads kernel stack pointer, 1 cycle.
4. Stacking the exception modes LR (the user PC), costs 2 cycles due to the load delay slot being unutilised.
5. Stacking the r1 - r14 of the user register bank, a cycle per register totalling in 14 cycles.
6. Stacking the relocated r0, 1 cycle
7. loading the SPSR into a register (the user CPSR), 1 cycle.
8. Stacking the SPSR, 1 cycle.

23 cycles are required for stacking, totalling in 29 cycles to stack the full user state before an exception can be handled.

5.5.2 Thread Dispatching

It was described in chapter 3, the overall constraint on the design is that all exceptions potentially result in a context switch. Most of these take form of IPC and the rest either yield the processor to a particular thread or invoke a schedule-next procedure. In the IPC and direct switch cases the actually thread dispatching procedure is part of the critical path, its performance can therefore adversely effect the performance of these operations.

```
/* From: Gauntlet/kernel/macros.h */

/* DISPATCH_THREAD(dest, tmp1, tmp2): Dispatch to destination thread.
 * Pre:                             dest = Destination TCB,
 *                                  tmp1 = Temp 1,
 *                                  tmp2 = Temp 2,
 *                                  r13 = Kernel Misc Data Pointer,
 *                                  Current thread stacked (sp stored in
 *                                  tcb).
 * Post:                            Switch to destination thread's context
 *                                  and jump to its return procedure,
 *                                  r13 unchanged.
 * Registers:                       r0 = Current Thread's Kernel Stack
 *                                  Pointer.
 * Status:                          On the way, untested so far. */
#define DISPATCH_THREAD(dest, tmp1, tmp2)
/* Check if task context needs switching */
```

```
        ldr     tmp1, [dest, #T_PAGE_DIR]    /* Load dest page_dir */
/* R */ RdCP15_PTBase(tmp2)                  /* Load current page_dir */
        teq     tmp2, tmp1                   /* Check they are the same */
        beq     255f                         /* If different, go on */
/* Switch Task Context */
        WtCP15_PTBase(tmp1)                  /* Set new page_dir */
        FLUSH_CACHES_TLBS()                  /* Flush/Clean Caches/TLBs */
/* Switch Thread Context */
        ldr     r0, [dest, #T_STACK_POINTER] /* Load dest thread's SP */
        add     tmp1, dest, #TCB_SIZE        /* Set thread's empty SP */
        str     tmp1, [r13, #K_CURRENT_STACK]/* Load it into kernel Data */
/* Call Thread's Return Procedure */
        POP(r0, pc)                          /* Branch, return procedure */
/* end DISPATCH_THREAD() */
```

As seen above the dispatcher looks quite compact and potentially fast. However in the case of a intra task dispatch, which results in a address space switch, the FLUSH_CACHES_TLBS() macro is executed. This macro has a best case execution time of over 4 * 256 cycles (4 cycles per data cache line) to clean the data cache! clearly then an IPC path of 100 cycles or so is going to be devastated by the cost of the 1000 cycles or more from the associated address space switch, rendering an implementation of L4 on the StrongARM impractical.

This highlights the importance of the success of the fast address space switching mechanisms detailed in Chapter 4, though only a real system running on top of Gauntlet can assess how successful (if at all) these mechanisms will be.

In the case where the dispatched thread is within the same address space as the destination (inter task) the procedure costs approximately 10 cycles, a detailed cycle count is irrelevant since this procedure will change once the fast address space switching mechanisms are implemented. It is merely provided to give a ball part comparison of the intra vs inter task dispatching costs.

5.6 Syscall Handler

The *software interrupt* (SWI) instruction provides an immediate field for use by the syscall handler to specify information like the syscall number. However, accessing this immediate requires that the instruction be loaded into a register. This load pollutes the data cache (in the case of split caches) with instructions which may increase cache misses and resulting stalls. For this reason the syscall number is passed in a register.

As mentioned in Section 3.2, syscalls are handled in a fundamentally different way then other exceptions since the user enters syscalls in a synchronous, controlled manor. As a result, the registers have a defined allocation, via syscall arguments. Despite this, before the syscall is dispatched to the relevant handler, no registers can be trashed, since in the case of the IPC syscall, all registers are arguments to the IPC operation. For this reason, syscalls need to be dispatched before any registers are trashed. Once the thread is dispatched the syscall number register is used as the unbanked kernel stack pointer.

In this section the general syscall dispatcher and two syscalls are examined:

id_nearest(nil): The so-called *null* system call to get the callers *thread id* TID. This syscall returns to the caller after only accessing a single item in the callers TCB, its TID.

Short IPC: The fast IPC paths, that is send only and receive only where the partner thread is ready to carry out the IPC in both cases. Both are *register only* (short) IPC operations. These IPC operations represent the baseline performance of the kernel. The send only operation results in a thread switch to the receiver while the receive only operation returns to the caller after unstacking the 'short' message.

These syscalls are presented in detail since they effectively illustrate the costs of the syscall handler design. While other syscalls were implemented, in particular *thread_switch()* and *lthread_ex_regs()*, they are not presented here.

5.6.1 Dispatching Syscalls

Syscalls are dispatched by the following code:

```
/* From: Gauntlet/kernel/syscall.S */

/* soft_int_dispatch: Syscall dispatcher, invokes the relevant syscall.
 * Pre:              SWI exception raised in user mode,
 *                   r0 = syscall number.
 * Post:             Syscall dispatched OR
 *                   Exception IPC generated to threads
 *                   exception handler thread if syscall was invalid,
 *                   Register set unmodified. */
EXCP(syscall_align)
.align 5                                @ Alignment
        nop
        nop
        nop
        nop
        nop
EXCP(soft_int_dispatch)
        cmp     r0, #SYSCALL_LIMIT      @ Check syscall number is valid
        ldrlt   pc, [pc, r0, lsl #2]    @ Jump to syscall entry point
        b       invalid_syscall         @ Syscall >= SYSCALL_LIMIT
.word   ipc_entry                       @ Syscall = 0, ipc()
.word   id_nearest_entry                @ Syscall = 1, id_nearest()
```

```
.word   fpage_unmap_entry          @ Syscall = 2, fpage_unmap()
.word   thread_switch_entry        @ Syscall = 3, thread_switch()
.word   thread_schedule_entry      @ Syscall = 4, thread_schedule()
.word   lthread_ex_regs_entry      @ Syscall = 5, lthread_ex_regs()
.word   task_new_entry             @ Syscall = 6, task_new()
        nop                        @ Align to fill DCache line
/* end soft_int_dispatch */
```

The alignment forces the syscall handler addresses on a cache boundary. The addresses will be loaded into the data cache while the instructions stay exclusively in the instruction cache. This dispatching procedure takes 3 cycles for dispatching of a valid syscall: one for the comparison and two for the taking the branch. The *software interrupt* (SWI) instruction issue accounts for 4 cycles (one issue, three stalls), until the relevant exception vector is accessed, and then a further 2 cycles to issue and execute the branch to the above dispatcher. At this point 9 cycles have expired.

Once the syscall is dispatched the current threads kernel stack pointer is loaded and the user state stacked by the SYSCALL_STACK_USER() macro:

```
/* From: Gauntlet/kernel/macros.h */

/* STACK_USER_SYSCALL(): Stack the user state for a syscall.
 * Pre:                 r13 = Kernel Misc Data Pointer,
 * Post:                r0 = Current Thread's Kernel Stack Pointer,
 *                      Syscall Stack Frame set up on caller's Stack.
 *                      r13 unchanged.
 * Registers:           r14 = Temp. */
#define SYSCALL_STACK_USER()
        ldr     r0, [r13, #K_CURRENT_STACK] /* Load Thread's SP */
/* S */ PUSH(r0, r14)                       /* Stack svc_lr */
        stmfd   r0, {sp}^                   /* Stack usr_sp */
        mrs     r14, spsr                   /* Load SPSR into Temp */
        str     r14, [r0, #-8]!             /* Stack SPSR */
        SYSCALL_MASK_FIQ(r14)               /* Mask out FIQs */
/* end SYSCALL_STACK_USER() */
```

The stacking procedure stores the caller thread's SP, CPSR and PC, with below listed exceptions each instruction requires a single cycle to execute.

1. The commented 'S' indicates a stall due to the load delay slot not being filled, even though the PUSH() macro is only a single instruction requiring a single cycle, the

code accessed r0 directly after it is loaded resulting in a single stall being introduced into the pipeline.

2. The *store multiple* (stmfd) to stack the user SP, takes a single cycle for each register or two cycles for just one register, its employment is required since it is the only instruction that can access the user view of the register set.

3. The remaining exception is the SYSCALL_MASK_FIQ():

```
/* From: Gauntlet/kernel/macros.h */

/* SYSCALL_MASK_FIQ(tmp): Set the CPSR to SVC mode with I/F bits set.
 * Pre:                   tmp = Temp.
 * Post:                  In supervisor mode,
 *                        IRQ/FIQ signals masked,
 *                        Status flags (N,V,C,Z) cleared. */
#define SYSCALL_MASK_FIQ(tmp)
        mov     tmp, #(SVC_MODE | IRQ_MASK | FIQ_MASK) /* New CPSR */
        msr     cpsr, tmp                              /* Set CPSR */
/* end SYSCALL_MASK_FIQ() */
```

This macro requires 2 issue cycles and 3 stalls due to the *move status from ARM register* (msr) instruction, totalling 5 cycles .

All up the procedure takes 12 cycles bringing the total to 21 cycles. At this stage the relevant syscall handler can go about handling the syscall, depending on the syscall and its result control may be returned to the caller, or dispatch a new thread.

5.6.2 The Null Syscall

The "null system call" is the id_nearest() call with a nil thread as the destination argument, it returns the callers TID (thread id) as the result. It is the simplest, therefor fastest, system call in L4:

```
/* From: Gauntlet/kernel/syscall.S */

/* id_nearest: id_nearest entry point.
 * Pre:        r1 = Destination TID,
 *             r13 = Kernel Misc Data Pointer.
 * Post:       r1 = Type,
 *             r2 = Nearest ID
```

```
 * Registers:  r0 = Current Thread's Kernel Stack Pointer,
 *             r7 = Current Thread's TCB. */
EXCP(id_nearest_entry)
        SYSCALL_STACK_USER()                @ Stack User State
/* Test for get_My_ID case */
        teq     r1, #NIL_TID                @ Test if Dest NIL Thread
        bne     nearest_chief               @ If not find Nearest ID
        KSP2TCB(r7, r0)                     @ Get current TCB from ksp
/* Get threads own TID */
        ldr     r2, [r7, #T_MYSELF_TID]     @ Load threads TID into nid
/* Return */
        b       fast_syscall_return
nearest_chief:                              @ Find chief of dest id
        ...
        <Code to find the nearest chief>
        ...
/* end id_nearest_entry */
```

Up to and including the SYSCALL_STACK_USER() macro takes 21 cycles as calculated above. Then:

1. To test the destination argument for the nil value is a single cycle.
2. Since the branch is not taken it is effectively a nop, costing a single cycle.
3. The KSP2TCB() macro derives the callers TCB in a single cycle from its kernel stack pointer.
4. Loading the TID result takes a single cycle.
5. The branch to the syscall return procedure takes 2 cycles

In total the actual work of the syscall takes 6 cycles bringing the count up to 27 cycles. Now the flow of execution must return to the caller thread via the syscall return procedure:

```
/* From: Gauntlet/kernel/syscall.S */

/* fast_syscall_return: Return to thread from syscall in .excp section.
 * Pre:                 r0 = Caller Thread's Kernel Stack Pointer,
 *                      All unbanked registers contain syscall return
 *                      values.
 * Post:                User sp, CPSR restored,
 *                      returned from syscall with results in registers.
```

```
 * Registers:           r14 = SPSR. */
EXCP(fast_syscall_return)
        POP(r0, r14)            @ Unstack SPSR
/* S */ msr     spsr, r14       @ Load SPSR
        ldmfd   r0, {sp, pc}^   @ Restore User sp, CPSR and pc
/* end fast_syscall_return */
```

A step by step breakdown of the above code is:

1. The POP() macro loads the threads CPSR into a register in a single cycle.
2. Moving the popped register into the SPSR takes two cycles due to the unutilised load delay slot adding a single stall.
3. The final *load multiple* (ldmfd) restores the user threads state in a single instruction. The timing of this instruction is not clear. The application note says the address loaded into the PC will be executed after all registers are loaded, at a single cycle per register this would be 2 cycles. However, the fact that the CPSR is touched would suggest that the operation would introduce at least 3 stalls into the pipeline as suggested. This would result in the instruction taking 4 cycles (one issue, three stalls), so this is taken as being the case until information shows otherwise.

The return procedure takes then 7 cycles to complete bringing the total number of cycles for the null syscall to 34 cycles or approximately 165ns @ 206.4MHz, the SA-1100's top clock rate. This prediction assumes no misses in the TLBs or caches.

5.6.3 The Fast IPC Paths

The *fast IPC paths* are the *receive only* and *send only* operations when the IPC partner is ready to undergo the IPC, that is when it's stacked and *polling* or *waiting* respectively.

These operations represent the critical factor of the microkernel performance and hence must be fully optimised. The paths presented are only fairly optimised and still require some work but should provide a ballpark figure of the expected performance as well as highlight the problem areas in the paths.

Send Only IPC

The biggest challenge with the fast send only path is the lack of registers at the disposal of the the send operation. This is compounded by the fact that most of the load delay slots are unutilised causes stalls.

Below is the code of the fast send only IPC path, followed by a quick evaluation of its performance:

```
/* From: Gauntlet/kernel/syscall.S */
```

```
/* ipc_entry: IPC entry point.
 * Pre:       r1 = Destination Thread's ID,
 *            r2 = Send Descriptor,
 *            r3 = Receive Descriptor,
 *            r4 = Timeouts,
 *            r5 - r12 = Short Message (to send),
 *            r13 = Kernel Misc Data Pointer.
 * Post:      r2 = Source Thread's ID,
 *            r3 = Message Dope + CC,
 *            r5 - r12 = Short Message (received), OR
 *            Caller stacked and polling or waiting on partner.
 * Registers: r0 = Current Thread's Kernel Stack Pointer,
 *            r1 = Destination Thread's Control Block,
 *            r2 = Temp/Current Thread's Control Block (receive phase),
 *            r13 = Temp,
 *            r14 = Temp/Current Thread's Control Block (send phase). */
EXCP(ipc_entry)
        SYSCALL_STACK_USER()            @ Stack User State
/* Send operation */
        cmn     r2, #NEG_INVALID        @ Test for send operation
        beq     ipc_receive_only        @ If sndd != INVALID, go on
/* Test valid destination */
        TID2TCB(r14, r1)                @ Load up partners TCB to Temp
        cmn     r14, #NEG_KERNEL_TCB_CHECK @ Check partner TCB range
        ldrb    r13, [r14, #T_COARSE_STATE]@ Load partners coarse state
/* R */ bhs     no_partner              @ If Kernel TCB, no_partner
        teq     r13, #CS_UNUSED_TCB     @ Test if partner is alive
        beq     no_partner              @ if dead, no_partner
        ldr     r13, [r14, #T_MYSELF_TID]  @ Load up partners TID
/* S */ teq     r1, r13                 @ Test if partner TID matches
        bne     no_partner              @ If version differs, no_partner
        mov     r1, r14                 @ Move partner TCB into dtcb
        KSP2TCB(r14, r0)                @ Load callers TCB into stcb
/* Is partner ready to receive from caller? */
        ldrb    r13, [r1, #T_FINE_STATE]   @ Load partners fine state
/* S */ teq     r13, #FS_WAITING        @ Test if partner waiting
        bne     receiver_not_ready      @ If receiver ready, go on
        ldr     r13, [r1, #T_PARTNER]   @ Load who partners waiting on
/* S */ teq     r13, #NULL              @ Test for open wait
        teqne   r13, r14                @ If closed, Test waiting on us
        bne     receiver_not_ready      @ If open/waiting on us, go on
```

```
/* Is this a short/non-fpage/non-deceiving IPC? */
        teq     r2, #NULL                   @ Test if short and fast send
        bne     not_short_send              @ If not, not_short_send
/* Short send */
GLOBAL(short_send)
        cmn     r3, #NEG_INVALID            @ Test for receive operation
        bne     stack_after_send            @ If no receive, go on
GLOBAL(finish_send)                         @ temp return point
        mov     r13, #IPC_OK                @ Load IPC_OK result
        PUSH(r0, r13)                       @ Stack IPC CC result
        adr     r13, ipc_send_return        @ Load return address into Temp
        PUSH(r0, r13)                       @ Stack return procedure addr
        str     r0, [r14, #T_STACK_POINTER]@ Store callers stack pointer
/* Switch to receivers context */
        mov     r13, #FS_READY              @ Set receiver to ready
        str     r13, [r1, #T_FINE_STATE]    @ Store receivers fine state
        LOAD_KDATA(r13)                     @ Load Kernel Misc Data Pointer
        INSERT_READY_QUEUE(r1, r3, r4)      @ Add to ready queue, if removed
        DISPATCH_THREAD(r1, r14, r3)        @ Switch to receiver
/* Receive only IPC operation */
ipc_receive_only:
        ...
        <IPC receive code>
        ...
/* end ipc_entry */
```

Stack caller state: This is the standard syscall stacking method costing 21 cycles.

Check for send operation: A comparison and branch checks for the send operation. This costs 2 cycles if a send phase is part of the IPC.

Test partners coarse state: This section of code tests the coarse state of the partner. The TCB of the partner is first calculated using the TID2TCB() macro:

```
/* From: Gauntlet/kernel/macros.h */

/* TID2TCB(tcb, tid): Return the address of the TCB for the given TID.
 * Pre:               tid = contains a valid TID (ie non-kernel tid).
 * Post:              tcb = address of TID's TCB,
 *                    tid unchanged. */
#define TID2TCB(tcb, tid)
        mov     tcb, tid, lsl #TCB_SIZE_LOG2
```

```
        orr     tcb, tcb, #TCBA_VBASE
/* end TID2TCB() */
```

This takes only 2 cycles but results in kernel TIDs returning TCB addresses in the main kernel code/data area, for this reason the TCB range check is required. One the partner TCB address is loaded the following tests are carried out:

1. TCB range is checked to see if the partner is a kernel thread.
2. The coarse state (the TCBs state) is checked to see the partner is an active thread.
3. The partners version number is checked

The resulting total cost is 13 cycles.

Test partners fine state: If the partner is alive and the thread the caller is after, its fine state is checked as follows:

1. Checks the partner is *waiting* on a sender.
2. Checks who the partner is waiting the caller, or anyone (*open receive*).

All up 9 cycles is spent here.

Test for fast send case: Tests whether the caller is doing a fast send (short with no mappings). This takes 2 cycles if its the fast send.

Send phase: If there is no receive phase the sender is stacked to be dispatched at a later stage. This takes 7 cycles.

Switch to receiver: Switching to the receiver requires the following:

1. Marking the receiver as ready and inserting it into the ready queue via the INSERT_READY_QUEUE() macro. If the receiver is still in the ready queue the macro only takes 5 cycles. Giving 7 cycles all up.
2. Dispatching to the receiver as examined above takes approximately 10 cycles if the receiver is in the same task as the sender. The receiver will be dispatched to the *ipc_receive_return* procedure:

```
/* From: Gauntlet/kernel/ipc.S */

/* ipc_receive_return: Return to thread from IPC receive phase.
 * Pre:                r0 = Caller Thread's Kernel Stack Pointer.
 * Post:               User sp, CPSR restored.
 * Registers:          r14 = Temp.
```

```
 * Status:             Untested. */
EXCP(ipc_receive_return)
        ldr     r3, [r0], #8    @ Unstack IPC CC result
        POP(r0, r14)            @ Unstack SPSR into Temp
/* S */ msr     spsr, r14       @ LOAD SPSR from Temp
        ldmfd   r0, {sp, pc}^   @ Restore User sp, CPSR and pc
/* end ipc_receive_return */
```

This procedure loads the return result and restores the receiver threads context in 8 cycles.

The grant total is approximately 79 cycles, a little more then 0.38us at 206.4MHz, again this result assumes no TLB/cache misses. This prediction proves promising however the dispatching cost will blow out from approximately 10 cycles to well over 1000 if the receiver is in a different task to the sender with the standard coherency method of flushing everything.

Receive Only IPC

The major differences between the fast receive only path and the fast send only path are:

- The message is stacked in the *polling* sender, as a result a large amount of scratch registers are available.
- The syscall returns directly to the caller, as a result no context switch is required.

These differences make the fast receive path much easier to code. Below is the code of the fast receive only IPC path, followed by a quick evaluation of its performance:

```
/* From: Gauntlet/kernel/syscall.S */

/* ipc_entry: IPC entry point.
 * Pre:       r1 = Destination Thread's ID,
 *            r2 = Send Descriptor,
 *            r3 = Receive Descriptor,
 *            r4 = Timeouts,
 *            r5 - r12 = Short Message (to send),
 *            r13 = Kernel Misc Data Pointer.
 * Post:      r2 = Source Thread's ID,
 *            r3 = Message Dope + CC,
 *            r5 - r12 = Short Message (received), OR
 *            Caller stacked and polling or waiting on partner.
 * Registers: r0 = Current Thread's Kernel Stack Pointer,
```

```
 *           r1 = Destination Thread's Control Block,
 *           r2 = Temp/Current Thread's Control Block (receive phase),
 *           r13 = Temp,
 *           r14 = Temp/Current Thread's Control Block (send phase). */
EXCP(ipc_entry)
        SYSCALL_STACK_USER()             @ Stack User State
/* Send operation */
        cmn     r2, #NEG_INVALID               @ Test for send operation
        beq     ipc_receive_only               @ If sndd != INVALID, go on
        ...
        <IPC send code>
        ...
/* Receive only IPC operation */
ipc_receive_only:
        cmn     r3, #NEG_INVALID         @ Test for receive operation
        beq     fast_syscall_return      @ If rcvd != INVALID, go on
/* Is receive open or closed? */
        tst     r3, #OPEN_RECEIVE        @ Test if receive is open
        bne     open_receive             @ If so, open_receive
/* Test valid sender */
        TID2TCB(r2, r1)                  @ Load up partners TCB into Temp
        cmn     r2, #NEG_KERNEL_TCB_CHECK @ Check partner TCB range
        ldrb    r14, [r2, #T_COARSE_STATE]@ Load partners coarse state
/* R */ bhs     kernel_sender            @ If Kernel TCB, kernel_partner
        teq     r14, #CS_UNUSED_TCB      @ Test if partner is alive
        ldr     r5, [r2, #T_MYSELF_TID]  @ Load up partners TID
/* R */ beq     no_partner               @ if dead, no_partner
        teq     r1, r5                   @ Test if partner TID matches
        bne     no_partner               @ If version differs, no_partner
        mov     r1, r2                   @ Move partner TCB into dtcb
        KSP2TCB(r2, r0)                  @ Load callers TCB into rtcb
/* Is partner ready to send to caller? */
        ldrb    r14, [r1, #T_FINE_STATE]  @ Load partners fine state
/* R */ ldr     r5, [r1, #T_PARTNER]     @ Load who partners polling
        teq     r14, #FS_POLLING         @ Test if partner polling
        bne     sender_not_ready         @ If sender ready, go on
        teq     r5, r2                   @ Test if waiting on caller
        bne     sender_not_ready         @ If polling on caller, go on
/* Is this a short/non-fpage/non-deceiving IPC */
GLOBAL(do_receive)
        teq     r3, #NULL                @ Test if short and fast receive
        bne     not_short_receive        @ If not, not_short_receive
```

```
/* Short receive */
GLOBAL(short_receive)
/* Remove sender from receivers send queue */
        ldr     r14, [r2, #T_SEND_START]  @ Load callers send queue head
/* R */ ldr     r5, [r2, #T_SEND_END]     @ Load callers send last in Temp
        teq     r14, r1                   @ Test if sender is head
        ldr     r14, [r1, #T_SEND_NEXT]   @ Load senders send queue next
        ldr     r13, [r1, #T_SEND_PREV]   @ Load senders send queue prev
/* Sender is head of send queue */
        streq   r14, [r2, #T_SEND_START]  @ Callers start is senders next
/* Sender is not head of send queue */
        strne   r14, [r13, #T_SEND_NEXT]  @ Prevs next is senders next
        teq     r5, r1                    @ Test if sender is last
/* Sender is end of send queue */
        streq   r13, [r2, #T_SEND_END]    @ Callers end is senders prev
/* Sender is not end of send queue */
        strne   r13, [r14, #T_SEND_PREV]  @ Nexts prev is senders prev
/* Unstack short message from sender */
        ldr     r3, [r1, #T_STACK_POINTER]@ Load Pointer to message
/* R */ mov     r13, #IPC_OK              @ Set sender CC to return value
        ldm     r3!, {r5 - r12}           @ Restore short message
/* Stack senders return state */
        ldr     r14, [r3], #8             @ Unstack senders rcv descriptor
/* S */ cmn     r14, #NEG_INVALID         @ Test sender for receive
        bne     stack_after_receive       @ If not receiving, go on
GLOBAL(finish_receive)
        PUSH(r3, r13)                     @ Stack ipc CC result
        adr     r14, ipc_send_return      @ Load return address into Temp
        PUSH(r3, r14)                     @ Stack return procedure addr
        str     r3, [r1, #T_STACK_POINTER]@ Store senders stack pointer
        mov     r14, #FS_WAITING          @ Set senders state to waiting
        str     r14, [r1, #T_FINE_STATE]  @ Store senders state
        LOAD_KDATA(r13)                   @ Load Kernel Misc Data Pointer
        INSERT_READY_QUEUE(r1, r3, r14)   @ Add to ready queue, if removed
/* Set receivers return values */
        ldr     r2, [r1, #T_MYSELF_TID]   @ Load Senders TID into src
        mov     r3, #IPC_OK               @ Set receivers CC value
/* Return flows into fast_syscall_return */
/* end ipc_entry */
```

Stack caller state: This is the standard syscall stacking method costing 21 cycles.

Check for send operation: A comparison and branch checks for the send operation. This costs 3 cycles if the IPC has only a receive phase.

Check for receive operation: A comparison and branch checks for the receive operation. This costs 2 cycles if the IPC has a receive phase.

Check if open or closed receive: If the receive is closed this takes only 2 cycles.

Test partners coarse state: This section of code tests the coarse state of the partner, similar to the fast send path. The resulting total cost is 12 cycles, unlike the send path all load delay slots are filled.

Test partners fine state: If the partner is alive and the thread the caller is after, its fine state is checked as follows:

1. Checks the partner is *polling* for a receiver.
2. Checks the partner is *polling* the caller.

All up 6 cycles is spent here.

Test for fast receive case: Tests whether the caller is doing a fast receive (short with no mappings). This takes 2 cycles if its the fast receive.

Remove sender from send queue: 10 cycles are required to remove the sender from the receivers send queue.

Unstack message: Unstacking the short message takes 10 cycles in total.

Marking sender as ready: Here the sender state is setup to return from the IPC call.

1. The sender must be checked for a receive operation. If there is none its state is stacked in 10 cycles.
2. It is marked as ready and inserted into the ready queue. Again 7 cycles all up if the sender is still in the queue.

17 cycles in total are required to set up the sender to be dispatched at a later stage.

Returning to caller: Returning to the caller requires the following steps:

1. Setting the results of the sender ID and message dope, 2 cycles.
2. Flowing into the *fast_syscall_return* procedure, 7 cycles.

A total of 9 cycles is required here.

The fast receive path then has a total of approximately 94 cycles or about 0.46us at 206.4MHz with no TLB/cache misses. Again the predictions prove promising.

5.7 Discussion

The estimated results clearly show the potential for high performance from the kernel design and implement framework presented, however they do not assure it.

One thing that is certain from the calculations is that the high address-space context switching cost has a critical influence in the IPC path and thus the performance of the kernel. This cost must be addressed and reduced.

5.8 Current Status

While the implementation is not in a complete state it is ready to undergo testing and benchmarking of the syscall and exception handlers presented in this chapter. To do this requires the implementation of the user space syscall stubs with some method of time-stamping. Neither the stubs nor the user space code to test and benchmark the syscalls could be prepared by the thesis deadline. However, following is the proposed method for providing initial benchmarks.

Since no cycle counter is available on the SA-1100 and the OS timers frequency is approximately 50 times slower then the maximum clock rate accurate benchmarks present something of a problem. However with the available hardware facilitates easy access to some of the *general purpose input/output* (GPIO) lines. These lines can be set in three instructions taking the same number of cycles: one to load the GPIO register base, one to load the relevant GPIO number into a register, the last to store the result to the relevant GPIO register and set/clear the GPIO line. The GPIO transition then occurs approximately 20ns later (4 cycles). The benchmarking then simply involves using test equipment to monitor a GPIO line and running something like the following syscall stub code:

```
/* EXAMPLE_SYSCALL_STUB(tmp): Example syscall benchmarking stub.
 * Pre:                       Arguments and syscall number loaded in
 *                            registers,
 *                            tmp, r14 = trashable registers (not used as
 *                            arguments).
 * Post:                      Syscall handled and time using GPIO */
#define EXAMPLE_SYSCALL_STUB(tmp)
        mov     tmp, #GPIO_VBASE       /* GPIO Registers Virtual Base */
        mov     r14, #BENCHMARK_GPIO   /* Load Benchmark GPIO number */
        str     r14, [tmp, #GPIO_GPSR] /* Set GPIO */
        swi     0x0                    /* Trap syscall */
        mov     tmp, #GPIO_VBASE       /* GPIO Registers Virtual Base */
        mov     r14, #BENCHMARK_GPIO   /* Load Benchmark GPIO number */
        str     r14, [tmp, #GPIO_GPCR] /* Clear GPIO */
/* end EXAMPLE_SYSCALL_STUB() */
```

The monitoring equipment can then time the period for which the GPIO line goes high and use this to calculate the syscalls time, by accounting for the 2 cycles after the trap (the SWI instruction) to load the tmp and r14 registers. While syscalls like IPC cause a thread switch the receive threads, the code after the trap is the same resulting in the same net effect of clearing the GPIO.

This should provide fairly accurate benchmark figures but further experimentation is required once the kernel and support code is ready for benchmarking.

Chapter 6

Future Work and Conclusions

6.1 Further Work

The following is a partial list of the future work that lies ahead.

6.1.1 Completion of the Design and Implementation

The most immediate further work is the completion of the kernel design and implementation. This is ongoing work and should be completed during the summer-break of 1999/2000.

Design Completion

- Scheduling and preemptability (wake-up queues, etc)
- Using the PID register (a detailed design analysis and implementation/s)
- Examining the use of shared 1MB sections with unique domains
- Exploring the extension of the *fast address-space switching* mechanism to implement cache coherency (aliasing problem) within an address space

Finishing the Implementation

Once a the design is completed (or in parallel as is the case), the completion of the implementation is simply a matter of going through the motions. A (incomplete) list of the work remaining is as follows:

- Implementing ARM GDB stubs for kernel debugging
- Implementing the Mapping Database.
- Implementing scheduling, preemptability
- Implementing Task delete, Long IPC, flushing of mappings

6.1.2 Kernel Debugger Support

While debugging support of the kernel is important for development of the implementation, a number of challenges are faced with supporting debugging of L4. The main ones are listed below:

- Supporting debugging of a completely assembly coded kernel
- Maintaining the debugger, i.e., keeping it up to date with changes to the kernels internals
- Keeping the debugger small and/or removable

The GNU Debugger (GDB) stubs provide most of the above and also allow the developer to select from a large number of debugger front ends to run on the host machine. Since the stubs are very small they can be left in the kernel without worrying about removing them (to reduce the kernel footprint).

The remaining challenge is how to debug assembly in light of the fact that the assembler has no concept of data structures. However if C is used for kernel initialisation (as in Gauntlet) all the kernel data structures are contained in the header files and can be used to enable the debugger to examine them easily. Also by using externs the debugger can be made aware of assembler and linker symbols.

At this time however GDB stubs for the ARM are not suitable for debugging kernel space (since they are unaware of the privileged mode register banking) and as such is beyond the scope of this thesis.

6.1.3 Further Benchmarking, Optimisations and Features

Once a complete implementation is available, real benchmarking can be undertaken to highlight what areas of the kernel need further optimisation. Porting real systems like L^4Linux [HHL$^+$97] and the Mungi [HEV$^+$98] single-address-space operating system to Gauntlet will provide some real system environments to benchmark. Gauntlet will provide an ideal implementation base on which to port the up coming Version 5 API of L4 which will be defined during the summer.

6.1.4 Further Related Work

As well as work directly relating to the design and implementation of Gauntlet on the SA-1100 a number of indirectly related projects have grown out the work covered in this thesis.

Firmware support

As mentioned in Section 3.1.5 a kernel-firmware protocol needs to be developed to support the sleep low power mode of operation provided by the processor. This is not only required for support of the sleep mode, to support user space processor speed selection also requires firmware support since some of the code must run in the firmware address space. The basic requirement then is for protocols to be established between the firmware-kernel as well as the firmware-user.

Porting Gauntlet to the ARM9

The ARM920 core [ARM99] is more suited to L4 and the presented design in many ways due to its improved operating system support.

- Selectable write-(back/through) caching using the bufferable bit of the page table.
- 7bit PID relocation (relocate first 32MB to any slot in the address space)
- Cache and TLB entry lock-down
- More flexible cache control

In particular the more flexible cache control offered allows more efficient implementation of the cache coherency procedures to support the fast address space context switches presented in Chapter 4.

Rethinking the ARM Architecture

While the ARM is the most commonly used processor for embedded systems which typically either run stand alone programs or do not require multiple address spaces it provides very low power/cost implementations and is finding use in more desktop style applications such as set-top boxes. An interesting future application area are the use of general purpose operating systems on Internet terminals as well as palmtop computers. Such systems undertake activities such as execution downloaded content and would benefit from multitasking Unix-style operating systems. The biggest problem facing high performance of general purpose kernels on the ARM is the ARMs privileged mode/s of operation and memory management design. So while ARM provides a very nice user programming model with very efficient implementations, the kernel's view point (i.e., the privileged view) is poor and inhibits high performance kernel implementations. Also while providing some nice memory management concepts like domains, a few flaws in the design limits them (i.e., only tagging 1MB sections).

One novel/extremist research avenue is to redesign and implement (using FPGA technology or simulation) a new privileged mode of operation for the ARM and port Gauntlet to it, making a comparison of performance, etc. Suggested changes include using software loaded TLBs and folding all privileged modes into a single mode with a single set banked registers covering the upper half of the register set. Refining the Domains concept as a more flexible form of ASID is also worth while. These changes would be transparent to the user space as the instruction set would not require modification and would most likely lead to a simpler core design leading to lower power/cost implementations then current ARM cores which are potentially faster.

6.2 Conclusions

In this thesis, a step by step analysis of the challenges facing a port of the L4 microkernel to the ARM and StrongARM SA-1100 has been presented, as well as possible solutions to each challenge. The resulting design presents solutions to most of the issues raised and outlines the basis from which to approach the remaining ones.

The design has been partially implemented and critically evaluated by predicting the performance based on cycle counts. It has been shown the the potential performance looks very promising. However, the cost of intra task address space switches on the StrongARM still looms and may prove too high due to the particular caching architecture. While these costs may severely limited the practicality of L4 on the StrongARM, the resulting kernel design is still valuable for ports to other ARM processors and potentially to the next generation of StrongARM processors which will hopefully address the problems in the caching architecture.

Even though the potential performance is promising, only a 'real' system running on top of a complete and optimised implementation can truly asses the kernels performance and thus the success of the design.

The final conclusion of this thesis is that the performance of the critical IPC system call critically depends on the *fast address-space context switching* mechanisms presented in Chapter 4. Whether these mechanisms will live up to their promise and in what situations is yet to be seen. This performance with dictate the practicality of L4 on the StrongARM.

Bibliography

[AH98] Alan Au and Gernot Heiser. *L4 User Manual*. School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, January 1998. UNSW-CSE-TR-9801. Latest version available from http://www.cse.unsw.edu.au/~disy/L4/.

[ARM98] ARM Limited. *ARM720T Datasheet*, July 1998. Order no: DDI 0087E.

[ARM99] ARM Limited. *ARM920T Technical Reference Manual*, June 1999. Order no: DDI 0150A.

[Bar97] Michael Barabanov. A linux-based real-time operating system. Master's thesis, New Mexico Institute of Mining and Technology, June 1997.

[BBH+98] R. Baumgartl, M. Borriss, H. Härtig, Cl.-J. Hamann, M. Hohmuth, L. Reuther, S. Schönberg, and J. Wolter. Dresden realtime operating system. In *Proceedings of the Workshop of System-Designed Automation (SDA'98)*, March 1998.

[BH70] Per Brinch Hansen. The nucleus of a multiprogramming operating system. *Communications of the ACM*, 13:238–250, 1970.

[BKW94] Kavita Bala, M. Frans Kaashoek, and William E. Weihl. Software prefetching and caching for translation lookaside buffers. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation*, pages 243–253, Monterey, CA, USA, 1994. USENIX/ACM/IEEE.

[CB93] J. Bradley Chen and Brian N. Bershad. The impact of operating system structure on memory system performance. In *Proceedings of the 14th ACM Symposium on OS Principles*, pages 120–133, Asheville, NC, USA, December 1993.

[CLFL94] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and protection in a single-address-space operating system. *ACM Transactions on Computer Systems*, 12:271–307, November 1994.

[Com99] Compaq Computer Corp. *Alpha 21264 Microprocessor Hardware Reference Manual*, 1999.

[Dig92] Digital Equipment Corp., Maynard, MA, USA. *Alpha Architecture Handbook*, 1992.

[EHL97] Kevin Elphinstone, Gernot Heiser, and Jochen Liedtke. *L4 Reference Manual — MIPS R4x00*. School of Computer Science and Engineering, University of NSW, Sydney 2052, Australia, December 1997. UNSW-CSE-TR-9709. Latest version available from http://www.cse.unsw.edu.au/~disy/L4/.

[Fur96] Steve Furber. *The ARM System Architecture*. Addision Wesley, 1996.

[Hen84] John L. Hennessy. VLSI processor architecture. *IEEE Transactions on Computers*, 33:1221–1246, 1984.

[HEV+98] Gernot Heiser, Kevin Elphinstone, Jerry Vochteloo, Stephen Russell, and Jochen Liedtke. The Mungi single-address-space operating system. *Software: Practice and Experience*, 28(9):901–928, July 1998.

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

[HHW98] Hermann Härtig, Michael Hohmuth, and Jean Wolter. Taming linux. In *Proceedings of the PART '98*, 1998.

[Hoh99] Michael Hohmuth. The fiasco kernel: System architecture. TU Dresden Technical Report, in review, 1999.

[Int98] Intel Corp. *StrongARM SA-110 Microprocessor Instruction Timing*, September 1998. Order no: 278194-001.

[Jag95] Dave Jagger, editor. *Advanced RISC Machines Architecture Reference Manual*. Prentice Hall, July 1995.

[LES+97] Jochen Liedtke, Kevin Elphinstone, Sebastian Schönberg, Herrman Härtig, Gernot Heiser, Nayeem Islam, and Trent Jaeger. Achieved IPC performance (still the foundation for efficiency). In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS)*, pages 28–31, Cape Cod, MA, USA, May 1997. IEEE.

[LHH97] Jochen Liedtke, Hermann Härtig, and Michael Hohmuth. Os-controlled cache predictability for real-time systems. In *Proceedings of the Third IEEE Real-time Technology and Applications Symposium (RTAS'97)*, 1997.

[Lie93] Jochen Liedtke. Improving IPC by kernel design. In *Proceedings of the 14th ACM Symposium on OS Principles*, pages 175–88, Asheville, NC, USA, December 1993.

[Lie95a] Jochen Liedtke. Improved address-space switching on Pentium processors by transparently multiplexing user address spaces. Technical Report 933, GMD SET-RS, Schloß Birlinghoven, 53754 Sankt Augustin, Germany, November 1995.

[Lie95b] Jochen Liedtke. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on OS Principles*, pages 237–250, Copper Mountain, CO, USA, December 1995.

[Lie96a] Jochen Liedtke. *L4 Reference Manual — 486/Pentium/PentiumPro, Version 2.0*. GMD, Schloß Birlighofen, Germany, September 1996. Working Paper 1021.

[Lie96b] Jochen Liedtke. μ-Kernels must and can be small. In *iwooos96*, pages 152–161, Seattle, WA, USA, October 1996. IEEE.

[Lie96c] Jochen Liedtke. A short note on cheap fine-grained time measurement. *Operating Systems Review*, 30(2):92–94, April 1996.

[Lie96d] Jochen Liedtke. Towards real microkernels. *Communications of the ACM*, 39(9):70–77, September 1996.

[SAT98] Intel Corp. *SA-1100 Microprocessor Technical Reference Manual*, September 1998. Order no: 278088-001.

[Sch96] S. Schönberg. The L4 microkernel on Alpha - design and implementation. Technical Report 407, Cambridge University, 1996.

[Sko99] Espen Skoglund. Private communication. Karlsruhe University, 1999.

[WH00] Adam Wiggins and Gernot Heiser. Fast address-space switching on the StrongARM SA-1100 processor. In *Proceedings of the 4th Australasian Computer Architecture Conference*, 2000. (To appear).

[Wig98a] Adam Wiggins. The L4 micro-kernel (L4μK) on the StrongARM SA-1100. Working paper. URL http://www.cse.unsw.edu.au/-~disy/papers/Wiggins_prop98.ps.gz, October 1998.

[Wig98b] Adam Wiggins. P.L.E.B. URL http://www.cse.unsw.edu.au/~pleb/docs/pleb-proposal.ps.gz, June 1998.