THE UNIVERSITY OF NEW SOUTH WALES

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

Optimising L4 on Blackfin 533/537:

an investigation into a high performance L4 microkernel without virtual memory

Clarence Dang (3101378)

Bachelor of Software Engineering Program

Submission Date: 2006-10-31

Supervisor: Kevin Elphinstone
Assessor: Gernot Heiser

Abstract

The L4 microkernel is used as the basis for several operating systems but was built on the assumption of virtual memory. This thesis examines general design issues for constructing a high performance port of L4 without virtual memory but with memory protection. It also aims to provide a concrete implementation by porting L4 to the Blackfin processor.

The results of our research were found to be general, and not just Blackfin-specific. Therefore, we enable L4 to support an additional category of processors – those without virtual memory.

Our L4 implementation on Blackfin verifies the validity of the design. While it outperforms ucLinux, at least on context switching time, there is still much work to be done before it is deployable.

Acknowledgements

Kevin Elphinstone supervised this thesis and the thesis prior to this one. Stefan Petters and Gernot Heiser also supported me during the difficult task of changing to this thesis.

Gernot Heiser gave me the summer project of which this project is based upon. He sparked my interest in kernel hacking.

Alex Webster, Ben Leslie, Matthew Warton and Carl van Schaik, helped answer my torrent of questions about L4 kernel implementation.

Aiden Williams lent me two STAMP boards and his expertise.

Table of Contents

Illustration Index

Index of Tables

Chapter 1

1 Introduction

The L4 microkernel [Lie96] is used as the basis for several operating systems but currently only supports processors with virtual memory.

The main advantages of virtual memory is that it provides application isolation and demand paging. However, the use of virtual caches, for performance reasons, creates extra complexity such as synonyms and homonyms.

Virtual memory is a two-edged sword – without virtual memory, these complexities are gone but others take their place. L4 was built on the assumption of virtual memory so the suitability of its primitives must be re-evaluated. We therefore examine the issues in supporting L4 on architectures with memory protection units (*MPU*).

The Blackfin [Ana05] is a popular embedded processor and a specific instance of an MPU architecture so we examine the problems in its context.

The remainder of this chapter defines the goals of this thesis and then presents the structure of the rest of this report.

1.1 Goals

This thesis will identify the microkernel construction issues that arise out of supporting L4 on MPU-based processors, specifically Blackfin. The findings will be generalised to other architectures without virtual memory, such as the ARM1156T2-S MPU-based processor, and even to architectures without memory protection, such as the ARM7TDMI.

We shall construct a port of L4 to Blackfin whose performance will compare favourably to ucLinux, whose minimum context switch time is approximately 2,485 cycles [Hen06]. We will even attempt to match the speed of other L4 ports to RISC architectures, whose context switches complete in the low hundreds of cycles.

In doing so, we will identify a number of Blackfin-specific optimisations as well as general techniques for constructing high performance kernels.

Finally, our work will mean that Blackfin will become the 4th mature, public port of NICTA L4-embedded after ARM, x86 and MIPS64. This will strengthen the applicability and competitiveness of L4 in a wider part of the embedded market.

1.2 Report Structure

This section describes the structure of the remainder of the thesis.

Our thesis begins by providing the necessary background for our work:

- Chapter 2 analyses the features and performance of the Blackfin processor. It also compares Blackfin to the variants of ARM without virtual memory, to provide a more general feel for the thesis.
- Chapter 3 discusses the benefits of a microkernel approach to operating system construction, the L4 microkernel and recent, relevant L4 developments.
- Chapter 4 reflects on a "quick and dirty" implementation of the kernel on Blackfin that ignored design. The subsequent chapters in the thesis are primarily devoted to addressing the design and performance issues identified here.

We then provide 5 chapters on design:

- Chapter 5 identifies the assumptions in the design of our kernel.
- Chapter 6 discusses the effects of a lack of virtual memory.
- Chapter 7 analyses the implications of *protected kernel addressing*, a feature of some processors.
- Chapter 8 considers ways of reducing the amount of memory used by UTCBs, a particular kernel data structure.
- Chapter 9 explores choices in the system call ABI.

Our thesis then implements and critiques some related work:

- Chapter 10 describes the implementation of Warton's single stack kernel [War05].
- Chapter 11 investigates the implementation of Nourai's physically addressed thread control blocks [Nou05].

We then shift our focus largely onto general implementation issues:

- Chapter 12 describes a small number of issues regarding IPC. Other IPC performance bottlenecks are also dealt during the coverage of more general issues in Chapters 7, 13 and 14.
- Chapter 13 presents micro-optimisations that dramatically improve kernel performance.
- Chapter 14 runs experiments that aim to improve the kernel's performance, through changing the way it manipulates the Blackfin's protection unit and considers address space switching in this context.

Finally we reflect on, and evaluate, the work we have done:

- Chapter 15 describes some peripheral observations we made the implementation of this thesis
- Chapter 16 presents problems with the L4 microkernel, that we met along the way.
- Chapter 17 evaluates the entire thesis.
- Chapter 18 reflects on what goals have been achieved and what remains to be done.

Chapter 2

2 The Blackfin Architecture

The chapter describes the characteristics and performance of the Blackfin architecture, providing a qualitative and quantitative foundation on which we will be able to make informed decisions regarding kernel design trade-offs. We perform measurement experiments and analyse aspects of the architecture. Relevant parts of the Blackfin reference manuals [Ana05, Ana05c, Ana05b] are also summarised to provide the minimal background required to understand this thesis. We begin by providing some light Blackfin and architecture background before moving on to describe the structure of the rest of this chapter.

The Blackfin is an embedded DSP processor from Analog Devices [Ana05]. It is a 32-bit RISC processor whose striking feature is that it has no virtual memory but it does have a memory protection unit. We shall concentrate on the Blackfin 533 and 537 due to gcc toolchain support [Bla05] and because they have identical cores. ucLinux [Bla06d] runs on this processor and is a key competitor to L4-based operating systems.

During the last decade, CPU frequency has slipped away from being the primary determinant of system performance. Other aspects such as caching, TLB coverage and pipelining have become equally, if not more, important. As a result, these issues must be analysed in some depth.

We now describe the structure of the remaining parts of this chapter:

1. The section *Basic Operations* will measure the cost of key, basic Blackfin instructions.
2. *Interrupts, Exceptions and Traps* will describe aspects of entering and exiting the kernel.
3. It is followed by the *Memory* section with extended discussions on caching and protection mechanisms, as well as performance implications.

4. *Instruction Pipeline* will examine instruction pipelining on the Blackfin.
5. The *Possible Blackfin Security Bug* section deals with a potential exploit due to a instruction set design flaw.
6. Finally, we look at the similarities between Blackfin and other virtual-memory-less architectures in the *Comparison to ARM* section, in order to make our findings more general.

2.1 Basic Operations

So that proper architecture trade-offs could be determined, the cost of basic operations were measured:

Operation	*With D-cache & I-cache*		*Uncached*
	Cycles	*Initial cycles*	*Cycles*
NOP	1	7	13
Register copy	1	7	13
Register addition	1	7	13
Read word from RAM [1]	1	8	49
Write word to RAM	1	8	28
Read from Core MMR [2]	4	7	13
Write to Core MMR	4	7	13
Call to empty function [3]	18	21	417
CSYNC [4]	10	10	71
SSYNC [5]	16	23	86

Table 1: Speed of basic Blackfin operations

We firstly discuss how these results were measured and then provide an analysis of these costs. The speed of branches is considered in another section [p144, Chapter 13.4].

1 RAM = main memory (SDRAM) write-back cached by L1.

2 Core MMR = uncached Memory Mapped Register storing CPU state. The innocuous DTEST_DATA0 (0xffe0 0400) was used for both tests. It is believed that reads and writes to this register do not have side-effects that could cause extra computation. Other registers (e.g. for changing memory protection) were expected to cause actual computation.

3 The function was:

```
00 e8 00 00     LINK 0x0;  // Create stack frame
01 e8 00 00     UNLINK;    // Tear down stack frame
10 00           RTS;       // Return to caller
```

4 CSYNC = pipeline flush and CPU buffer synchronisation with L1.

5 SSYNC = CSYNC followed by synchronisation of L1 with the rest of the system.

2.1.1 How they were measured

The statistics were measured by executing the operations in question 256 times:

```
// Flush pipeline & CPU buffers, sync L1, sync with SDRAM
ssync;
// Guarantee pipeline flush (pipeline is only 10 long)
nop; [256 times]

.align 32:  // Align on cache boundaries

r0 = cycles;  // Read 64-bit cycle counter
r1 = cycles2;

// All instructions are at least 2 bytes.
// Therefore, at least 256 * 2 / 32 = 16 I-cache lines will be
allocated.

<the operation in question>; [256 times]

// Unfortunately, this will cause another I-cache line to be
// allocated.  This is not a big problem as the above operation
// causes 16x more allocations and the total cycle count will be
// divided by 256 anyway.
r4 = cycles;  // Read 64-bit cycle counter
r5 = cycles2;

nop; [256 times]  // Ensure no pipeline speculation warping results
```

50 runs of this were executed.

For the values for caching off, *Cycles* is the average of the cycles counts of all 50 runs, divided by 256. The standard deviation of the 50 runs divided by 256 (the number of times the operation was repeated) was 0, rounded to the nearest integer, in all cases.

For the values with caching on, *Initial Cycles* represents cycle count of the first run, with a cold D-cache and I-cache, divided by 256. *Cycles* is the average of the cycle counts of the last 49 runs divided by 256 on a warm D-cache and I-cache. The standard deviation of the last 49 runs was always 0, rounded to the nearest integer.

2.1.2 Analysis of costs

In this section, we discuss the measured instruction costs.

Under normal circumstances, the processor is capable of executing 1 basic computation per cycle. Note that access to L1 – or equivalently, cached main memory – is just as fast as register access!

However, memory mapped registers (MMRs) require 4 cycles to access even though they are in L1 memory, albeit in a different section. Firstly, this may be because interrogating and manipulating CPU state cannot come completely for free or that testing. Secondly, by running

the accesses back-to-back, we have introduced pipeline dependencies. In further testing, where we placed 16 NOPs (enough to flush the pipeline) between MMR writes, we found that the cost of MMR writes drops to 3 cycles, on average. However, given that we do tend to place MMR writes very close to each other, we quote the figure with pipeline dependencies – 4 cycles – in the table.

The function call exposes the cost of flushing the pipeline on both the call and return unconditional jumps. The cost of `CSYNC` pipeline flush is 10 which is surprisingly fairly cheap and equal to the length of the (short) pipeline. However, the values for both `CSYNC` and `SSYNC` are expected to be larger in real code due to extra required buffer synchronisation. The initial `CSYNC` cycle cost is the same as the cycle cost on a warm I-cache – this is an anomaly with the data that should be investigated.

The initial L4/Blackfin port was uncached so for completeness, the uncached values have been provided. The CPU store buffer can be seen to absorb the cost of 256 writes to the same RAM address but there is no similar "load buffer" for the more expensive reads. The uncached performance is dominated by instruction loads from SDRAM as can be seen from 2 benchmarks:

1. Call to empty function: The repeated function calls cause constant pipeline flushes, on both the jump into the functions and the returns, resulting in a high rate of instruction refetch.

2. `CSYNC`/`SSYNC`: We see the cost of refilling the pipeline, from SDRAM, for each of the 256 `CSYNC`/`SYNC` instructions (normally, 64-bits of instructions can be read in at a time but every instruction is a pipeline flush).

In summary, we ran experiments to measure the speed of basic operations on the Blackfin. The Blackfin, with no cache misses, can execute a basic instruction or memory access in a single cycle. Memory Mapped Registers can be accessed in 4 cycles. Function calls are very expensive, at 18 cycles. Finally, flushing the pipeline takes at least 10 cycles.

2.2 Interrupts, Exceptions and Traps

In this section, we describe the program sequencing in the context of entering and exiting the kernel. There are 16 interrupt/exception levels. The important ones are:

1. General-purpose, asynchronous hardware interrupts 7-15: These are generated by hardware or synthesised by the kernel mode `RAISE` instruction.

2. Synchronous exceptions "posing" as interrupt level 3: These are caused by events such as accesses to unmapped memory and illegal instructions. System call exceptions can be invoked by using the user and kernel mode `EXCPT` instruction.

Low priority interrupts (high numbered ones) can be preempted by equal or higher priority ones (low numbered).

However, exceptions are not allowed to occur when servicing an exception and cause the processor to event a *double fault* state. To bypass this problem, one simply needs to switch the processor to an interrupt mode on the initial exception, as exceptions are permitted during interrupts.

Interrupts are automatically disabled on kernel entry. Interrupts in kernel mode can be re-enabled by exploiting a bizarre instruction side effect – pushing the register containing the address of the user instruction (`RETI`) onto the kernel stack. So if one wishes to save `RETI` without re-enabling interrupts in kernel mode, one must copy it to another register first:

```
R0 = RETI;
[--SP] = R0;
```

Interrupts can be masked using the `IMASK` memory-mapped register. If userspace is, for some reason, given access permissions to the `IMASK` register, it can also mask interrupts. Regardless of processor mode, exceptions cannot be masked.

Trapping can be done without reserving userspace registers for the kernel (like MIPS32's `k0` and `k1`). This is because there are separate userspace and kernel space stack and instruction pointers.

A readonly memory-mapped `IPEND` register contains bits set for each interrupt level currently being serviced in order to keep track of nested interrupts/exceptions. Usermode is defined as the state of the CPU when it is not serving any interrupts. But since `IPEND` cannot be directly written to, it is difficult to return to usermode when serving nested interrupts. This is in contrast to the writeable *CPU status register*, found on other chips, which normally contains a single bit stating whether the chip is currently in kernel mode.

In summary, the Blackfin supports nested interrupts and different interrupt priorities but it is difficult to return from usermode in the presence of nesting. Exceptions cannot be nested so if we wish to allow exceptions while serving a first exception, we must switch to an interrupt mode on the initial exception.

2.3 Memory

We will now analyse the all the memory aspects of the Blackfin – layout, caching and protection.

It is important to emphasise that the Blackfin is a pure physical memory system. Virtual cache issues [Uns05] that plague virtual memory architectures, such as cache synonyms (multiple virtual addresses per physical addresses) and homonyms (a virtual address referring to different physical addresses depending on the current address space) are therefore non existent.

2.3.1 Memory layout

The Blackfin has a flat 4GB 32-bit address space and supports a maximum of 128MB of conventional, SDRAM [Bla06, Ana05b]:

0 - up to 128MB	conventional memory
128MB - 0xffc00000	memory mapped registers (*MMRs*) for devices(there are many reserved holes in this region)
0xffc00000 - 0xffe00000	CPU's system MMRs
0xffe00000 - 4GB	CPU's core MMRs

Table 2: Blackfin memory layout

The system MMRs access "on-chip peripherals outside of the core" [Ana05 p1-6] and run at the system clock frequency (99MHz on the STAMP board). Conventional memory is therefore capped as this speed but this overhead is absorbed by cache.

The core MMRs run at the CPU clock frequency (497MHz on the STAMP board).

2.3.2 Memory caching

In this section, we provide a general, architecture-neutral motivation for caching and then discuss it in terms of the Blackfin.

Motivation for caching

The speed of main memory has not improved as quickly as the speed of CPUs. As a result, the CPU may stall waiting for memory to be copied to and from registers, when it could be serving literally hundreds of instructions. One way to reduce the impact of this is to have a small but very fast amount of memory *cache* near the CPU. The hope is that because of locality, most accesses can be served directly by the cache (a *cache hit*) instead of needing to access slow main memory (a *cache miss*). As a result, cache hits have a dramatic beneficial effect on system performance. To amortise memory access costs, on a cache miss, the cache copies to

and from memory the size of a cache line ("typically 16-32 bytes" [Uns05]).

An *associative* cache is split into a number of *sets* of cache lines. The bits in a memory address are viewed as and split into the following (the number of bits for each varies from cache to cache) [Uns05]:

1. Tag (most significant bits)
2. Set number
3. Byte within the cache line's data (least significant bits)

The *set number* determines which cache set a particular memory address is in. The hardware performs a parallel (also known as associative) lookup of all lines in the set and determines whether the memory address is in the cache by matching the *tag* bits of the address. A *way* consists of all of the same numbered lines from each set i.e. it cuts across all sets.

Higher associativity – more cache lines per set – results in complex and expensive hardware due to the parallel lookup. However, this leads to fewer cache conflicts because different memory addresses with the same set number have a number of lines to choose from. One extreme is a *fully associative* cache, where there is only one set. The other extreme is *direct mapped* where each set contains only one line.

From a programmer's point of view, in order to minimise the number of cache misses:

1. Minimise the size of code and data so that our working set is more likely to fit into the high-speed cache.
2. Do not unnecessarily cross an alignment boundary. For instance, if the cache lines are 32 bytes each and an 8 byte variable is stored at the address `n * 32 + 28` (for arbitrary `n`), accessing this variable would result in 2 cache misses – a line for `n * 32` and the line for `(n + 1) * 32` – as the variable is split between 2 cache lines.

In summary, caching uses the principle of locality to hide the latency of memory. Given the size of the memory-CPU speed gap, caching is essential for system performance. However, this only works if we optimise our code and data for size so that the working set fits into cache.

Blackfin's caching

We now consider different aspects of the Blackfin's caching features, building on top of the caching background we just provided. The topics that will be covered include L1 memory, data and instruction caches, store buffer, cache coherency and the cost of cache misses.

L1 memory

L1 memory runs "at the full processor speed with little or no latency" [Ana05 p1-4] so is approximately 5 times (`497 / 99`) faster than the main SDRAM memory.

The Blackfin 53x series of chips contain [Bla06, Ana05c]:

- **16 KB of instruction SRAM/Cache**
- **16 – 32 KB of data SRAM/Cache (32 KB for the 533)**
- 16 – 64 KB of instruction SRAM (64 KB for the 533)
- 0 – 32 KB of data SRAM (32 KB for the 533)
- 4 KB of *scratchpad* data SRAM

SRAM/Cache means that part or all of the respective memory block can be configured as cache – we have highlighted these in **bold**. A cache is bypassed if it is disabled or if a newly fetched line is not deemed "important" enough (see our *Instruction Cache* discussion below) to replace an existing line [Ana05 p6-16].

All other SRAM can potentially be used for locking in kernel code and data without polluting the caches. Instruction SRAM can only be filled using DMA ([Ana05c p307]) which means code that is to be placed there must be specially linked to the right address or be relocatable.

The Blackfin 53x's have no L2 memory [Bla06].

Instruction cache

This is a 16KB 4-way associative cache with each line containing 32 bytes of data. Tags and line attributes are stored separately from each line.

The processor keeps track of 2-bits per line for an LRU replacement policy. The operating system can state an LRU "importance" bit in each cache line's tag so that "unimportant" lines cannot evict "important" ones.

On an instruction cache miss, the CPU begins reading 32 bytes (the size of a cache line) from the line fill buffer, which consists of four 64-bit entries [Ana05 p6-15]. The line fill buffer will read from external memory. The CPU restarts as soon as the line buffer returns a word as "the line fill buffer allows the core to access the data from the new cache line ... [before] the line has been written".

It is possible to flush and modify lines of the cache.

Data cache

The data cache is 32KB 2-way associative cache with each line containing 32 bytes of data. Again, tags and line attributes are stored separately from each line.

Data cache lines can be configured to either "write-through" or "write-back" and the line replacement policy is determined by the Cache Controller [Ana05 p6-29].

A line fill buffer similar to the instruction cache's one exists. An `SSYNC` instruction is required to flush the write buffer [Ana05 p6-35].

It is possible to prefetch, flush and invalidate lines.

Store buffer

A `6 * 32-bit` store buffer sits between L1 and the CPU core [Ana05 p6-28]. The CPU can read two 32-bit words simultaneously from the store buffer and or write 32-bits.

Cache coherency

Modifying cached instructions requires invalidating the respective cache lines to guarantee cache coherence [Ana05 p6-16].

Similarly, the data cache lines must be invalidated before a DMA operation [Ana05 p17-2].

Cache misses

Consistent values for I-cache and D-cache misses and refills could not be obtained. The value appears ranges from 47-226 but seems to hover around 113-134 cycles. This section describes some our experiments in attempting to measure cache miss costs and attempts to justify the measurement variability.

D-Cache refill cost

We attempted to measure D-cache costs by flushing the D-cache and then using this code (different to the previous test [p17, Chapter 2.12.1]):

```
// Flush pipeline & CPU buffers, sync L1, sync with SDRAM
ssync;
// Guarantee pipeline flush (pipeline is only 10 long)
nop; [256 times]

.align 32:  // Align on cache boundaries

r0 = cycles;  // Read 64-bit cycle counter
r1 = cycles2;

p1.h = <arbitrary_cplb_mapped_address>;
p1.l = <arbitrary_cplb_mapped_address>;
r6 = [p1];

r4 = cycles;  // Read 64-bit cycle counter
r5 = cycles2;

nop; [256 times]  // Ensure no pipeline speculation warping results
```

The code was run with and without the statement that caused a D-cache line pull `r6 = [p1]` (D-cache line pull) and we concluded that the cycle difference of 66 was the cost of a D-cache miss. Adding the `r6 = [p1]` could not have caused an extra I-cache line pull as the total instruction size between the cycle counter reads inclusive is 14 bytes including it.

I-Cache refill cost

Similarly, I-cache misses were measured at approximately 113.5 per line by adding and removing the italicised `nop; [256 times]` below. This was based on the claim that there would be an addition or removal of `256 * sizeof(NOP) / 32 = 16` lines (ignoring the comparatively insignificant cost of executing the NOPs themselves):

```
// Flush pipeline & CPU buffers, sync L1, sync with SDRAM
ssync;
// Guarantee pipeline flush (pipeline is only 10 long)
nop; [256 times]

.align 32:  // Align on cache boundaries

r0 = cycles;  // Read 64-bit cycle counter
r1 = cycles2;

// Each NOP instruction is 2 bytes
nop; [256 times]

r4 = cycles;  // Read 64-bit cycle counter
r5 = cycles2;

nop; [256 times]  // Ensure no pipeline speculation warping results
```

However, we found that of values of 66 cycles for a D-cache miss and 113.5 cycles for an I-cache miss were not consistent, as described in the following section.

Inconsistent results

At other times, D-cache has been measured to be slower than we initial measured (e.g. 134 cycles instead of 66) and I-cache faster (e.g. 60 cycles instead of 113.5). For instance, we ran 3 tests, with different chip configurations, that flushed the D-cache and touched the first 32-bit word of every 32 aligned bytes of 32KB (the size of the D-cache). It resulted in the following cycle counts and suggest that D-cache misses take about 114 cycles (not our previously measured 66):

Chip Configuration	*Cycles per D-cache Line (32 bytes)*
1. D-cache and I-cache enabled	114
2. I-cache only	48
3. I-cache only baseline (our D-cache line touches were replaced with NOPs)	9

Table 3: Blackfin D-cache miss cycles

This involved 1024 repeated instances of:

```
r2 = [p0];  // Changed to NOP for the 3rd case in the table
r4 = p0;
r5 = 32;
r4 = r4 + r5;
```

```
p0 = r4;
```

The results in the table above are given for the second run so that the I-cache is warm. I-cache conflicts for this code (`1024 copies * 5 instructions = 5KB`) are assumed to be small. The cost of executing these instructions is assumed to be much smaller than the D-cache misses.

Because the 1st test executed the worst case for caching – causing 32 bytes to be loaded for every 4 bytes read – this 2nd test case performs expectedly better when D-cache is *off*.

To double check that the bulk of the costs we are measuring in the 1st test can be attributed to D-cache refills, we performed a 3rd test which replaced the memory access with a NOP. The 3rd test executed in far fewer cycles (9) so we can conclude that the cost of the 1st test is largely due to D-cache lines being faulted in, in approximately 114 cycles. But this is almost double the figure of 66 we arrived at previously.

Possible reasons for inconsistent cache miss measurements

The inconsistent values that we are getting for D-cache and I-cache line fills are worrying. Evidently, there is something happening inside the CPU that we were not expecting

One possibility is the line fill buffers (between L1 & SDRAM) enabling CPU access to the data before the entire cache line is filled. Because the refill protocol stipulates that the pipeline restarts as soon as the word, to be accessed, is loaded (and the rest of the line is refilled in the background), the matter is made more complicated. There may be following instructions, accessing the same cache line, proceeding in the pipeline while the cache line is being filled. Depending on the offsets of the accesses within the cache line and timing, these following accesses could potentially succeed without stalling the pipeline if the fill buffer is sufficiently filled.

A "back of the envelope" calculation suggests that for D-cache, the cost of copying a line from SDRAM into the *fill buffer* is:

`497 / 99 * 32 / 4` = approximately `40` CPU cycles

where:

`497` = CPU frequency
`99` = system frequency i.e. SDRAM upper bound
`32` = cache line size
`4` = word size (assuming word size bus transfers [Ana05c]).

An extra `32 / 4 = 8` cycles would then be used to copy from the fill buffer to L1. The total of `40 + 8 = 48` cycles is fairly close to the 60 odd cycles sometimes seen for I-cache and D-cache misses (also given the variability and impact of SDRAM refreshes).

In any case, we can conclude that I-cache and D-cache misses cost at least 60 cycles and sometimes take more than 100.

Memory caching conclusions

System performance is strongly linked to cache footprint. The Blackfin's split L1 instruction and data caches can be accessed as almost quickly as registers, amortised, and assuming no cache misses. We ran experiments but could not determine precise costs for D-cache and I-cache misses but we do know that they are in the order of 100 cycles each.

2.3.3 Memory protection

In this section, we will firstly discuss the Translation Lookaside Buffer used for memory protection. Then we will analyse the Blackfin's version of this mechanism.

Translation Lookaside Buffer

Most CPUs contain a *Memory Management Unit (MMU)* which provides virtual to physical translation and checks memory accesses by looking up permissions from a Page Table stored in main memory. Unfortunately, if the CPU were to access the page table (memory) on every memory access, this would double the number of memory transactions resulting in suboptimal performance. Therefore, CPUs contain a small (e.g. 16 entries) but very fast (fully associative) cache of page table entries [Uns05], called a *Translation Lookaside Buffer (TLB)*.

MMUs are far more common than the *Memory Protection Unit (MPU)* used on architectures without virtual memory. MPUs contain all the protection functionality of MMUs without the translation.

If the memory address of the access to be checked is not in the TLB, a *TLB miss* occurs and the TLB must be filled from the entry in the Page Table corresponding to the address. This is performed by either hardware or the kernel depending on the architecture.

If it is not in the page table, a *page fault* exception is sent to the kernel. The kernel must either insert an entry into the Page Table or if the faulting thread does not have the appropriate permissions to access the faulting address, killing the offending thread.

We will now consider TLB replacement policies and other operating system considerations.

TLB replacement policies

As with cache lines, in order for a TLB refill to occur, an existing line must be evicted to make room for the new one, unless there are free lines. Ideally, the line that will not be used for the longest should be removed. Three policies are First In First Out (FIFO), Random and Least Recently Used (LRU).

FIFO is fast and simple to implement but fails to exploit locality as occasional accesses to other addresses evicts commonly used pages. Random is almost as fast and simple but also avoids the pathological worst case of cyclical access to a working set slightly larger than the TLB coverage. Random is the most popular method for TLB replacement.

LRU assumes prior behaviour is suggestive of future behaviour i.e. that line used least recently will not be used for the longest period of time. Without an oracle, LRU is as close as practical for optimal exploitation of locality. Unfortunately, tracking which line has been used most recently would require updating state on every memory access resulting in unacceptable performance.

An approximation of LRU is the *2nd chance* or *Clock* eviction policy. Clock maintains a

pointer to a "current" line. Whenever a line needs to be evicted, the clock increments its pointer (wrapping around to the first line), evicts the first line with the *reference bit* clear and finishes. For all lines it encounters along the way, the Clock clears the reference bit. Accesses to lines with zero reference bits results in a fault (TLB readonly exception). The kernel will respond by setting the reference bit of the faulting line. In this way, the reference bit simulates a "least-recently-used" bit. Lines which are in use are less likely to be evicted.

However, each run of the Clock algorithm may mark many pages as readonly. The subsequent protection violations, on writes to these many pages, usually outweighs the benefit of avoiding the refill cost of the accidental eviction of a single page. Simple and fast policies such as Random are much better.

Other operating system considerations

The size of memory has been growing rapidly but the TLB, and the memory coverage of the TLB, has not [Uns05]. Therefore, TLB pressure is increasing. In order to reduce the overhead of TLB misses, an operating system should minimise the number of TLB misses by ensuring that each entry covers the largest address range possible (as long as each address in the range shares the same protection attributes). Therefore, the TLB may support a number of page sizes. "Large" page sizes such as 1MB and 4MB are often referred to as *superpages*.

Tagged TLBs contain Address Space IDs (*ASIDs*) in order to avoid leaking protection between address spaces. Untagged TLBs without ASIDs normally require that the operating system flush the TLB on each address space switch. This performance penalty is compounded by the extra cost of serving the TLB misses that occur immediately after the switch.

Blackfin

As Blackfin does not support virtual memory, it has MPUs instead of TLBs. We now describe the details of its MPUs.

The *Data Cacheability Protection Lookaside Buffer (DCPLB)* is a 16 entry, untagged and fully associative MPU for data accesses. Each entry controls caching attributes and enforces user read and write access permissions.

The Instruction *Cacheability Protection Lookaside Buffer (ICPLB)* is a similar MPU but for controlling instruction fetches. It operates totally independently of the DCPLB. It is therefore possible to execute instructions (ICPLB) but not be able to read them (DCPLB).

Both are refilled in software by the kernel and therefore, suffer from pipeline and cache pollution. This means that system performance depends more heavily on reducing the number of MPU entry refills, compared to a hardware-refilled MPU.

1KB, 4KB, 1MB and 4MB pages are supported. There are bits for locking a line and marking a page as kernel accessible only. Pages cannot overlap so they can only be shared between userspace and kernel space at the same fine or coarse granularity.

The MPUs are optional and are off by default. If you enable them for memory protection, even kernel accesses must go through the MPUs. MPU pressure is therefore increased and we

discuss further implications of this *protected kernel addressing* later [p86, Chapter 7].

The MPU must be disabled before changing any MPU entries. This requires an `SSYNC` to flush the pipeline and propagate the protection disable notification. After changing some MPU entries, one must re-enable protection and `SSYNC` again. These 2 `SSYNC`s alone mean that protection changes require at least `2 * 10 = 20` cycles [p17, Chapter 2.1].

Protection can be turned on without caching. However, caching requires protection. Therefore turning off protection to bypass the MPU for accessing memory in kernel mode should not be done for long periods of time due to the performance hit of no caching.

Summary of memory protection

The TLB is a hardware cache of the page table, providing virtual to physical translation and protection. TLB misses are either served by the kernel or hardware depending on the architecture. A simple and effective TLB replacement policy is Random. The operating system should attempt to use large superpages for greater TLB coverage and to reduce TLB pressure. Untagged TLBs cannot allow for entries from different protection domains so mandate expensive flushes on address space switches.

The Blackfin has software-refilled, split data and instruction protection units that offer no translation – the DCPLB and ICPLB. Kernel addressing is also subject to the CPLB protection mechanisms. Changing CPLB entries is expensive, as is leaving them off as they control caching as well.

2.3.4 Conclusions about memory and Blackfin

The Blackfin does not have virtual memory so has a single 32-bit address space, with many CPU memory-mapped control registers at the top. Accessing cached data in the split instruction and data caches is just as fast as register accesses. The Blackfin has software-refilled and separate data and instruction CPLB protection units.

2.4 Instruction Pipeline

In this section we discuss the features of CPU instruction pipelines. We then delve into the details of the Blackfin's pipeline.

2.4.1 Pipelining

Modern RISC processors split instruction execution into multiple stages. Therefore, multiple instructions can be "in flight" at different stages of execution, maximising the usage of different CPU units, as long as the instructions are independent.

The pipeline is stalled in the presence of data dependencies i.e. "when an instruction depends on the results of a previous instruction" [CNC00] but that previous instruction has not yet written its output.

Branch hazards can occur because the processor speculatively executes instructions based on a *prediction* of whether a branch will be taken. If this prediction is wrong, the pipeline must be flushed of such instructions and refilled – expensive with a long pipeline. Speculative reads or writes from I/O devices may cause unexpected behaviour.

2.4.2 Blackfin

The processor contains a 10 stage pipeline which stalls automatically in several places so that software does not have to deal with hazards in order to guarantee correctness [Ana05]. Assuming no stalls, the Blackfin can perform at least 1 instruction per cycle [p17, Chapter 2.1].

Write operations write to the store buffer [p24, Chapter 2.3.2] and complete before "the data is actually written to an external memory or I/O location". Reads may be satisfied from the store buffer.

Read instructions, placed after a branch instruction, may be speculatively executed before the branch instruction is committed. This is unexpected for an I/O devices and so the `CSYNC` instruction must be used before I/O reads to flush the store buffer and to ensure "any pending interrupts, speculative states (such as branch predictions), or exceptions" are completed. The `SSYNC` instruction performs a `CSYNC` and then "flushes any write buffers between the L1 memory and the system domain and generates a sync request to the system that requires acknowledgement".

Interrupts can cause I/O accesses to be aborted, possibly after a "memory-read cycle was initiated". In this case the access will occur again after the interrupt, which off-chip devices do not guard against. Therefore, interrupts should be disabled before such accesses.

Static branch prediction is used for conditional branches and we discuss the performance of this, on the Blackfin, in another chapter [p144, Chapter 13.4].

2.4.3 Conclusions on instruction pipelining

Modern chips execute multiple instructions simultaneously at different stages of a potentially long *pipeline*. Branches decrease performance because they prevent effective instruction readahead into the pipeline.

On the Blackfin, the pipeline is only 10 stages long and can normally execute 1 instruction per cycle. Speculative execution and work store ordering mean that I/O device accesses must be surrounded by `CSYNC`/`SSYNC` barriers and depending on the situation, require interrupts to be disabled.

2.5 Possible Blackfin Security Bug

We strongly believe that the Blackfin contains a security hole so this section describes an exploit in detail and also workarounds. The hardware feature that enables this exploit, *zero overhead loops*, will be explained followed by a presentation of the code and an analysis.

2.5.1 Zero overhead loops

The Blackfin enables this exploit through its support for zero overhead loops [Ana05]. Normally, one points an `LT` (Loop Top) register to the start instruction address of a loop. `LB` (Loop Bottom) will point to the last address in the loop. `LC` (Loop Count) will be set to the number of iterations of a loop. These registers can only setup loops a short distance away from the setup code.

However, if user loop setup code is placed sufficiently close to the kernel trap or untrap code and executed, a zero overhead loop may be setup in the *kernel* code. The kernel would then loop when trapping in (or out) which would corrupt the kernel stack. Worse still, possible stack overflows (or underflows) would either corrupt other kernel memory and/or cause a in-kernel faults due to unexpected accesses to unmapped memory.

On the following page, we present the code for the exploit that sent to Analog Devices on 2006-03-02. This will be followed by a subsequent analysis of the attack.

2.5.2 Exploit code

Userspace

```
// Point the loop registers to kernel code.
// For whatever reason, the user is aware of the position of the
// kernel untrap code ("untrap_middle" & "untrap-end").
r0.h = untrap_middle; r0.l = untrap_middle; lt0 = r0;
r0.h = untrap_end; r0.l = untrap_end; lb0 = r0;
r0.h = 0x7fff; r0.l = 0xffff; lc0 = r0;  // >2*10^9 loop iterations.

// Loop forever waiting for an interrupt to fire.
l:
        jump.l l;
```

Kernel space

```
untrap_from_interrupt:
        usp = [sp++];
        fp = [sp++];

        [... restoring of more registers ...]

        // Loop registers get restore here, forming a loop!
        lt0 = [sp++];
        lb0 = [sp++];
        lc0 = [sp++];

// This label is here just to make it clear as to what userspace is
// pointing to.
untrap_middle:
        b3 = [sp++];
        b2 = [sp++];
        b1 = [sp++];
        b0 = [sp++];
        l3 = [sp++];
        l2 = [sp++];

        [... restoring of more registers ...]

        a0.w = [sp++];

// This label is here just to make it clear as to what userspace is
// pointing to.
untrap_end:
        a0.x = [sp++];

        // Return to usermode.
        rti
```

2.5.3 Analysis of exploit code

We firstly describe the sequence of events, arising from the previous exploit code, followed by a discussion on how to defeat the attack.

Sequence of events

1. Userspace sets up malicious loop registers that point to kernel code.

2. Userspace spins in a loop waiting for an interrupt to fire.

3. An interrupt fires.

4. The kernel saves all registers on the trap in.

5. The kernel serves the interrupt.

6. The kernel attempts to exit back into usermode via `untrap_from_interrupt`.

7. The kernel begins restoring registers.

8. As part of 7. it restores the loop registers `lt0`, `lb0`, `lc0` (just before `untrap_middle`). But the user pointed these registers to kernel code (specifically to loop between `untrap_middle` and `untrap_end` and for more than 2 billion iterations). Therefore a hardware loop is now active in *kernel mode*.

9. The kernel continues restoring registers between `untrap_middle` and `untrap_end`.

10. But before it reaches the `rti` instruction, it decides to loop back to `untrap_middle` due to the hardware loop. This is repeat 2 billion times until the kernel stack is underflowed.

Can we defeat this exploit?

While this could be defeated by masking the `LB0` and `LB1` registers, which are stored on stack, to ensure that they only point to userspace, a similar exploit that forms a hardware loop before the kernel *trap* is entered, at first, seems certain to be triggered by helpless kernel entry code. But this too can be defeated by making use of a feature of hardware loops – they can only cause code at most 30 bytes away to loop. Therefore, ensuring that there is enough padding between user code and the kernel trap code is sufficient to ensure the exploit will fail.

Nevertheless, we had significant difficulty in getting either exploit to work. Analog Devices only provided canned responses.

2.5.4 Blackfin bug conclusions

The zero overhead hardware loop feature of the Blackfin opens up the possibility of a malicious

userspace process creating loops in the kernel. However, we are experiencing difficulty verifying whether it is a genuine exploit and in any case, it can be worked around.

2.6 Comparison to ARM

Recall that our focus is on supporting L4 efficiently on architectures without virtual memory, by drawing conclusions based on an implementation on the Blackfin architecture [p12, Chapter 1.1]. So while our focus will be on Blackfin, most of the results will generalise to other architectures without virtual memory.

In this section, we introduce two such architectures – ARM1156T2-S [Arm05, Wig99] and ARM7TDMI [Arm04] – and compare them with the Blackfin to provide a broader context for our research. We find that ARM1156T2-S is a very similar MPU-based chip to the Blackfin and that ARM7TDMI is similar to ARM1156T2-S but has no memory protection at all.

Throughout this document, all our conclusions should generalise to these chips except for the areas where we point out Blackfin-specific problems or features.

2.6.1 ARM1156T2-S

This is a 32-bit RISC architecture with 37 registers, with 16 visible at any one time. There are different privileged modes for different types of exceptions and some registers are banked (e.g. The stack pointer and the link register) between usermode and kernel mode. The ARM features DSP instructions and a separate *Thumb-2* 16/32-bit instruction set for improving code density. The pipeline has 9 stages and features global branch prediction. This is all fairly similar to the Blackfin except that Blackfin has fewer protected modes and only has static branch prediction.

Both the ARM and Blackfin have trouble dealing with nested exceptions. Recall that Blackfin disallows exceptions within exceptions, so exception handlers must defer their work into interrupt handlers (which support a nested exception) [p16, Chapter 2.2]. With ARM, in the event that an exception occurs during the handling of the same type of exception, the link register is trashed. Therefore, the kernel must ensure that this situation never occurs – or switch modes just as Blackfin kernels are forced to.

Like the Blackfin, the ARM has separate instruction and data caches, which are both usually 4-way associative (like the Blackfin instruction cache). The line length is also identical to Blackfin with eight 32-bit words.

The ARM also has separate instruction and data untagged MPUs, each containing 16 entries. It also features protected kernel addressing, in which kernel accesses are constrained by the MPUs [p86, Chapter 7]. These aspects are identical to the Blackfin.

However, there are significant differences between the 2 chips. The ARM supports page sizes of every power of 2 from 32 *bytes* to 4GB while the Blackfin only supports 1KB, 4KB, 1MB and 4MB. MPU entries are permitted to refer to pages that overlap, unlike the Blackfin. Furthermore the ARM has hardware-loaded MPUs (based on a 2 level page table), while the Blackfin suffers from slower software MPU refills.

Recall that turning off the Blackfin MPU completely disables caching [p30, Chapter 2.3.3]. ARM is similar except that it reverts to a memory layout where the bottom 1.5GB or 2GB of the address space – according to a runtime configuration option – is cacheable.

The ARM supports changing of protection entries without disabling the MPU, unlike the Blackfin. However, disabling or enabling the MPU, for whatever motivation, requires flushing both caches. In contrast, disabling or enabling the MPU on the Blackfin only requires `SSYNC` pipeline flushes.

2.6.2 ARM7TDMI

The ARM7TDMI is a simpler ARM chip with no virtual memory but its main difference is that it does not have memory protection. It used in Apple's extremely popular iPod music player so supporting L4 on this architecture would create the potential for massive, global L4 microkernel use. L4 would also provide an alternative to iPodLinux – a port of ucLinux to the iPod [Ipo06].

We aim to provide a unified L4 API for all non-virtual-memory architectures regardless of whether the architecture has an MPU, to reduce the porting effort for applications built on top of the microkernel. Ideally, to migrate applications to an MPU-less architecture, all we would need to do is simply need to disable the protection mechanisms.

2.6.3 ARM comparison conclusion

Our goal is to support L4 on architectures without virtual memory. While our focus is on the Blackfin, we realise that another popular MPU-based architecture – ARM1156T2-S – is extremely similar so our results will generalise. The important differences are that ARM's MPU is hardware-loaded, supports a much broader set of pages sizes and that MPU entries can refer to overlapping pages. We can also trivially support architectures without even MPUs, such as the ARM7TDMI (and also very similar), by simply turning off the protection.

2.7 Blackfin Architecture Summary

The following characteristics of the Blackfin were discussed:

- No virtual memory
- 32-bit architecture and 32-bit address space
- Exceptions cannot be nested so if we wish to allow exceptions while serving a first exception, we must switch to an interrupt mode on the initial exception
- Software-refilled Data Caching Protection Lookaside Buffer and Instruction Caching Protection Lookaside Buffer as MPUs.

We ran experiments to measure the speed of basic operations and cache misses. The important results we found were:

- Primitive instructions can be executed in 1 cycle
- Cached word accesses are just as fast as register accesses and also complete in just 1 cycle
- Cache misses cost in the order of 100 cycles so a key to good kernel performance is to minimise cache footprint

A potential kernel exploit utilising Blackfin's zero overhead hardware loop feature was described and workarounds suggested.

We also compared the Blackfin to other architectures without virtual memory so that our work can generalise:

1. ARM1156T2-S is a very similar chip, also with split instruction and data MPUs. However, its MPU is hardware-loaded and supports overlapping pages.
2. ARM7TDMI has an instruction set similar to ARM1156T2-S but does not have an MPU. We hope to support such MPU-less architectures using the same L4 API as for MPU architectures but with the protection functionality ignored.

Chapter 3

3 L4 Background

Traditional *monolithic* operating systems, such as Linux or Windows, have millions of lines of operating system code running in the privileged kernel mode. A single bug – security or otherwise – in the kernel is likely to compromise the entire system.

This implies that the only way to develop reliable systems is to restructure operating systems to minimise the amount of privileged code. The centrepiece of this radically different structure is the L4 microkernel [Lie96] foundation, on which operating systems can be built to execute safely and efficiently in userspace.

In this chapter, we analyse *The Microkernel Approach* and motivate the need for operating systems to be built this way for reliability.

We then describe the various *Versions of L4*.

Finally, *Recent Developments* will cover relatively new, related work that will be incorporated into our thesis – namely, the N2 API, Single Stack Kernel and Physical TCB Arrays.

3.1 The Microkernel Approach

In an L4-based system, only the small microkernel (tens of thousands of lines of code) runs in kernel mode. A well-known claim is that a small "trusted computing base" [Nic06a] is much more likely to be correctly implemented and lends itself to formal verification. Traditional operating systems services are instead run as servers in usermode and should they fail, they can be restarted.

In this section, we introduce basic L4 abstractions and then provide further motivation for microkernel-based operating systems.

3.1.1 Abstractions

The main L4 abstractions are address spaces, threads and Inter-Process Communication messages (*IPC*) [Nic05a].

Address Spaces

Each address space can be viewed as a traditional "process" whose memory is isolated from other address spaces. Each address spaces contains 1 or more threads and maintains, in kernel space, a page table.

Address spaces provide for bug isolation as memory corruption from unrelated processes cannot occur.

Threads

L4 threads are the same as lightweight kernel threads in traditional operating systems.

Threads' IDs, kernel stacks, and other such information are stored in *Thread Control Blocks (TCBs)* in kernel space. Thread state shared between userspace and kernel space (e.g. message registers for IPC) is stored in *User Thread Control Blocks (UTCBs)*.

IPC

Threads synchronise and communicate with each other via IPC and potentially, shared memory. This provides a strict, well-defined interface between different threads and processes in the system – in contrast to the ad-hoc structuring of monolithic systems such as Linux, where device drivers manipulate page table structures apparently directly [RC01 ch. 13].

As the primary communication mechanism in a microkernel-based operating system, IPC must be very fast even across address spaces. Ideally, this should be completed in tens of cycles but on most architectures, this can only be implemented in the order of hundreds of cycles or at

worst, the low thousands. A distinguishing characteristic between L4 and previous generation microkernels, such as Mach, is in IPC performance [Lie96].

3.1.2 The microkernel advantage

Process isolation with address spaces and well-defined interfaces using IPC means that each process can be individually engineered and mathematically verified. This is not possible in a monolithic system where a memory bug in say, a driver, could compromise an unrelated system such as the virtual memory manager (unless a protection system hack such as Nooks is used [SML+02]).

But microkernels come at a cost. Firstly, proper object-orientated operating system design is, in the short term, more difficult than ad-hoc implementation. However, in the long term, it tackles the problem of increasing data and control flow complexity in traditional operating systems. Secondly, well-defined interfaces that happen to depend on address space switches are slower than simply accessing a variable or calling a function. It remains to be seen whether a real-world system can be constructed using a pure microkernel architecture and exhibit comparable performance to a traditional operating system.

3.2 Versions of L4

In this section, we describe the foundation for our work, NICTA L4-embedded, and previous work on porting different versions of L4 to other architectures.

3.2.1 NICTA L4-embedded

NICTA L4-embedded [Nic05a] is NICTA's version of L4 with a focus on embedded systems. Based on the Pistachio tree, it is largely written in C++ so the code for any single architecture is between ten thousand to fifteen thousand lines of code. It is commercially relevant – more recently, Qualcomm has started developing chipsets based on L4-embedded [Nic05b]. Therefore, this thesis focuses on NICTA L4-embedded due to its active development and support within the ERTOS research group.

3.2.2 Ports

The L4-embedded [Nic06b] source tree contains working ports to other architectures – namely, ARM, IA32 and MIPS64. Their code served as reference implementations that guided the development of the Blackfin port. The MIPS64 port was referred to because it was straightforward and the IA32 port was used because it has the same word size and endian as the Blackfin.

Alpha [Pot99, Sch96], MIPS32 [Bla06e] and SPARC-V9 (by Philip Derrin but report is not published) ports of L4 – to name a few – are well-documented outside of the source code. However, as the discussions are very architecture-specific and given that they describe versions of the API that differ greatly from NICTA's [Nic05a], they are of limited use. Ports prior to Pistachio, of which L4-embedded is based upon, were written in assembler and are even less useful as references.

3.3 Recent Developments

The Blackfin port was based on an "N1.5" (between *N1* and *N2*) version of the NICTA L4-embedded kernel. Recent research and development within the ERTOS group has steered the port in additional directions. We consider *N2*, the single stack kernel and the physically-addressed TCB changes.

3.3.1 N2

We wrote an initial port of L4 to Blackfin to get a feel for the issues involved in microkernel construction [p47, Chapter 4]. However, in the meantime, the L4 *N-series* API [Nic05a] underwent continual refinement and progressed to the *N2* version. In order to pick up architectural changes, the Blackfin port was updated. This was so that performance analyses would still be valid for the current architectural state of the kernel rather than one long gone by.

This was time-consuming because there were many breaking changes on an implementation level but luckily, it was found that nothing significant – other than the addition of a cache management API – had actually been done.

3.3.2 Single Stack Kernel

When the initial port of L4 to the Blackfin was written, there was a kernel stack for every user thread, making context switches transparent to the majority of kernel code. However, reserving a stack for every thread in the system consumes a lot of memory and cache footprint. As a result, L4 is now moving to a continuation model with a single kernel stack [War05]. We will implement this and report our findings in this thesis [p113, Chapter 10].

3.3.3 Physical TCB Arrays

On virtual memory architectures, the TCB array is normally a large virtual array. Individual entries are only physically backed on demand.

This virtual allocation permits a large thread ID space, avoiding thread ID reuse. On top of version numbers, reducing thread ID reuse increases the difficulty for threads to masquerade as dead trusted threads that happened to have the same thread IDs.

Note the necessity for the L4 microkernel to provide a large thread ID space. It is tempting to move this responsibility up to the operating system, layered on top and in userspace. The idea would be that the OS would provide its own thread ID abstraction, that maps to L4 thread IDs, with the goal of providing a much larger thread ID space. However, this is flawed because it could be bypassed by malicious processes by using L4 system calls (which only accept L4 thread IDs).

Nourai [Nou05] describes different ways of storing TCB arrays in physical memory in the

interests of simplicity and performance. For virtual memory systems, he cites the following advantages:

1. Physically addressing the TCB array reduces the use of virtual address space which is scarce on 32-bit systems.

2. TLB misses no longer occur on TCB accesses. The TLB refill cost is eliminated and TLB pressure is reduced. Linear kernel execution is achieved and this simplifies formal verification.

   Furthermore, some architectures clobber registers on nested exceptions, such as in-kernel TLB misses, requiring workarounds [p37, Chapter 2.6.1]. If we eliminate such nested exceptions, we simplify the kernel as we do not have to consider this register trashing possibility. With additional, usable registers, the fastpath can be more efficient.

Blackfin, ARM1156T2-S (MPU) and ARM7TDMI (no memory protection) [p37, Chapter 2.6] do not have virtual memory so for us, Physical TCB Arrays are not a matter of performance but rather something we *must* implement. Our actual implementation is later described in detail [p125, Chapter 11].

3.4 Summary

The L4 microkernel can be used to build potentially more reliable operating systems, compared to traditional systems, as only a small amount of code is privileged.

L4-based operating systems will consist of servers in separate address spaces, providing for bug isolation. Threads communicate primarily using IPC so this must be fast.

We are basing our work on NICTA L4-embedded and will implement the recent Single Kernel Stack and Physical TCB Array developments.

Chapter 4

4 An Initial Blackfin Port

Having developed the necessary Blackfin and L4 background in the previous 2 chapters, we wrote an "quick and dirty" implementation to gain a feel for the issues in writing a high performance kernel, without virtual memory, and to provide a base for experimentation. Specifically, it implemented a full pre-*N2* API to the Blackfin 533 Revision 0.3 architecture on an *ADDS-BF533 STAMP REV 1.2* board [Bla06c].

A naive implementation, and as the first ever port of L4 to a physical-only memory architecture, it performed badly as it was written quickly and ignored design issues. This chapter analyses its shortcomings and introduces design and implementation changes to improve its performance.

As our kernel was mature enough to run the *pingpong* microbenchmark, which will be described shortly, we start by providing performance figures. We then analyse different aspects of the kernel – the kernel entry and exit path, system calls, IPC, all aspects related to memory, thread ID space and UTCBs.

4.1 Pingpong Microbenchmark

No serious – not toy and contrived – macrobenchmarks were available so we could take advantage of Linux-based benchmarks as ucLinux had not yet been ported to L4/Blackfin. Instead, we made a straightforward port of the *pingpong* microbenchmark to the N2 API. We also fixed a bug that resulted in the reported cost of non-IPC tests being half of what they should have been.

Pingpong measures operations by executing them 10,000 times (outer loop of 2, inner loops of 5,000).

The *NOP* test measures the baseline loop overhead of the benchmark.

Kernel entry and exit measures the speed of entering and exiting the kernel, without any shortcuts. This is the baseline overhead for all kernel operations. A *System call* is the speed of essentially the null system call. These are our special additions to pingpong and the difference between *Kernel entry and exit* and *System call* will be discussed shortly [p55, Chapter 4.3].

The IPC tests repeatedly bounce a message between 2 threads. This has a very small working set – and therefore small MPU and cache footprint – so represents best case performance. There is a real danger that, in the absence of macrobenchmarks, drawing conclusions based on microbenchmarks will result in suboptimal design decisions for real workloads. *Intra-AS IPC* is the cost of an IPC from one thread to another in the same address space. *Inter-AS IPC* is across address spaces.

On the next page, we present the performance figures for our initial kernel, with cache enabled and disabled.

Operation	*With D-cache & I-cache*		*Uncached*	
	Cycles	*us*	*Cycles*	*us*
NOP	9.2	0.2	141.1	0.2
Kernel entry and exit	4,049.2	9.4	68,407.6	137.6
System call [*]	7,775.3	15.6	128,987.8	259.6
Intra-AS IPC				
0 MRs	17,566.3	35.0	271,083.8	545.0
4 MRs	17,218.7	35.0	272,283.3	548.0
8 MRs	17,272.2	35.0	273,160.2	549.0
60 MRs	17,890.7	36.0	285,650.0	575.0
Inter-AS IPC				
0 MRs	39,203.1	79.0	633,899.7	1,275.0
4 MRs	38,907.2	78.0	635,059.7	1,278.0
8 MRs	38,962.8	78.0	636,004.9	1,280.0
60 MRs	39,582.0	79.0	648,468.4	1,305.0

Table 4: Performance of the initial L4/Blackfin port

The above numbers are the cycle and microsecond costs divided by 10,000 (the number of iterations) and rounded to the nearest .1 (just so that the microsecond column didn't show 0 for NOP). The timer interrupt was enabled and fired every 2ms to simulate a more realistic working environment. This would only have perturbed the much larger uncached results.

The kernel was uncached by default as we feared that, with a lack of testing, it would be unstable. However, these fears were unfounded as, without virtual memory, the ordinarily troubling aspects of caching – virtual cache issues – were non-existent. Turning on both I-cache and D-cache consistently gave 1 or 2 orders of magnitude improvement in cycle costs. As this was virtually for free, the subsequent discussion will only analyse statistics with caching enabled.

We see that even with caching enabled, the performance of the kernel was atrocious. For instance, the IPC figures are approximately 2 orders of magnitude worse than other L4 ports. Therefore, the following sections analyse the initial port's kernel entry and exit path, system calls, IPC and memory issues.

[*] The system call was the Blackfin-specific `L4_KDB_SystemClock()` which merely looks up a 64-bit value from the scheduler class.

4.2 Kernel Entry and Exit

The initial port implemented the majority of the trap code in C++. Nevertheless, 4,049.2 cycles is an extremely high cost for merely entering and exiting the kernel and this cost is part of all kernel operations such as IPC and protection unit refills. Therefore, we had a strong motivation to instrument the kernel to find out where the cycles were going.

We begin by looking at the cache performance of the whole code path, excluding the saving and restoring of the trapframe (due to technical difficulties). We then analyse the trapframe overhead, followed by an analysis of the remaining code written in C++.

4.2.1 Cache performance of the whole path

We measured the warm cache performance of entering and exiting the kernel – the number of I-cache and D-cache misses and the number of cycles lost to them:

Run	***I-cache***		***D-cache***	
	Misses	***Cycles***	***Misses***	***Cycles***
1	27	544	-	-
2	-	-	74	175

Table 5: Cache performance of the initial L4/Blackfin port

Due to a limitation in our instrumentation, these figures exclude the cost of saving and restoring the trapframe. Two separate runs were required to switch the 2 performance monitor registers from measuring I-Cache to D-Cache.

The performance monitoring unit is known to give impossibly high I-cache and D-cache cycle counts – in some tests, even higher than the total number of cycles for the entire operation! So the values in the table are probably higher than they should be and should be interpreted with caution. On top of this, the indirect cache effects of our instrumentation code have not been measured. Nevertheless, we do believe these figures can provide a very rough insight into the kernel port's cache behaviour.

These figures suggest that I-cache misses cost only `544 / 27 = 20.1` cycles on average and D-cache misses take an incredibly small `175 / 74 = 2.4` cycles. Note that this is a far cry from our measured hundred cycle cache refill costs [p24, Chapter 2.3.2] so again, we should be cautious.

The cache misses were only analysed briefly. The D-cache cost is insignificant as it is only `175 / 4049.2 = 4.3%` of the total cycle cost. The I-cache miss rate is higher at `544 / 4049.2 = 13.4%` of the total cycle cost and warrants further investigation.

As these runs were on warm caches, there could not be compulsory misses. Capacity misses were also not possible because, in this test, userspace was a very small loop that touched almost no data and the kernel would not be able to blow 16KB of instruction cache, nor 32KB of data cache.

So we can only conclude that we are witnessing conflict misses. Although the instruction cache is more associative than the data cache (4-way compared to 2-way) [p22, Chapter 2.3.2], the larger number of cycles lost to I-cache misses can be attributed to a kernel entry and exit path that executes code far more frequently than accessing memory. Conflict misses can be reduced by minimising the amount of sparsely distributed kernel code and data by firstly reducing the size and memory access footprint of the kernel and secondly, by packing together code and data into aligned cache lines.

4.2.2 Trapframe cost

As our previous analysis excluded the cost of saving and restoring the trapframe, we investigate it here. The trap into the kernel saves 45 words representing the entire user state:

```
[--sp] = (r7:0, p5:0);
[--sp] = a0.x; [--sp] = a0.w;
[--sp] = a1.x; [--sp] = a1.w;
[--sp] = i0; [--sp] = i1; [--sp] = i2; [--sp] = i3;
[--sp] = m0; [--sp] = m1; [--sp] = m2; [--sp] = m3;
[--sp] = l0; [--sp] = l1; [--sp] = l2; [--sp] = l3;
[--sp] = b0; [--sp] = b1; [--sp] = b2; [--sp] = b3;
[--sp] = lc0; [--sp] = lc1;
[--sp] = lb0; [--sp] = lb1;
[--sp] = lt0; [--sp] = lt1;
[--sp] = astat;
[--sp] = rets;
[--sp] = fp;
[--sp] = usp;
[--sp] = retx;  // or "[--sp] = reti;" if we're handling an interrupt
```

It was measured to take 45 cycles in the absence of cache misses, which is optimal for the Blackfin as it executes a maximum of 1 instruction per cycle, in almost all situations [p17, Chapter 2.1]. So we are not doing anything wrong here.

The cache footprint is as follows:

I-Cache: `31` instructions * 2 bytes for each instruction (even push multiple) / `32` bytes per cache line = `2` cache lines

D-cache: `45` registers * `4` bytes for each on the stack / `32` bytes per cache line = `6` cache lines

The I-Cache footprint is optimal as we could not possibly express these register saves in a smaller amount of code. The D-cache footprint is optimal as we could not possible pack the registers into a smaller space.

If we want to be pessimistic and assume that these `2 + 6 = 8` cache lines will always miss, then we can expect a penalty of more than 800 cycles [p24, Chapter 2.3.2]. However, unless the system is under an extreme working set, one would expect most of these cache lines to be in the cache. Nevertheless, placing the trap code, and storing user context, outside of ordinary memory and in L1 SRAM [p22, Chapter 2.3.2] would minimise the kernel's cache footprint.

Trapping out of the kernel performs a similar set of operations to trapping in, which we just

measured, so the analysis should be identical for trapping out.

4.2.3 Performance of the C++

In this section, will illustrate the importance of micro-optimisations. An analysis of the cost of kernel entry and exit path revealed that most of the cost comes from the C++ called between the assembler trap and untrap code. As just a taste of the problems we describe the `l4bfin_interrupt_level()` call, inefficiencies with regards to returning to userspace, assertions and consistency checks. These issues motivate the need for micro-optimisations not just for the simple kernel entry and exit case but also for system calls, the DCPLB refill handler and the like.

l4bfin_interrupt_level()

This function determine whether we came from user mode or kernel mode and is frequently called. We came from kernel mode if the `BFIN_IPEND` register shows that two interrupts or exceptions are currently nested:

```
int
l4bfin_interrupt_level (bool *was_in_kernel_mode) {
      if (was_in_kernel_mode)
            *was_in_kernel_mode = false;

      int ilevel = -1;
      for (int i = 0; i < 16; i++) {
            // Global Interrupt Disable is not an interrupt.
            if (i == 4)
                  continue;

            if (*BFIN_IPEND & (1 << i)) {  // Memory Mapped Register
                  if (ilevel == -1) {
                        ilevel = i;
                        if (!was_in_kernel_mode)
                              break;
                  } else {
                        ASSERT (NORMAL, was_in_kernel_mode);
                        *was_in_kernel_mode = true;
                        break;
                  }
            }
      }

      return ilevel;
}
```

Unfortunately, it is hopelessly inefficient. The above loop usually executes 16 times and on *each* iteration, performs at least (except for bit 4 of `BFIN_IPEND`):

1. 3-4 conditional branches (1 test for the `for` loop and 2-3 `if` statements)
2. 1 unconditional branch (end of the loop)

3. 1 access to an uncached memory mapped register, `BFIN_IPEND`

This was measured to take approximately 400 cycles and is executed 2-3 times depending on the particular code path used to enter or exit the kernel (exception or interrupt). In other words, 800-1,200 cycles is being spent in this function, per kernel entry and exit.

However, if we were to set the kernel entry points to different addresses for each interrupt level, the trap code could easily write the current interrupt level (and whether we came from usermode or kernel mode) to a variable that could be looked up. We would not even need to loop 16 times nor use tricky bit manipulation optimisation tricks. To handle nested exceptions (e.g. CPLB misses in kernel mode), we would simply need to maintain the current and previous interrupt levels, in a similar spirit to the MIPS32's status register.

Returning to userspace inefficiencies

There was some very inefficient code that manipulated Data CPLB protection unit entries on the trap out:

```
// Either have current space XOR on IDLE thread.
ASSERT (NORMAL, !!get_current_space () ^
      !!(get_current_tcb () == get_idle_tcb ()));
// TODO: walking page table is really bad performance.
const bool user_can_access_mmrs = get_current_space () ?
      get_current_space ()->lookup_mapping ((addr_t) BFIN_MMR_START,
                        0/*no output*/, 0/*no output*/) :
                  false;
l4bfin_dcplb_mmr_entry_enable_user (user_can_access_mmrs);

l4bfin_dcplb_set_utcb_entry (*((addr_t *) UTCB_PTR_START));
```

This code cost approximately 1,400 cycles as:

1. It walks the page table (`lookup_mapping()`) to determine if the thread is permitted to access the memory mapped registers. We underestimated the importance of inlining the page table entries' accessors, which are frequently called by this and so result in massive branching.

   Furthermore, we could have avoided this page table walk entirely by simply kept track of this special permission as a single bit in the TCB.

2. The code sets 2 DCPLB protection unit entries, which involves a total of 4 pipeline flushes and synchronisations [p29, Chapter 2.3.3].

Furthermore, it turns out that if we change our memory protection regime, we can avoid executing all of the above code entirely [p60, Chapter 4.5.2].

Assertions and consistency checks

We assumed that assertions and sanity checks would only take a small number of cycles and that their cost would be insignificant. Here we analyse just one such check used to determine if the kernel has accidentally tricked a processor double fault:

```
void
l4bfin_if_double_fault_panic (const char *trip_point)
{
      u32_t seqstat;
      BFIN_SEQSTAT (seqstat);
      if (exception_cause (seqstat) == 0x25/*Unrecoverable event*/)
      {
            l4bfin_double_fault_panic (trip_point);
      }
}
```

This costs approximately 46 cycles and is executed approximately 8 times for a total of `8 * 46 = 368` cycles. There are many other assertions and checks that should not have been compiled in.

4.2.4 Conclusions about the kernel entry and exit

Our initial kernel took too many cycles (thousands) to enter and exit the kernel. A preliminary cache analysis, although perhaps skewed by a buggy performance unit, suggested that conflict misses are occurring.

Our code for saving and restoring the trapframe, at least, is optimal and should exhibit good performance except under high cache pressure. Therefore, most of the cycles in the kernel entry and exit path come from the C++ due to the simply inefficient implementation and the enabling of assertions. We believe other kernel operations, such as DCPLB refill, suffer from the same problems and can be fixed in the same way.

Our inefficient implementation can be improved using micro-optimisation techniques, which we touched on briefly, and later discuss in great detail [p137, Chapter 13]. Some code can also be thrown out, saving valuable cycles, if we were to correct the design, as we noted with the memory protection unit manipulation on the untrap path.

We later find that disabling assertions alone results in more than a doubling of performance [p138, Chapter 13.1]. The smaller code size would also reduce the number of conflict misses we believe we observed (the less code, the less chance of a conflict, not to mention fewer capacity misses).

What we have learnt is a lesson in optimisation. The kernel entry and exit paths may be invoked hundreds of times per second (for IPC) so even the cost of operations that take a millionth of a second (500 cycles at 500 MHz) add up.

4.3 System Calls

In this section, we discuss the mechanism for system calls, the calling convention and revisit a Blackfin problem with nested exceptions, that creates extra inefficiency.

4.3.1 System call mechanism

System calls were implemented as exception numbers 0 (KDB debugging system calls) and 1 (L4 system calls) and triggered by the Blackfin `EXCPT` instruction. Recall that system calls must be implemented as exceptions as the user cannot generate interrupts [p20, Chapter 2.2].

4.3.2 System call convention

Out of laziness, the system call convention was made to match the Blackfin gcc function calling convention [Bla05] i.e. the first 3 arguments are register backed and further arguments are placed on the user stack. However, placing arguments on the user stack results in extra page table manipulations inside the kernel, resulting in unnecessary complexity.

The C calling convention already preserves too many registers, namely 10 of them. But our system calls were even worse than this – because they shared the trap code with interrupts and MPU miss exceptions, they saved and restored the full set of registers instead of the just C calling convention.

4.3.3 Nested exceptions

Because in-kernel CPLB protection unit miss exceptions might occur (on UTCB memory-backed message registers) during a system call and system calls can only be implemented as exceptions, we must switch the processor to an interrupt mode on the initial system call exception [p20, Chapter 2.2]. The kernel achieves this by raising a reserved interrupt line, masking all other usermode interrupts and exiting the kernel. The masking is required to prevent an interrupt – timer or otherwise – triggering an unexpected kernel entry. After entering the kernel but before the CPU executes any user code, the processor realises that an interrupt is active and re-enters the kernel immediately. Hence, the system call exception has caused a mode switch and deferred its work into an interrupt handler. This trick is also used by ucLinux but this mode switch is a waste of cycles so we later investigate ways to avoid in-kernel CPLB misses, by moving UTCBs, so that it is not required [p91, Chapter 7.3].

To make matters worse, we restore the full trapframe when exiting from the initial system call exception and then save it again on the entry into the interrupt. But no user code has been executed – the user context is unchanged and so these steps are pointless.

As usermode CPLB misses are exceptions, just like system calls, they too must defer their work to an interrupt handler.

4.3.4 System call summary

System calls are implemented as exceptions. However, they save and restore too many registers – too often – and involve the complexity of the kernel accessing the user stack.

System calls and usermode CPLB exceptions must waste cycles deferring their work into an interrupt handler due to the possibility of in-kernel CPLB miss exceptions. We will investigate eliminating such in-kernel exceptions by considering different UTCB schemes.

4.4 IPC

We noted that the IPC performance is 2 orders of magnitude worse than other L4 ports [p48, Chapter 4.1]. Recall that IPC is a very important part of L4's performance [p41, Chapter 3.1.1] so it is critical that we analyse the cause of this.

Recall that in the pingpong microbenchmark, 2 threads, `ping_thread` and `pong_thread`, bounce a message using 2-phase `L4_Call()` IPCs. Here, we present the sequence of events in Intra-AS IPC and Inter-AS IPC, before and after the thread switch. Take special note of the number of times the kernel is entered (*italicised*). We then follow with analysis and conclusions.

4.4.1 Before the thread switch

1. **System call**: The thread begins send using the `L4_Ipc()` system call. The *kernel is entered* via an exception. The kernel defers the call to a lower-priority interrupt (to cater for potential DCPLB misses in kernel mode [p55, Chapter 4.3.3]). The kernel returns to usermode.
2. **Deferred system call**: Immediately, the *kernel is entered* on the interrupt and begins the send phase of the IPC:
   a) The *kernel is re-entered* in kernel mode on a DCPLB miss on the receiver's UTCB MRs. The miss is resolved. Note that no exception deferring is required as a kernel-mode DCPLB miss will never cause a nested DCPLB miss or other exception (unless there is a kernel bug).
   b) The send phase is completed. The thread blocks in the kernel for the receive phase. The kernel switches to the receiver.
   c) If it is an Inter-AS IPC, the DCPLB is flushed, for the address space switch.
   d) Usermode is returned to.

4.4.2 After the thread switch (for Inter-AS IPC only)

Because, Step 2c flushes the DCPLB for Inter-AS IPCs, we immediately incur usermode DCPLB misses. The following steps only apply to Inter-AS IPC and are skipped for Intra-AS IPC.

3. **Stack miss**: Usermode triggers a DCPLB miss on writing a variable to the stack. The *kernel is entered* and defers the exception to a lower-priority interrupt. The kernel returns to usermode.
4. **Deferred stack miss**: Immediately, the *kernel is entered* on the interrupt and resolves the DCPLB miss. The kernel returns to usermode.

5. **Data segment miss**: Same as 3. but due to a global variable access. As a result, the *kernel is entered*.

6. **Deferred data segment miss**: Same as 4. but due to 5's global variable access. As a result, the *kernel is entered.*

4.4.3 Analysis

After step 6., usermode then loops back to 1. regardless of whether it's the `ping_thread` or the `pong_thread` (as they both do they same thing but alternate from being the receiver to the sender). Steps 3. & 4. and 5. & 6. may be swapped around depending the stack locations of the 2 threads.

For each Inter-AS, we enter the kernel 7 times (see italicised text in above sequence of steps). The kernel entry and exit cost multiplied by 7 (`4,049.2 * 7 = 28,344.4`) is approximately 10,000 cycles short of the figure for Inter-AS IPC (39,203.1), according to our benchmarks [p48, Chapter 4.1]. We can safely assume this to be the overhead of the kernel code which refills the DCPLBs for the multiple misses – and other computation costs (e.g. IPC kernel code). After all, if a kernel entry and exit, which by itself achieves nothing, costs more than 4,000 cycles, a missing 10,000 cycles – lost to *actual* computation – should not be surprising.

Intra-AS IPC is the same as Inter-AS IPC except that it does not switch address spaces so does not flush the DCPLB (skipping steps 2b and 3-6). Therefore, for each Intra-AS IPC, we only enter the kernel 3 times (in steps 1, 2 and 2a). Multiplying the kernel entry and exit cost by 3 (`4,049.2 * 3 = 12,147.6`) reveals approximately 5,000 missing cycles from the figure for Intra-AS IPC (17,566.3). Again, this assumed to be computation and overhead on top of the kernel entry costs.

4.4.4 Ways to speed up IPC

Having identified the IPC performance is heavily dependent on kernel traps and DCPLB refills, we draw conclusions on how to make the IPC perform well.

For Inter-AS IPC, recall that as the Blackfin DCPLB protection unit is untagged, we need to flush the DCPLBs on address space switches [p29, Chapter 2.3.3]. As we can see from steps 3-6, this results in compulsory and expensive DCPLB misses. However, if we saved and restored address spaces' DCPLB entries on address space switches, we could avoid these 4 traps. We later investigate this possibility [p153, Chapter 14.1].

We could speed up both types of IPC in the following ways:

1. Optimise the kernel trap, as previously described [p54, Chapter 4.2.4]

2. Eliminate in-kernel DCPLB misses [p91, Chapter 7.3] and therefore, the need to defer system calls and usermode DCPLB handling into interrupts, halving the number of traps or otherwise, speed up the DCPLB refill handler

3. Use register-backed message registers [p132, Chapter 12.2]

4. Write an assembler fastpath [p187, Chapter 18.2.2]

4.5 Memory

In this section, we begin by looking at the initial port's instruction protection. We then follow with a discussion of the port's data protection and DCPLB unit usage. The page table is discussed and we analyse the port's memory layout. Finally, we combine all this to propose a new memory layout.

4.5.1 No instruction protection

Instruction protection was disabled as Blackfin seemed to have buggy support for this when caching was off.

This is a potential security hole although it is hard to imagine a meaningful attack with being able to execute another address spaces' code but not being able to read or write its code nor data. Despite this, Kevin Elphinstone suggests that a secret algorithm could be used without authorisation or its internal structure could be discovered by observing its inputs and outputs. However, for this to work, the secret code could only access registers and data in the attacking address space (not the address space where the code resides). Nevertheless, it would be better to cover this potential security hole.

Instruction protection was later implemented and found to work when caching was *enabled*.

4.5.2 Data protection

Out of convenience, 6 pages are locked into the 16-entry DCPLB. We discuss why this too many for good performance and then analyse which pages do not have to be locked.

TLB and MPU pressure is an increasing performance bottleneck, especially on slow software-refilled architectures like Blackfin. As general rule, the kernel should not make assumptions about what kind of data the user is *likely* to access. For instance, should the kernel lock in a particular page, a program that primarily accesses its own data but not that locked page, would not be able to take advantage of that MPU entry, increasing MPU pressure. On the other hand, unlocking that page will not substantial penalise tasks that do access that page frequently as it will simply be loaded into the CPLB via the ordinary refill mechanism, and maintained there if the replacement policy respects temporal locality. Therefore, an almost always optimal policy for the kernel is to reserve as few MPU entries as possible.

The following entries were locked into the DCPLB:

Index	Purpose	Page Size
0	page containing UTCB pointer	1KB
1	Kernel Interface Page	4KB
2	kernel image	4MB
3	kmem heap	4MB
14	0xffc00000 CPU MMRs (user accessibility set on untrap)	4MB
15	UTCB of current thread (set on untrap)	1KB

Table 6: Locked DCPLB entries in the initial L4/Blackfin port

Clearly, locking almost half of the 16 DCPLB entries is denying data-bound processes optimal performance. We now investigate which entries do and do not have to be locked.

0 : page containing the UTCB pointer

This entry is for a fixed page (so that userspace can access it easily) whose first word contains a pointer to the current thread's UTCB, which is locked in entry 15.

The UTCB pointer could be moved into the KIP to free up a DCPLB entry.

1: Kernel Interface Page

The Kernel Interface Page (KIP) contains information about the architecture and the kernel, such as supported page sizes, as well as system call wrappers.

There is no need to lock this page into the DCPLB. If a thread wants to access the KIP frequently, it will be mapped into the DCPLB using the ordinary DCPLB refill mechanism. Otherwise, an extra DCPLB entry will be free for data-bound processes.

2: Kernel image and 3: kmem heap

The kernel image contains all the kernel-mode code and global variables. The kmem heap contains all of the dynamically allocated data structures in L4 – TCBs, threads' kernel stacks inside those TCBs, address spaces and page tables.

Blackfin and ARM1156T2-S suffer from protected kernel addressing in that all kernel accesses go through the protection unit [p86, Chapter 7]. Therefore, at least one of these pages must be locked into the DCPLB so that our trap code has a place to save the context. We cannot leave this page unlocked and hope that the trap code can load this page into the DCPLB for the purposes of having a place to save the context. This is because it would need to perform the DCPLB fill without clobbering the context – impossible on Blackfin due to an insufficient number of banked registers.

Once instruction protection is enabled, the kernel image must definitely be locked into the ICPLB for the trap code, as well as the DCPLB.

Note that the kernel image only occupies 200KB at worst. A 1KB or 4KB superpage is too small so our remaining choices of a 1MB or 4MB means that a lot of space is left unused. It therefore makes perfect sense to put the kmem heap at the end, saving a DCPLB entry and memory.

14: Memory Mapped Registers

The CPU's Memory Mapped Registers (known as *MMRs*) are located in the top 4MB of the 4GB address space. These are largely protection, caching and event vector registers used by the kernel. There are also registers for controlling hardware devices, debugging and the performance unit, which should be accessible by user level.

Due to protected kernel addressing [p86, Chapter 7], kernel memory accesses are still subject to the DCPLB protection unit. We need the MMRs to configure the DCPLB protection unit so one would expect that the MMRs would need to be locked into the DCPLB, else we would lose the ability to reconfigure the DCPLB.

As a result, we locked the 4MB superpage into the DCPLB. Fine grained sharing of the MMRs with userspace was impossible given that the Blackfin protection unit does not allow for overlapping mappings (ARM1156T2-S does however). So sharing the MMR superpage with userspace was an all or nothing matter. As a consequence, device drivers had to have full access to all MMRs instead of a small section (e.g. 1KB) resulting in reduced reliability since some MMRs can be used to crash the whole system.

However, after some experimentation, we found that the MMRs do not act like ordinary memory where a DCPLB miss would occur if the relevant page was not in the DCPLB. The actual behaviour is alluded to by the CPU manual [Ana05] but not properly documented:

During Usermode

1. If the address is covered by a CPLB entry with the appropriate permissions, the access goes ahead. This is just like ordinary memory.
2. Otherwise, a protection violation occurs – *never* a DCPLB miss. This is unusual behaviour.

During Kernel Mode

1. If the address is covered by a CPLB entry with the appropriate permissions, the access goes ahead. This is just like ordinary memory.
2. If it is covered with inappropriate permissions, a protection violation occurs. Again, this is like ordinary memory.
3. In this last case, the address is not in the CPLB but a “default CPLB descriptor” [Ana05]

will allow the access. This is unusual behaviour but means that we do not need to lock the MMRs into the DCPLB.

Therefore, we do not need to lock in the MMR superpage for the kernel and with the overlapping entry problem gone as a result, userspace could be given MMR access with the granularity of the minimum page size (1KB).

15: Current UTCB

This entry is for the page of the current thread's UTCB.

There is no need to lock this in as the application may be computation bound instead of IPC-bound (and utilising memory-backed message registers). If the thread, on the other hand, is involved in many IPCs, it is likely that this page is already in the DCPLB.

Data protection conclusions

Too many pages are arbitrarily locked into the DCPLB for good performance. On architectures with protected kernel addressing kernel addressing, at least one kernel page needs to be locked for the trap. In our case – Blackfin – only the kernel image entry needs to be locked.

DCPLB entries can be saved by moving the UTCB pointer into the KIP and placing the kmem heap at the end of the kernel image.

Memory-mapped register pages require special handling by the DCPLB refill path.

4.5.3 CPLB replacement policy

Our replacement policy for the DCPLB was crazily inefficient and this would also have been the case for the ICPLB, had we enabled it. In this section, we describe our scheme and consider different design alternatives.

The policy probed the entire DCPLB for a free line. Recall that there are 16 lines in a DCPLB and that each DCPLB register read is 4 cycles [p17, Chapter 2.1], giving a total of `16 x 4 = 64` cycles in the worst case, excluding the actual tests and loop overhead. If it failed to find a free line, it returned the first unlocked line it encountered. It began scanning from the line after the last victim in an attempt to simulate FIFO.

Recall that FIFO fails to exploit locality – occasional accesses to different parts of memory evict commonly used lines – but this is far better than Clock LRU's overhead [p28, Chapter 2.3.3].

Psuedorandom allocation fixes the pathological FIFO worst case – cyclical access of a working set larger than the CPLB coverage. A fast algorithm for 32-bit processors is described by Carta [Car90] and a MIPS32 reference implementation can be found in the lottery scheduling paper [WW94].

However, an even simpler and slightly faster scheme is to use the lower bits of the instruction counter register (called `CYCLES`). Let us consider a general, hypothetical problem caused by this with a unified TLB (does not distinguish between data and code pages). Assume that the current instruction's code page is in the TLB. If the instruction were to access a data operand whose page is not in the TLB, this would generate a TLB miss. We must be careful that the cycle cost of our TLB refill path is not a multiple of the number of TLB entries or else our replacement strategy, based on the lower bits of `CYCLES`, would replace the instruction page with the data page. On the return to userspace, the instruction opcode would now cause a TLB miss and the reverse would now occur. We would then be stuck in a loop continually swapping the data TLB entry with the instruction TLB entry and vice-versa. However, as the Blackfin has split instruction and data CPLB protection units, this cannot occur.

To simplify debugging, we opted to simply optimise our initial FIFO scheme. Instead of an expensive probe of the CPLB for a free line, we simply did not care if we evicted a line that was in use, even when another was free. This is a fairly reasonable strategy given that if there are many CPLB misses occurring, the CPLB is already likely to be full. In the future, we could avoid this problem by maintaining a low overhead queue of free CPLB entries, implemented as a simple array.

Once the kernel is very stable, we could trivially switch to psuedorandom scheme using the lower bits of the instruction counter but at the risk of being unable to reproduce "non-deterministic" bugs. On an attempt to reproduce such a bug, the psuedorandomness would be affected, for example, by a change in the number of cycles that the user waits at a prompt or if a hardware interrupts fires with a slightly different period.

In summary, the initial replacement policy was a FIFO one that wasted too many cycles probing the CPLB. We should use a much faster psuedorandom scheme based on the lower bits of the instruction counter. However, to guarantee determinism in the early stages of developing and

debugging the kernel, we will simply micro-optimise our FIFO variant and remove the CPLB probe.

4.5.4 Page table

The Blackfin port uses the L4 Linear Page Table Walker to manipulate a two level page table.

The first level is indexed by the most significant 10 bits of the address. It can either be a 4MB superpage mapping or a pointer to the second level.

The second level is indexed by the next most significant 12 bits of the address. It is searched multiple times (similar in style to what is described in Philip Derrin's unpublished L4/SPARC report) to determine whether the mapping is a 1MB, 4KB or 1KB page (in that order) so it is effectively 4 levels deep.

Unfortunately, the Linear Page Table Walker is too general – it walks virtually any kind of page table. More specialised code for our particular two level page table structure would be more efficient, as it would avoid unnecessary tests.

4.5.5 Initial memory layout

In this section, we present and critique the initial memory layout. Note that some of the pages we list here are locked into the DCPLB as described in the previous section on locked DCPLB entries [p60, Chapter 4.5.2].

1. Null pointer catcher

0 to 1KB	unmapped

This is permanently left unmapped to ensure that user and kernel null pointer bugs are trapped by the MPU. 1KB covers far much than a single word at address 0 but we could not do better as it is the smallest supported page size. On the bright side, this allows us to catch more than just simple null pointer accesses – it catches array accesses where the base of the array is at null but the array subscript is not.

2. Kernel Area 0 – shared with userspace

1KB to 2KB [user readonly]	UTCB pointer
2KB to 3KB [user executable]	idle thread user code
4KB to 8KB [user executable and readonly]	Kernel Interface Page (KIP)
8KB to 8KB + 256 threads * 1KB = 264KB [current thread's UTCB is user readable and writeable]	global UTCB array

In our previous discussion regarding optimising DCPLB usage, the UTCB pointer was to move into the KIP [p60, Chapter 4.5.2].

The motivation for having idle thread user code is discussed in a later section [p76, Chapter 5.1.2]. As we will implement instruction protection [p60, Chapter 4.5.1], accesses to the idle thread user code will result in ICPLB misses. However, the KIP already contains system call wrappers that the user must use to invoke system calls. Therefore, we can reduce ICPLB pressure but moving the idle thread user code into the KIP.

The global UTCB array is later discussed at length in the context of eliminating DCPLB misses in kernel mode [p91, Chapter 7.3].

3. User Area 0 – small area for the user

264KB to 4MB	user defined

This is a userspace "memory hole" disjoint from the main region from 12MB to 128MB, described shortly. We cannot move the subsequent kernel superpages, each 4MB, to start at

264KB as superpages must be aligned on their size. As a result, this hole is known as *external fragmentation*.

Discontiguous memory is inconvenient so we would like to avoid it. One approach is to move Kernel Area 0 (described above) to after the kernel superpages (described below). The kernel superpages would then start on address 0.

But this leaves us with the issue of the null pointer catcher page. One solution is to not have it but then null pointer bugs will not be trapped. On architectures which support overlapping MPU entries, such as the ARM1156T2-S, the null pointer catcher page could be locked into the MPU, with no permissions, and overlap the kernel superpages. It must be locked into the MPU if we wish to detect kernel null pointer bugs consistently, as else null pointer accesses could be satisfied by the locked kernel superpage entry below. This permanently wastes a DCPLB entry, which we have been trying to avoid [p60, Chapter 4.5.2]. So if we either dislike this solution or our architecture does not support overlapping DCPLB entries – such as Blackfin – a we simply have to accept this discontiguous memory trade-off.

4. Kernel Area 1 – kernel image

4MB to 8MB	kernel image

5. Kernel Area 2 – kernel heap

8MB to 12MB	kmem heap

We discuss the future directions of Kernel Area 1 and Kernel Area 2 together, recalling that we previously decide to merge them into a single 4MB superpage [p60, Chapter 4.5.2].

The 4MB size for this superpage is rather arbitrary, however it should be an important consideration as kernel memory pages reserved by L4 cannot be used by the user. Unfortunately, we later find that there is no single "correct" amount of kernel memory to reserve [p172, Chapter 16.1].

But in a quest to support, say, more threads, we might want a kernel heap larger than 4MB – larger than the maximum size of a superpage – would require locking more precious DCPLB entries, slowing hungry userland processes with large working sets.

Recall that the page sizes supported by the DCPLB (and ICPLB) are 1KB, 4KB, 1MB and 4MB. To minimise the number of locked DCPLB entries for the kernel heap, one would like to use the largest page size. Unfortunately, one must align pages on their size resulting in greater external fragmentation with larger page sizes.

Also, the desired page size may be too big but the next smallest page size too small resulting in internal fragmentation e.g. 2.5MB can be covered using three 1MB pages, locking 3 DCPLB entries or by one 4MB page, locking 1 DCPLB entry but wasting 1.5MB of RAM. So there is a trade-off between saving DCPLB entries and saving memory. DCPLB entries are precious for even small working sets since there are only 16 of them and the common page size of 4KB constrains the DCPLB coverage to just `16 * 4KB = 64KB`. But memory is also scarce because Blackfin supports a maximum of only 128MB of RAM.

6. User Area 1 – general area for user

12MB to 0xffc00000	user defined

This is where userspace can put the bulk of its code and data.

7. CPU memory-mapped registers

0xffc00000 to 4GB	memory-mapped registers (MMRs)

We discussed memory-mapped registers in a previous section [p60, Chapter 4.5.2].

In summary, we presented the memory layout of the initial Blackfin port. We should move the idle thread user code into the KIP. We have discontiguous user memory due to our desire to catch user and kernel null pointer bugs. On architectures supporting overlapping MPU entries, we can avoid this contiguous region and still trap null pointer bugs but at the expense of an extra locked MPU entry. Due to a flaw in the design of L4, we cannot determine what the size of the kernel heap should be. Supporting other sizes may waste memory or consume extra DCPLB entries. Changes to the memory layout are therefore only motivated by the previous section on DCPLB usage [p60, Chapter 4.5.2].

4.5.6 New memory layout

We now present our new memory layout based on our previous analysis of locked DCPLB entries [p60, Chapter 4.5.2] and the initial memory layout [p66, Chapter 4.5.5]. This section serves to clarify what the new layout is so no new conclusions are drawn here.

1. Null pointer catcher

0 to 1KB	unmapped

This is unchanged from the initial memory layout.

2. Kernel Area 0 – shared with userspace

4KB to 8KB [user readonly and executable]	Kernel Interface Page (KIP) containing the UTCB pointer and idle thread user code
8KB to 8KB + 256 threads * 1KB = 264KB [current thread's UTCB is user readable and writeable]	global UTCB array

The UTCB pointer is now placed in the KIP, saving a DCPLB entry. The idle thread's code has also been placed in the KIP to save an ICPLB entry as the system call wrappers are already there. The KIP is no longer locked as we should not anticipate application behaviour.

The global UTCB array is unchanged.

3. User Area 0 – small area for the user

264KB to 4MB	user defined

This is unchanged from the initial memory layout.

4. Kernel Area 1 – kernel image

4MB – 8MB	kernel image and kmem heap

We combined the separate 4MB superpages for the kernel image and kmem heap, as the kernel image needs 200KB. This new superpage must be locked into both the DCPLB and ICPLB for the trap code and is the only page, in our new memory layout, that needs locking.

6. User Area 1 – general area for user

8MB – 4GB	user defined

With one fewer kernel superpage, the user area starts 4MB earlier, providing more memory.

The memory-mapped registers superpage is no longer considered in our memory map as the kernel does not actually need to deal with it specially – DCPLB locking or otherwise. Fine-grained MMR sharing with userspace is now possible.

In summary, the new memory layout saves memory over the initial layout and significantly reduces CPLB pressure. Only the kernel image needs to be locked and in both the DCPLB and ICPLB.

4.5.7 Conclusions about memory issues

Instruction protection was initially disabled after Blackfin was found to be buggy with caching off. It works with caching on.

Six pages were arbitrarily locked into the DCPLB causing great DCPLB pressure. Some of the pages could be combined and in the end, we proposed a new memory layout where only a single page, for the kernel image, needed to be locked for the trap code. This is the minimum for a protected kernel addressing architecture such as Blackfin.

We have discontiguous user memory due to our desire to catch user and kernel null pointer bugs. It is unclear what the size of the kernel heap should be.

The initial FIFO-based CPLB replacement policy wasted too many cycles probing the CPLB. We simply propose to micro-optimise by removing the probing. In the future, we hope to use a psuedorandom scheme instead of the FIFO one and also maintain a low overhead queue of free CPLB entries to prevent unnecessary evictions.

We are using a two level page table structure but we need to use a specialised walker – rather than L4's generic one – for better performance.

4.6 Thread ID Space

As we do not have virtual memory, our TCB array is physically addressed. However, we literally have it as a fixed array of pre-allocated TCBs. As they are preallocated, we could only stomach the thought of 256 pre-allocated TCBs hogging kernel memory, as many of them could be unused (`256 * 4KB`, the size of the TCB = `1MB`). This severely limitations the thread ID space, which decreases system security [p44, Chapter 3.3.3]. Even after the implementation of the single kernel stack, where TCBs are only 512 bytes, should we only use 1MB for the TCB array, our thread ID space would still be small (`1MB / 512B`, the new sizeof the TCB = `2,048` TCBs). We would like to support a 16-bit thread ID space as suggested by Nourai [Nou05] and using the methods he describes.

The number of threads is also constrained by a fixed size, global UTCB array stored in kernel space. The global UTCB array is later discussed at length in this context and that of eliminating DCPLB misses in kernel mode [p91, Chapter 7.3].

4.7 UTCBs

Our UTCB size of 1KB was only decided upon based on convenience. Excluding message registers, UTCBs only need to be 64 bytes. Blackfin architecture supports a maximum of 128MB of RAM so memory is scarce. We should therefore investigate reducing the size of UTCBs.

No message registers are register-backed – all are stored in the UTCB. This means that the IPC path always has to copy message registers to and from UTCBs, stored in memory, touching cache lines. Register-backed message registers permit the IPC path to merely context switch to the destination thread with no copying so this is a great motivation for register-backing.

4.8 Conclusion

We wrote an "quick and dirty" implementation to gain a feel for the issues in writing a high performance kernel, without virtual memory. We found that the performance of the kernel was extremely poor in all aspects – kernel entry and exit, system calls and memory management.

We identified that most of the cycles in the kernel entry and exit path came from the C++ and could be dramatically improved using micro-optimisations. These same techniques could also be applied to improve the performance of other parts of the kernel.

System calls preserve too much of the user context and involve the complexity of the kernel accessing the user stack.

The possibility of in-kernel CPLB misses means that almost all kernel traps require an expensive mode switch from exception mode to interrupt mode. We will investigate design changes to eliminate in-kernel CPLB misses by considering trade-offs in UTCB placement.

IPC performance is heavily dependent on kernel traps and DCPLB refills. For Inter-AS IPC, we will consider reducing DCPLB misses by changing the design of address space switches to not flush the CPLBs. IPC, in general, would also benefit from the above implementation issue of micro-optimising of the kernel trap and the above design issue of removing in-kernel CPLB misses.

Six pages were arbitrarily locked into the DCPLB causing great DCPLB pressure. We proposed a new memory layout where only a single page, for the kernel image, needed to be locked for the trap code.

We wish to increase our limited 8-bit thread ID space to 16-bit of for security and need to do two things to achieve this. Firstly, we must implement Nourai's physical TCB allocation schemes. Secondly, we must reconsider the fixed array of all UTCBs, stored in kernel space.

Our UTCBs are too large and should be made smaller. We need to register-back message registers.

In conclusion, this chapter has identified a number of implementation and design issues, which we have been summarised here. The rest of thesis aims to address these issues.

Chapter 5

5 Design Assumptions

This chapter identifies the primary assumptions in our design which, of course, ultimately affect the implementation. Firstly, we justify why we do not want preemption and its effects on the kernel idle loop. We then identify the assumptions created by the L4 N-series API.

5.1 No Kernel Preemption

Preemption reduces latency by allowing the kernel to be interrupted in an operation. Generally speaking, the L4 kernel implementation is only preemptible in the page table code as mapping or unmapping a large number of pages, such as during an address space teardown, is very time consuming. In this section, we describe why we should not enable it and the subsequent consequences on the kernel idle loop.

5.1.1 Costs

In this section we describe the general and Blackfin-specific costs of enabling preemption and conclude that we do not want it.

General

There are a number of architecture-independent costs for in-kernel preemption:

1. increased difficulty in maintenance and debugging due to the possibility race conditions
2. cycles lost to potential, extra locking (which become significant if preemption is rare)
3. with the single stack kernel [War05], additional cycles lost to recovering from a preemption, as the stack has been dropped (in fact, Matthew Warton later found that this cost was so high that interrupts were later changed to use a separate kernel stack in the *single stack* kernel!)
4. non-linear execution complicates mathematical verification

Blackfin

The Blackfin architecture adds an additional number of hurdles in supporting preemption.

Firstly, the Blackfin masks interrupts during exceptions so preemption is impossible during exceptions unless we defer the work to an interrupt. Such a mode switch is expensive and, as we later describe ways of eliminating the switch [p89, Chapter 7.2.2], it is counterproductive to force us to perform the switch just for the sake of supporting preemption.

Secondly, recall that on the Blackfin, it is difficult to return to userspace when serving the nested interrupts [p20, Chapter 2.2] caused by preemption.

We do not want preemption

Preemption decreases interrupt latency but also decreases maintainability. Blackfin makes

preemption even more difficult. As a result, we do not want preemption – at least not for the duration of this thesis.

Therefore, the Blackfin port only permits:

1. interrupts
2. exceptions
3. exceptions during an interrupt (for in-kernel DCPLB misses and trapping kernel bugs)

In-kernel exceptions during an interrupt handler never need to switch threads as in-kernel CPLB misses can never result in a pagefault. This is convenient as the single kernel stack design does not permit thread switches when trapped from kernel mode.

5.1.2 Idle Loop

There is one other place in L4 that is normally preemptible: the idle loop `processor_sleep()` which sleeps, waiting for interrupts in kernel mode.

However, as discussed in the previous section, our kernel is strictly not preemptible and disallows interrupts in kernel mode. Therefore, `processor_sleep()` is simulated by a usermode idle loop and the kernel is not preemptible by interrupts at all.

5.2 N-series API

In order to remain supported within the research group and to restrict the thesis to Blackfin and physical memory issues, the design of the Blackfin port is based on the N-series of the L4 kernel, which requires the root task to be completed trusted. In this section, we deal with consequences of this in terms of security, memory mapped registers and DMA.

5.2.1 Security

Versions of L4 prior to the N-series contained a mapping database in the kernel, which allowed arbitrary threads to share pages with other threads (known as *mapping*) and irrevocably transfer a page mapping to another thread (known as *granting*) [Uns03]. The N-series did away with the mapping database claiming kernel complexity and code size reasons.

Both kernel series had the concept of *privileged threads* (i.e. those in the root task's address space) which can execute privileged calls, such as `L4_ThreadControl()` for creating a thread. Mapping and granting pages were normal unprivileged IPC operations that any thread could execute. However, in the N-series, only a privileged `L4_MapControl()` call is provided. The responsibilities of the mapping database have been pushed out of the kernel and into the privileged root task.

As a result, the root task must now be completely trusted. While the old kernel could continue mapping pages even if the root server died, the new kernel cannot. Furthermore, the root task can now access the memory of all other threads in the system. This seems in opposition to the memory/bug isolation and security ideals of object-oriented microkernel-based systems. A single bug in the root task can now compromise the whole system. In the old kernel, an encryption task could, for example, be irrevocably granted pages by the root task and after that, the kernel would ensure that the root task would not be able to read the encrypter's memory, which could contain sensitive data such as private keys.

5.2.2 Memory Mapped Registers

Since the root task is already trusted, we can allow it to access the CPU's memory mapped registers and also, device mapped memory. These pages can also be mapped to other tasks.

Of course, accesses to such memory registers can crash the whole system. So we could make the kernel more robust by combing through all the exposed memory mapped registers and determining which can and cannot be accessed by the root task. For instance, the root task should not be able to mask interrupts or change protection unit entries. However, as the root task is completely trusted anyway, there is little motivation to do this and we simplify kernel construction as a result.

5.2.3 DMA

On the Blackfin, DMA is manipulated using memory mapped registers. It is up to the root task to ensure cache coherency by using the new *N2* cache management API. We can leave this to the root task as the N2 design already trusts it.

5.3 Conclusion

Preemption causes too many problems so will not be enabled. As a result, our port will not support interrupts in kernel mode. Only in-kernel exceptions for DCPLB misses will be permitted. The idle thread is moved into userspace for a strict rejection of preemption.

We base our work on the N-series API which features a completely trusted root task. This is insecure but means that kernel construction is simplified: we can lazily expose all memory-mapped registers, and the responsibility of guaranteeing memory coherence when dealing with DMA, to root task.

Chapter 6

6 Physical Memory Implications

One key features of Blackfin is that it does not support virtual memory. In this chapter we analyse the effects of this on the design of both userspace and the kernel.

6.1 Userspace changes

In this section, we discuss the changes to userspace that result form the lack of virtual memory. We consider program loading, a physical memory space and swapping.

6.1.1 Program Loading

Programs must be compiled with fixed addresses unless Position Independent Code (PIC) is used in conjunction with a loader. As an example of the latter, ucLinux programs use the "flat binary" format and a kernel loader performs the necessary relocations [Hen06b].

6.1.2 Physical memory space

A single physical memory space is shared between the kernel and *every* program and obviously, collisions are not permitted. Depending on the kernel and other loaded programs, the space left for a new program may be too discontiguous to use. So the kernel must be constructed to minimise gaps in the memory layout, as must userspace.

Furthermore, the root task, when given an overlapping page mapping request, must decide whether the page is to be intentionally shared with another address space or whether the intention was to map a page in the same place as an existing page (an illegal operation). Hence, the root task hence has similarities to a *single-address-space operating system* [SLF+94].

6.1.3 Swapping

Transparent paging to disk is impossible as there is no virtual memory. However, explicit paging such as the use of overlays, where different code or data resides at the same address at different times [Pan68], is possible but this is not transparent to the program.

6.2 Kernel changes

In this section we consider possible changes to the kernel with the removal of virtual memory. , Mappings, the Kernel Interface Page, the TCB array and UTCB issues are discussed.

6.2.1 Mappings

As virtual addresses are identical to physical addresses, 1:1 mappings are enforced in the kernel's page table code.

6.2.2 Kernel Interface Page

Mapping the Kernel Interface Page (known as the *KIP*) into multiple address spaces (and therefore, at multiple addresses) would result in data redundancy. Therefore, the KIP is in a fixed location.

The L4 API already allows for this. The KIP's size is reported as the special value of 0 meaning that "the KIP is not part of the user address space and cannot be controlled" [Nic05a].

There is really no loss in trademark microkernel flexibility here. Regardless of whether a system has virtual memory or not, some part of the address space will always be reserved for the kernel image. So if one considers the KIP to be part of the kernel image, then fixing the position of the KIP has not lost us anything.

6.2.3 TCB Array

As previously noted, architectures without physical memory, by definition must use physical TCB addressing [Nou05].

6.2.4 UTCB Area

In L4 ports with virtual memory, the address space creator, through the `L4_SpaceControl()` system call, reserves an aligned UTCB array large enough to support the maximum number of threads. However, this does not work well for physical memory architectures.

A motivation for this scheme is that different threads in the same address space can share an MPU superpage entry, covering multiple UTCBs, saving on the cost of MPU refills for userspace UTCB accesses. A previous additional motivation was to be able to calculate a thread's UTCB address, given a *Local Thread ID*, without accessing the TCB and touching a data cache line. But Local Thread IDs no longer exist so this is no longer a consideration.

However, on a physical memory system where virtual memory is physical memory, this

reservation is very wasteful. A solution is to move all UTCBs into the kernel heap and leave UTCB allocation up to the kernel. The API already provides for this, in a similar fashion to the KIP: the UTCB minimum size is reported as the special value of 0.

Tempting as this solution may be, it does not work well for the Blackfin. As the reasoning is quite involved, we defer the discussion until we discuss the underlying problem of *protected kernel addressing* [p86, Chapter 7].

6.2.5 Message Registers in the UTCB

In this section, we discuss the motivation for storing message registers in the UTCB and then consider whether they are a good abstraction on architectures without virtual memory.

Why are they normally in the UTCB?

Some message registers stored are in memory as there is usually an insufficient number of real CPU registers on the architecture to store medium-sized messages – of say 60 words. However, they are specifically stored in pinned UTCBs to avoid page faults in kernel mode.

On a system with virtual memory, an invariant is that a UTCB will be backed by physical memory and will not be swapped to disk. This is enforced by the kernel allocating the backing inside the unswappable kernel heap.

To construct a message, the user thread copies data from its swappable virtual memory (probably the data segment) into the pinned UTCB's message registers (using the userspace function `L4_MsgLoad()`). This pinning ensures that kernel accesses to memory-backed message registers during an IPC will not result in page faults. This simplifies kernel construction, makes it a better candidate for mathematical verification and removes a class of security holes. In fact, the same reasoning motivated the removal of Long IPC from the N-series.

We might be tempted to dump them

However, for systems without virtual memory, pages cannot be swapped. So at first, there seems to be little sense to force a thread to do an extra memory copy from its *already unswappable* data segment into the message registers before a message can be sent. The memory-backed message registers abstraction would appear to make no sense – it could be dumped and the IPC call modified to be given the address and length of a message.

Register-backed message registers would still be maintained as they would not suffer from this double copying.

Of course, userlevel would still have the flexibility to simulate memory-backed message registers by copying data to an array before sending it, if it so desired – perhaps most of the data to be sent is scattered across memory and needs to be brought together. But the kernel would not mandate it as it would be an unnecessary performance penalty for other applications.

However, this scheme is flawed because the IPC implementation would have to walk the page table to check every given address. This is an unacceptable performance penalty compared to simply using a fixed UTCB address, which is known by the kernel to always be valid.

Therefore, the memory-backed message registers abstraction is not changed by the removal of virtual memory.

6.3 Conclusion

The number of changes required to support a full operating system without virtual memory are surprisingly small, although each change is quite significant.

The changes to userspace are:

1. Programs must be compiled with fixed addresses or relocated at load time.
2. With a single physical memory space:
   a) the user and kernel must aim to minimise external fragmentation
   b) the root task must determine whether attempts to overlap mappings are for the purposes of sharing pages.
3. Transparent swapping is not available but explicit overlays are.

The changes to the kernel are:

1. The kernel must enforce 1:1 virtual-to-physical mappings.
2. The Kernel Interface Page should be fixed in one global location to avoid duplicating it in every address space and wasting physical memory.
3. The TCB array must be physically addressed.
4. A virtual UTCB area per address space wastes physical memory. However, Blackfin's protected kernel addressing adds further complications and warrants extra investigation

The memory-backed message registers abstraction is not changed by the removal of virtual memory.

Chapter 7

7 Protected Kernel Addressing

We call a processor a *protected kernel addressing* architecture if both:

1. Kernel memory accesses are subject to the protection and refill mechanisms of its MPU
2. There is no way to avoid this without unacceptable performance loss

This is independent of whether it supports virtual memory. Protected kernel addressing potentially introduces MPU misses in kernel mode.

Blackfin has protected kernel addressing as all kernel accesses are subject to the protection and refill mechanisms of its DCPLB and ICPLB units. Turning off protection disables caching which results in unacceptable performance.

ARM1156T2-S (MPU) also has protected kernel addressing. While disabling protection merely results in a default memory map with accesses to the bottom 1.5GB or 2GB being cached (depending on the CP15 register), it can only be performed after an expensive flush of both instruction and data caches [Arm05].

In this chapter, we introduce the advantages and disadvantages of this feature. We then consider how we work around in-kernel MPU misses' nested exceptions, which are disallowed by Blackfin. We the analyse one approach to eliminating MPU misses in detail – restructuring the UTCB Area.

7.1 Advantages and Disadvantages

The advantage of protected kernel addressing is that in-kernel MPU misses are a way of detecting kernel memory bugs.

The disadvantage is that MPU misses in kernel mode result in additional complexity [p44, Chapter 3.3.3] and performance issues.

Firstly, the kernel must lock its trap code into the MPU, reducing the number of scarce MPU entries available to userland.

Secondly, on architectures with software-refilled MPUs, such as Blackfin, the cost of MPU misses is very high and made worse by the associated pipeline and cache pollution (the final L4/Blackfin port performs a refill in 1,425.31 cycles [p177, Chapter 17.1]). However, on a hardware-refilled architecture, such as the ARM1156T2-S, the performance impact is less pronounced.

Thirdly, as we discussed, in-kernel MPU misses occur on UTCBs. If we wish to maintain the ability of detecting kernel memory bugs, we must be able to distinguish buggy accesses from UTCB misses. In order to check MPU misses on UTCBs in non-current address spaces, such as via an inter-address-space IPC, we need an additional data structure containing pointers to all UTCBs. For relative simplicity, we currently insert *all* threads' UTCB pointers into kernel space's page table, in addition to the threads' respective address spaces' page tables. This is safe because the only thread in kernel space is the idle thread and it has no UTCBs of its own. However, this extra need to have a global UTCB list, for the sake of being able to trap kernel bugs, adds to kernel complexity.

Ideally, we would like to use architectures where protected kernel addressing can be turned on and off. Then one would have the best of both worlds: for a debug kernel, one would enable it for trapping kernel memory bugs and for a release kernel, one would disable it to reduce kernel complexity.

In summary, protected kernel addressing can trap kernel memory bugs. However, the introduced MPU misses increase kernel complexity and reduce performance.

7.2 Nested Exceptions

The possibility of MPU misses on UTCBs means that MPU miss exceptions may occur in the middle of servicing a system call exception. We firstly consider the effect this has on Blackfin system calls. We then propose solutions for eliminating these miss exceptions.

7.2.1 Blackfin system calls

This section provides the implementation details for system calls on Blackfin in the presence of in-kernel MPU misses and justifies that the implementation works even in the trickiest of solutions.

On the Blackfin, system calls can only be implemented as exceptions. Jumping into or accessing an invalid address also generates an exception. Raising interrupts in software can only be done in kernel mode.

The Blackfin core enters an unrecoverable double fault condition if an exception occurs during an exception. Since system calls can only be implemented as exceptions, a *potential D*CPLB miss exception in kernel mode would double fault.

However, exceptions (and therefore, DCPLB miss exceptions) are allowed to occur during an interrupt. Therefore, to cater for possible DCPLB miss exceptions, system calls must switch from the CPU exception mode to the interrupt mode. Recall that the ARM also does not support nested exceptions of the same type, as the link register is trashed [p37, Chapter 2.6.1], so a similar trick must be used on ARM.

System calls exit the kernel and arrange for a special software interrupt 15 (used only by the kernel) to fire straight away to re-enter the kernel so that the system call can be safely serviced. Other interrupts are masked during this mode switch to prevent an unexpected thread switch.

Works even in tricky situations

This mechanism has been well-tested even in the situation where the user jumps to a system call instruction at the end of an unmapped page and the next page is also unmapped.

The Blackfin issues an ICPLB miss for the first unmapped page which contains the system call instruction.

After this is resolved, it executes the system call exception instruction. The kernel is entered, raises interrupt 15 and exits, expecting re-entry immediately.

As exiting from a system call exception automatically advances the instruction pointer to the next instruction, which in this case, is pointing to the second unmapped page, we might expect that instead of our interrupt 15 being triggered next, that an ICPLB miss on the second unmapped page would be prioritised over it. After all, the Blackfin normally prioritises an exception over an interrupt.

However, as if it were an act from god, this amazingly does not happen. Our interrupt 15 is served before the ICPLB miss, as we wanted. This might be because the Blackfin realises that the interrupt is pending so does not trigger a ICPLB miss on the second unmapped page as it has not started to access it.

7.2.2 Avoiding nested exceptions

The trick we described in the previous section – system call exceptions switching to interrupt mode in anticipation of a possible in-kernel MPU miss – while also used by ucLinux, is a waste of cycles. We would like to find some alternative, less complex solutions that don't perform crazy mode switches and hopefully perform well also. Therefore, we consider 4 different methods:

1. Avoid DCPLB misses using locking

We later consider locking UTCBs in a kernel superpage to eliminate in-kernel DCPLB misses [p92, Chapter 7.3.3].

2. Mode switch only when strictly necessary

Some system calls, such as `L4_KernelInterface()`, and others for some combination of arguments, do not access UTCBs. Therefore DCPLB misses will never occur so the system call need not perform the interrupt switch.

Unfortunately, this does not help with the main indicator of performance, IPC, as it may generate DCPLB misses on UTCBs. However, if the DCPLB refill path becomes hand-optimised assembler, this may not be such a problem.

3. Pre-fill the DCPLB

At all points at which a DCPLB miss may occur, we could call the DCPLB refill code. This is pessimistic because we add this overhead for every access on the assumption that is miss is more likely to happen that not.

This introduces an expensive probe of the DCPLB to see if the relevant page is already in there. However, in exchange, when an actual DCPLB refill is required, we've avoided an unnecessary trap and untrap into the kernel to serve the DCPLB miss.

And of course, we also avoid this crazy interrupt switching scheme.

DCPLB misses on the KIP too

Recall that the current UTCB pointer needs to be updated on every thread switch and that we proposed placing this pointer into the unlocked KIP page [p61, Chapter 4.5.2]. The potential

DCPLB miss on the KIP in kernel mode is a very similar problem to the one we are discussing – DCPLB misses on UTCBs. The difference is that there is a single KIP but many UTCBs. Therefore, it is quite feasible to keep a single global variable stating whether the KIP is in the DCPLB – we do not need to probe the DCPLB. Before every kernel KIP access, we simply query this variable to see if we need to invoke the DCPLB refill code.

As the issue of DCPLB misses on the KIP is straightforward to work around, we avoid mentioning it in the future, to simplify explanations.

4. Disable the protection

At all points at which a DCPLB miss may occur, we could disable data protection temporarily. This bypasses the protected kernel addressing problem of Blackfin and ARM1156T2-S.

However, on the Blackfin, disabling data protection disables data caching. This is probably not so significant since I-cache will still be enabled and we usually aren't copying very much (e.g. a small number of message registers) minimising the effect of no D-cache. Unfortunately ARM requires expensive flushes of its caches to disable protection [Arm05].

Also, disabling data protection means that kernel bugs will not get trapped but for short code sequences, this should not be a problem. Nevertheless this is also a pessimistic approach since we may be disabling protection unnecessarily when the page in question is already in the DCPLB.

5. Analysis of these 4 options

We consider locking UTCBs in a later section. Only mode switching when strictly necessary does not improve IPC performance, which is unfortunately, basically the only thing we care about. Due to time constraints and the intention to minimise changes to the kernel, we did not implement pessimistic approaches of pre-filling the DCPLB and disabling the protection. However, we have described the approaches so that future work can investigate them.

7.2.3 Conclusion about nested exceptions

Blackfin system calls, implemented as exceptions, waste time switching to an interrupt mode, in case an in-kernel DCPLB miss on a UTCB occurs. We briefly describe ways of avoid DCPLB misses. The next section is devoted to one such way – locking UTCBs.

7.3 UTCB Area

Up until now, the reader has had to accept our claim that protected kernel addressing combined with a lack of virtual memory results in in-kernel MPU misses on the UTCB Area. In this section, we justify why. We then propose ways to restructure the UTCB Area.

On a protected kernel addressing architecture *with* virtual memory however, two virtual mappings for the UTCBs are made. One is inside the UTCB Area specified by the `L4_SpaceControl()` system call. The other is inside the locked kernel image. In-kernel MPU misses on UTCBs will not occur since the kernel can use the address inside the locked kernel image.

But mappings, let alone multiple ones, cannot exist on architectures *without* virtual memory. Therefore on such architectures, like Blackfin, we suffer from in-kernel MPU misses on UTCBs.

So, let us consider 4 ways of placing UTCBs in the address space, under the assumptions of protected kernel addressing but no virtual memory, with a view to minimising memory use and avoiding UTCB misses in kernel mode.

7.3.1 Option A: Reserve UTCB Areas in userspace

Our first option is the standard `L4_SpaceControl()` approach of reserving a UTCB Area, on address space creation. Unfortunately, as we previously mentioned, this wastes memory because virtual memory is physical memory [p82, Chapter 6.2.4]. Furthermore, there are those in-kernel MPU misses we wish to avoid due to the use of userspace memory.

Address spaces often wish to support widely varying numbers of threads so the maximum number of supported threads may be very large. As a result, the UTCB Area, reserved in physical memory may be very large. This is a waste of memory unless the maximum number of threads are actually created, which we anticipate does not occur very often.

Kernel accesses to the UTCBs cause MPU misses. Locking the UTCB Area superpage of the current address space into the protection unit will not eliminate all misses because it does not handle inter-address-space IPCs where the destination address space's UTCB Area superpage is not in the protection unit.

This is the worst possible approach since it wastes memory and has in-kernel MPU misses.

7.3.2 Option B: Never reserve UTCB Areas

An improvement on the previous solution is to eliminate memory wastage by changing the API so that `L4_SpaceControl()` will not reserve any actual memory for the UTCB Area – only stipulate where UTCBs may be placed. In this section, we elaborate on the features of this scheme and then consider a problem it raises with overlapping mappings.

Normal pages can be mapped into the UTCB Area reserved by `L4_SpaceControl()`. Therefore, there is nothing wrong with saying that the UTCB Area is as big as the entire Blackfin address space. UTCB memory will only actually be allocated by `L4_ThreadControl()` in userspace. The use of user memory for storing the UTCBs means that kernel-mode UTCB misses can still occur.

As with Option A, it is still possible to have, and take advantage of, a DCPLB superpage that covers the UTCB Area, for reducing DCPLB pressure – it's just that such a superpage may be covering a mix of UTCBs and ordinary data/instruction pages.

A problem with this scheme is that userspace may attempt to map a normal page where a UTCB already exists or vice-versa. However, this is a similar problem to mapping a normal page where another page from another address space is already mapped. As the root task already needs to have the ability to determine whether normal map requests to the same page, from different address spaces, are for the purposes of sharing data or are bugs [p81, Chapter 6.1.2], it is already likely to be maintaining a frametable. This frametable could also be used to prevent UTCB mappings overlapping normal mappings in userspace – it would be a waste of space to implement this functionality in the kernel since it would duplicate the frametable already in userspace.

In summary, this option eliminates memory wastage as no memory is reserved for UTCBs. However, in-kernel UTCB misses still occur.

7.3.3 Option C: Create UTCBs on demand in kspace

This option moves all UTCBs into the kernel heap and leaves UTCB allocation up to the kernel, as previously noted [p82, Chapter 6.2.4]. They can be allocated on demand using the in-kernel kmem allocator and so, this option never wastes memory. As all UTCBs are inside the locked kernel superpage, in-kernel DCPLB misses also cannot occur. We now describe the necessary implementation details.

However, as all UTCBs are stored in kernel space, when a user thread attempts to access its UTCB, a protection violation. The solution, in response to this, is to open up a window to the current UTCB, by adding an appropriate MPU entry. Of course, this entry would have to overlap the kernel image superpage. This option, by no means, wastes an extra MPU entry as no matter what scheme we attempt, the user still needs MPU entries covering what ever pages it wishes to access (in this case, its UTCBs).

On an address space switch, we would need to incur the overhead of removing the window to prevent a permission leak. Strictly speaking, it should be removed on even intra-AS thread switches (as threads should only be able to access their own UTCBs) but no security hole is created by not doing so as all threads inside an address space have the same level of trust.

So while this scheme saves memory and eliminates in-kernel MPU misses, it increases the switching overhead. It also only works on the ARM1156T2-S (MPU) but not the Blackfin because Blackfin does not support overlapping MPU entries. So we need to consider another way of placing the UTCBs in locked kernel space:

Option C2: Create UTCBs on demand outside kernel heap

The problem with the previous solution is that the proposed window would overlap the locked kernel superpage, as UTCBs are stored inside that superpage. Therefore, we propose moving UTCBs into a separate superpage, still in kernel space, that we handle specially to avoid overlapping MPU entries. But this means that we cannot use the dynamic kmem allocator so we end up with a fixed-size UTCB array.

But how are we to open up a window for the current UTCB when this new UTCB array is locked and Blackfin does not support overlapping pages? The solution is as follows. On the kernel trap, we ensure that the UTCB array is locked so that no in-kernel MPU misses occur. On the untrap back into userspace, we *exchange* the MPU entry, pointing to the UTCB array, with a much smaller entry that points to the current thread's UTCB. Notice that in our previous design, we could not perform this exchange as it would have been with the kernel image superpage and the kernel image must always be locked for the trap code [p87, Chapter 7.1].

However, this scheme permanently uses an extra DCPLB entry, reducing the number of available entries for userspace, especially for data-bound tasks that rarely access the UTCB. Furthermore, changing a protection unit entry on every trap and every untrap may be expensive, depending on the architecture. On the Blackfin each protection unit changes takes at least 50 cycles, excluding the time needed to calculate the immediate values used below:

```
// Disable protection, as required by the chip [p29, Chapter 2.3.3].
p0.h = DMEM_CONTROL;                    // 1 cycle
p0.l = DMEM_CONTROL;                    // 1 cycle
r0.h = mask for disabling protection;   // 1 cycle
r0.l = mask for disabling protection;   // 1 cycle
[p0] = r0;                              // 4 cycles (MMR access)
ssync;                              // min. 10 cycles (pipeline flush)

// Set the address.
p0.h = DCPLB_ADDR + dcplb_index;        // 1 cycle
p0.l = DCPLB_ADDR + dcplb_index;        // 1 cycle
r0.h = address;                         // 1 cycle
r0.l = address;                         // 1 cycle
[p0] = r0;                              // 4 cycles (MMR access)

// Set the page size and permissions.
p0.h = DCPLB_DATA + dcplb_index;        // 1 cycle
p0.l = DCPLB_DATA + dcplb_index;        // 1 cycle
r0.h = page size and permissions;       // 1 cycle
r0.l = page size and permissions;       // 1 cycle
p0 = r0;                                // 4 cycles (MMR access)

// Re-enable protection.
r0.h = mask for enabling protection;    // 1 cycle
r0.l = mask for enable protection;      // 1 cycle
[p0] = r0;                              // 4 cycles (MMR access)
ssync;                              // min. 10 cycles (pipeline flush)
```

This adds a penalty of more than 100 cycles per IPC, which is almost the cost of an *entire* IPC on the virtual memory ARM port!

Furthermore, a fixed-size UTCB array puts an additional cap on the number of threads

supported because it does not share the kmem heap with other kernel structures. Recall that if we were to store the array in the kmem heap, we could not open up a window to the current UTCB as we cannot unlock the kernel superpage nor does Blackfin support overlapping DCPLB entries. A fixed-size UTCB array wastes memory except in the unlikely case that all UTCBs are in use. However it is a big saving over reserving a UTCB array for every address space, as proposed by Option A.

In summary, this solution wastes memory in a global UTCB array, caps the number of threads, permanently occupies an additional MPU entry and incurs additional trapping overhead. However, it wastes less memory than reserving a UTCB array per address space and eliminates in-kernel MPU misses.

7.3.4 Choosing a solution

The following table summarises the findings of the above discussion on design alternatives for the UTCB Area:

Option	***Memory wasted on unused UTCBs?***	***In-kernel MPU misses?***	***Requires MPU overlapping page support?***	***Other disadvantages?***
A [A]	Yes: UTCB Area per address space	Yes	No	No
B [B]	No	Yes	No	No
C [C]	No	No	Yes	Yes: thread switching overhead
C2 [C2]	Yes: global UTCB array (less wastage than Option A)	No	No	Yes: trapping overhead, extra DCPLB entry, additional cap on number of threads supported

Table 7: UTCB Area design alternatives

In this section, we stress again that we are only considering architectures without virtual memory. We provide an analysis of the above options without, and with, protected kernel addressing. Of the latter, we consider the case where the architecture supports MPU entries that refer to overlapping pages and where the architecture does not.

Physical memory but no protected kernel addressing

As an interesting side-note, had the architectures in question not featured protected kernel addressing (and therefore, in-kernel MPU misses would never occur), Option B, in which

[A] *Reserve UTCB areas in userspace* – standard `L4_SpaceControl()` approach of reserving a UTCB area per address space

[B] *Never reserve UTCB areas* – `L4_ThreadControl()` allocates UTCBs, not `L4_SpaceControl()`

[C] *Create UTCBs on demand in kspace* – moves all UTCBs into the kernel heap and leaves UTCB allocation up to the kernel

[C2] *Create UTCBs on demand outside kernel heap* – global UTCB array in kernel space but outside of the kernel heap

`L4_ThreadControl()` allocates UTCBs, not `L4_SpaceControl()`, is the best solution as it has no real minuses.

ARM1156T2-S – effect of supporting overlapping MPU pages

Recall that ARM1156T2-S supports overlapping pages in its MPU. Furthermore, protection changes are cheap because, unlike Blackfin, the protection unit need not be disabled before modifying an entry. Therefore, an excellent solution for ARM is Option C, which creates UTCBs on demand in kernel space, as it does not waste memory, avoids in-kernel MPU misses and thread switching overhead is minimal due to cheap protection changes.

Blackfin – inability to support overlapping MPU pages

However as Blackfin does not support overlapping MPU pages we can only choose between options A, B and C2 – all of which have significant disadvantages. However, Option B is clearly equal to, or better than, Option A in all respects. Therefore, we need to decide between Option B and Option C2.

Option C2's global UTCB array might not waste so much memory on unused UTCBs as we might think. Blackfin only supports a maximum of 128MB of RAM. For systems of this size, we would usually – but of course, this depends on system design – not expect more than 1,024 threads. Therefore, a fixed-size UTCB array does not really cap the number of supported threads and reserving an array of 1,024 UTCBs, which are currently 1KB in size, would take only 1MB of RAM, which is palatable (if just). Even better, we know of ways to decrease the size of UTCBs by an order of magnitude [p100, Chapter 8.1] making the amount of wasted memory far smaller. Furthermore, we could minimise this wastage even further by specifying of the maximum number of threads as a kernel compile option, to adapt the kernel to different expected workloads.

We wish to avoid Option B's in-kernel UTCB misses due to the complexity reasons cited before [p44, Chapter 3.3.3]. For this reason alone, we believe Option B should be avoided.

As Blackfin has a software-refilled DCPLB, Option B's in-kernel UTCB misses are even more expensive. And furthermore, our implementation of DCPLB refilling is expensive, even in the final kernel, at 1,425.31 cycles. But is it possible to optimise this to below the additional trapping overhead of Option C2, which is believed to be slightly over 100 cycles [p93, Chapter 7.3.3]? We don't think so as we know that 50 cycles are needed for inserting an entry in the DCPLB alone [p93, Chapter 7.3.3] and we still need to traverse a 2 level page table that is actually 4 levels deep [p65, Chapter 4.5.4] in order to find the entry in the first place.

While we are leaning towards Option C2, we have not discussed its remaining disadvantage over Option B – namely that it hogs a precious DCPLB entry. This is regrettable but with no macrobenchmarks available [p48, Chapter 4.1], it is difficult to quantify the effect of this. So we are really stuck between a rock and a hard place where Option B suffers from the complexity and the performance hit of in-kernel UTCB misses but Option C2 uses an extra DCPLB entry. However, at least in microbenchmark terms, Option C2 is definitely faster than Option B as it avoids DCPLB refills. Having previously stated that Option B should be avoided due to its in-kernel MPU misses, we hesitantly suggest implementing Option C2.

Summary

In summary, if we do not have protected kernel addressing, Option B, in which `L4_ThreadControl()` allocates UTCBs instead of `L4_SpaceControl()`, is the best solution as it does not waste memory. If we do have protected kernel addressing, the deciding factor is whether or not the MPU supports entries pointing to overlapping pages.

On an architecture that supports overlapping MPU entries, such as ARM1156T-2, Option C, which creates UTCBs on demand in the kernel heap, is optimal as it does not waste physical memory and avoids the complexity of in-kernel MPU misses.

On architectures without overlapping MPU entry support, such as Blackfin, we have a difficult choice between Option B, in which `L4_ThreadControl()` allocates UTCBs instead of `L4_SpaceControl()`, and Option C2, which allocates UTCBs on demand from a global UTCB array in kernel space. While future work is to measure the performance difference under a macrobenchmark, we choose Option C2 for now, as it avoids the in-kernel MPU miss complexity of Option B.

7.3.5 Conclusions on the UTCB Area

The location and structure of UTCB Areas, on physical memory and protected kernel addressing architectures, is subject to a *complex* number of trade-offs involving memory, in-kernel MPU misses on UTCBs and other factors:

1. Ordinary UTCB Area reservation, in user memory, on address space creation (the standard L4 behaviour) wastes physical memory, introduces in-kernel MPU misses and is therefore, the worst design

2. Never reserving UTCB Areas but instead allocating UTCBs on demand in userspace does not waste physical memory but in-kernel MPU misses still occur

   - Requires an API change where `L4_SpaceControl()` sets the possible area where UTCBs can be located, but does not actually reserve any space, and `L4_ThreadControl()` allocates a UTCB anywhere in this area

   - UTCB and ordinary pages can be interleaved

   As a side note, if a physical architecture did not feature protected kernel addressing, we do not have to worry about the MPU misses and it is the ideal design since it does not waste memory.

3. Allocating UTCBs on demand in the kernel image does not waste physical memory. The locked kernel image avoids in-kernel MPU misses but introduces the following issues:

   - A userspace window for the current UTCB must be opened and is easily done on architectures that permit overlapping MPU pages (e.g. ARM1156T2-S)

   - Without overlapping MPU pages (e.g. Blackfin), a global UTCB array must be located outside of the kernel image and locked into the MPU, occupying another

MPU entry

- Exchanges of this entry with one that opens up the current UTCB for userspace must be performed on untraps and the vice-versa for traps
- The fixed-size global array caps the number of supported threads and wastes physical memory if few threads are created (physical memory wastage can be minimised by adjusting the number of supported threads to the expected maximum load at compilation time)

For Blackfin, we hesitantly choose the last option with a fixed global array to avoid in-kernel UTCB misses and the overhead of DCPLB refill routine. The latter could of course be avoided if Blackfin had a hardware-refilled MPU.

7.4 Conclusion

Protected kernel addressing, which kernel accesses are constrained by the MPU, creates the potential for in-kernel MPU misses, which increases kernel complexity and reduces performance (especially with software-refilled MPUs). It mandates the locking of the kernel image into the MPU, increasing MPU pressure.

Blackfin system calls need to waste time switching processor modes to work around this problem. We considered ways of avoiding in-kernel MPU misses with an in-depth analysis of UTCB Areas.

The location and structure of UTCB Areas, on physical memory and protected kernel addressing architectures, is subject to a *complex* number of trade-offs involving memory, in-kernel MPU misses on UTCBs and other factors.

Chapter 8

8 Saving Memory on UTCBs

In this chapter, we consider ways to reduce the amount of memory consumed by UTCBs. While other structures in L4 certainly take a lot more memory, there is no harm in considering improvements here.

We firstly describe the motivation for reducing the size of UTCBs and analyse the effects of achieving this by decreasing the number of memory-backed registers.

We then consider whether the memory for UTCBs should reclaimed as soon as the respective threads are deleted or whether we should wait until the address space is deleted.

8.1 Smaller UTCBs

The UTCB in the final L4/Blackfin port is made up of the following:

Offset (bytes)	*Component*	*Size (bytes)*
0	Non-Message-Register Fields	64
64	64 Message Registers	64 x 4 = 256
320	(padding to 1KB)	1024 – (64 + 256) = 704

Table 8: L4/Blackfin UTCB components

The padding conveniently ensures that the UTCB takes a full 1KB, to match the minimum Blackfin page size. However, we would like to reduce the size of UTCBs to save memory and as a result, increase MPU coverage.

If we remove the padding, only 320 bytes are actually needed for the UTCB. Even if we round that up to the nearest power of 2 (512) for simplicity, we can still fit more than 1 UTCB on a page.

However, we can do better than this and so the next section considers sacrificing message registers to make the UTCB even smaller. The subsequent section then reflects on whether a pathological worst case can defeat our optimisation.

8.1.1 Sacrificing message registers

In practice, current L4-based systems tend not to use more than half a dozen message registers. It therefore makes sense to reduce the size of the UTCB by decreasing the number of message registers or moving memory-backed message registers into CPU registers. Of course, the UTCB's 64 bytes of non-message-register fields are untouched by our optimisation attempts.

Suppose we would like a UTCB size of 256 bytes. We would only have room for `(256 - 64) / 4 = 48` memory-backed message registers. A UTCB size of 128 bytes would allow for just `(128 - 64) / 4 = 16` memory-backed message registers.

Currently, our kernel does not use *register-backed* message registers even though Blackfin has more than 40 registers. If a sufficient number of registers were used, one could do away with memory-backed message registers entirely, squeeze the UTCB down to 64 bytes and still support more than 32 message registers (as they would fit into registers).

We have to be careful with dumping message registers instead of moving them into CPU registers, which is necessary on other architectures which do not have as many available registers. With insufficient numbers of message registers, larger messages have to be sent either via multiple IPCs (which adds the overhead of extra system calls) or via shared memory (and one page per client and server would be an unreasonable waste of space for messages much smaller than the size of a page).

8.1.2 Pathological worst case

Assume that we have reduced the size of UTCBs, as per the previous section, so that more than one UTCB can fit on a page. Then, consider the case where the thread creator ensures that each UTCB is on a separate page, creating internal fragmentation. The benefits of reducing UTCB size – saving memory and increasing MPU coverage – have been eliminated since UTCBs still effectively consume one page each.

Thankfully, as the location of the UTCB is specified by privileged threads, we only have to trust the privileged threads (i.e. the root task), to not do this. This is consistent with the N-series design, where the root task is already so trusted that it can access the memory of all tasks in the system.

8.1.3 Conclusions on smaller UTCBs

Reducing the size of UTCBs saves memory and increases MPU coverage. Two ways to reduce the size of our UTCBs are:

1. Remove the padding up to the minimum page size
2. Reduce the number of message registers or move memory-backed message registers into CPU registers.

Decreasing the number of message registers may result in messages being inefficiently split into multiple IPC calls or force large messages to be sent via shared memory pages, which may suffer from significant internal fragmentation.

The trusted root task can ensure that the savings, made from reducing UTCB size, are realised by ensuring that the maximum number of UTCBs are packed into each page.

8.2 UTCB Deletion Trade-offs

In most ports, when a thread is deleted, its UTCB's page table mapping and physical backing are not freed. Instead, all UTCBs in an address space are finally deleted when the address space is torn down. We now consider the trade-offs in deleting UTCBs as soon as the respective threads are deleted.

8.2.1 Saving cycles v.s. saving memory

The delayed deletion scheme provides for efficient UTCB recycling as it avoids repeated UTCB deallocation and allocation when a thread ID is reused – cycles are being saved.

The trade-off is that memory is being wasted for non-existent threads. This could be significant if many threads, with different IDs, are created and deleted over time but only a very small number of threads are active at any one time.

We choose to delete UTCBs on thread deletion, instead of address space deletion, to save memory as it is precious on Blackfin – the architecture limits the maximum amount of supported memory to 128MB. However, there is an additional complication here:

8.2.2 Deleting immediately may waste memory!

If we do try to deallocate UTCBs immediately and we support multiple UTCBs per page, we have to avoid unmapping the page if there is another UTCB on that page. This would force us to introduce reference counting into the page table entries. However, this may actually cause us to waste more memory than we save due to enlarged page table entries or the need for a shadow page table, as we explain shortly.

Firstly, the addition of a reference counter into page table entries may enlarge the entries, if there are not enough free bits. If this occurs, we waste memory for the vast majority of page table entries, which are for normal, non-UTCB pages that do not require reference counting.

If there are few threads in the systems (and therefore few UTCBs) but many normal page mappings, we may be wasting more memory than is saved by the reduction in UTCB size. We could avoid this by creating a separate data structure just to maintain UTCB page reference counts but this adds even more complexity and chance of introducing bugs.

Secondly, hardware-walked page tables may specify a strict format for the page table entries. In this case, we would need a shadow page table, to accommodate the reference counting, consuming additional memory.

8.2.3 Conclusions on UTCB deletion

Deleting threads' UTCBs as the threads are deleted saves memory but consumes more cycles if

thread IDs are reused, compared to deleting the UTCBs when the address space is torn down. However, if multiple UTCBs per page are also supported, deleting UTCBs on thread deletion may actually waste more memory than is saved, through page reference counting bits. Therefore we cannot draw general conclusions on this matter without details of expected workload and page table entry structures.

8.3 Conclusion

Reducing the size of UTCBs saves memory and increases MPU coverage. However, if the number of message registers are reduced to achieve this goal, sending larger messages takes additional cycles or memory.

In the general case, it is unclear whether UTCBs should be deleted as soon as their respective threads are deleted, as opposed to deleting UTCBs at address space destruction.

Chapter 9

9 System Calls

The initial Blackfin port preserved the entire user context on system calls. In this chapter, we start by investigating the performance benefits and implications of the standard L4 behaviour of only preserving part of the context. We then consider what part of the context needs to be preserved.

Finally, the initial port also made the system call convention match the C function calling convention and forced the kernel to access the user stack to retrieve arguments. So we discuss how we can improve this – we discuss what registers should be used for the system call convention.

9.1 Preserving Part of the Trapframe

Some kernels, such as DOS, OS/161 [HLS02] and Linux save and restore all registers on system calls [Joh97]. No registers are clobbered except those used for return values.

L4 takes a more lightweight approach. Realising that it is only normal exceptions, such as MPU misses, that require full trapframe preservation in order to be transparent to the flow of execution, system calls are allowed to clobber almost all registers.

In this section, we discuss the performance advantages of this, as well as security concerns and implications for the trap code.

9.1.1 Performance improvement

On architectures with a large user context, such as the Blackfin where the context is the equivalent of forty-five 32-bit registers, preserving only part of the trapframe is a significant cycle and data cache saving, especially on the IPC fastpath.

By pushing the responsibility of saving registers from the kernel up into userland, we minimise the lower bound of the kernel's system call cost. We also reap performance benefits if userland either does not care that the vast majority of registers are clobbered and/or with smart compiler register allocation, avoids unnecessary register saves.

Of course, a few registers, such as the instruction and stack pointers, would still have to be maintained to permit userspace to save and retrieve any registers that need to be saved.

9.1.2 Security concerns

A concern is that saving less than a full trapframe introduces an information leak and is a security problem. For instance, consider a future L4 implementation that uses sparse capabilities, where the possession of a 64-bit value would entitle the holder to extra privileges. If L4 were to check if a system call was authorised by comparing the given supposed capability (provided by userspace) to the real capability value, the latter would be placed into registers by assembler. If the contents of these registers leaked into userspace, because we did not restore the full trapframe, then userspace could steal the capability.

9.1.3 Trap code implications

Saving less than the full trapframe introduced some implementation complexities in the trap.

In our initial kernel port, functions that implemented system calls (*syscall functions*) saved and restored the entire trapframe. The system calls returned values by writing to the memory copy of the user context stored in the TCB. The C++ code that called the syscall functions then returned to assembler. We shortly discuss the implications of the use of C++ here. The

assembler, when restoring the trapframe from the user context stored in the TCB, would, as a side effect, propagate the returned values to userland. System call arguments were handled in a similar fashion.

However, when we changed system calls to save and restore only a handful of registers to and from the trapframe stored in the TCB, a problem arose. Some user registers containing system call arguments were no longer being stored into the trapframe (which is read by the kernel syscall functions). Similarly, some registers used for syscall return values were no longer being restored from the trapframe (which is modified by the system call functions) and therefore, were not sent to userspace.

Also save and restore registers used by system calls

To solve the problem, the assembler could save and restore all the registers that *might* be used for system call arguments and return values to the memory-backed trapframe stored in the TCB. However, this solution defeats the point of saving and restoring a minimal number of registers. It makes no sense to save and restore 7 extra argument/return-value registers, for all system calls – when IPC only requires 2 – just because one system call (`L4_ExchangeRegisters()`) requires 7 registers.

What would be better would be to save and restore only the *necessary* argument/return-value registers, for the particular system call, to the memory-backed trapframe. Nevertheless, the logic required to determine which of the 7 registers to save and restore probably takes longer to execute that just blindly saving and restoring those registers.

Nevertheless, we haven't explained why we need such registers to save to a trapframe in the first place.

But why are we saving to the trapframe?

We are saving registers to the memory-backed trapframe, in the TCB, because the code that invokes syscall functions is written in C++ and the registers used for system call arguments might otherwise have been clobbered by the time the syscall functions were to access them. So what is happening is that we are saving registers to the memory-backed trapframe, only to read them back, in the syscall functions, almost immediately afterwards. It is a similar story with system call return registers.

The Blackfin port implements the code, that calls the syscall functions, in C++ in line with the philosophy of avoiding assembler at all costs for maintainability [p169, Chapter 15.3]. This is the only L4 port that does this. Therefore, the trade-off is that we must accept this minor performance penalty on the slowpath. For the hand-optimised assembler IPC fastpath, we can certainly use this optimisation, which avoids touching memory (the trapframe stored in the TCB).

9.1.4 Trapframe preservation conclusions

L4 system calls do not preserve the full trapframe and clobber almost all registers, in the hope that cycles can be saved through userlevel being able to avoid unnecessary register saves.

There is a concern that this creates an information leak.

Because we value debuggability and wrote some kernel system call handling code in C++, we are forced to perform extra register saves and restores. This does not matter, from a performance standpoint, since the main determinant of performance – the fastpath – will be written in hand-crafted assembler.

9.2 Registers to be Preserved

We want the trap to save the minimum number of registers – just enough for userland to recover from the system call. Recall that by saving as few registers as possible in kernel-mode, we save cycles if userspace does not care that certain registers are clobbered [p106, Chapter 9.1.1]. Userspace only needs to save the ones it is interested in. In this section, we identify the 3 categories of registers the trap code needs to preserve:

1. **Instruction Pointer**: To continue execution after a system call, the kernel needs to restore the user instruction pointer.

2. **Stack Pointer**: In order for userspace to be able to restore the registers it need to preserve across a system call, a pointer register must be preserved by the kernel. Given that even assembler programs are likely to use the stack, the stack pointer is the perfect candidate.

3. **"Register Room"**: Our trap code needs to decide whether the CPU exception is a system call or a CPLB miss. On a CPLB miss, where all registers must be saved, we would not have wanted this test to have clobbered any registers. Therefore, the trap code needs to save some registers before it does anything – enough to make this decision.

In general, there is no need nor a desire – due to efficiency reasons – to save any more registers in kernel mode. And in fact, if the architecture has banked registers (separate user and kernel registers), the banked registers need not even be saved, until a context switch.

Let us analyse our Blackfin port in terms of these 3 categories:

1. **Instruction Pointer**: The instruction pointer is banked.

2. **Stack Pointer**: The stack pointer is also banked. For both the instruction and stack pointers, we actually save them on the trap, instead of on context switches, which is unnecessarily inefficient for system calls that do not switch. However, this is irrelevant as the only system call we want to optimise on this level – IPC – always context switches.

3. **"Register Room"**: As Blackfin has no more banked registers, we need to save the flags (zero, carry etc.) register and a few general registers. However, we currently save all 10 registers preserved by C function call convention (4 'R' data registers, 4 'P' pointer registers, the `RETS` link register and the frame pointer). This is convenient because our userspace system call wrappers are currently functions so the wrappers need not do any extra register saving. Having said that, this is lazy and increases the lower bound system call cost for programs that do not require the C function call convention to be preserved e.g. assembler programs.

In summary, the 3 categories of registers that trap code needs to save are the instruction pointer, stack pointer and some "register room". The Blackfin currently saves too many registers to be efficient.

9.3 Registers for Args / Return Values

Now that we have decided what registers must be preserved to permit recovery after a system call, we need to determine which registers should be used for system call arguments and return values. An observation is that one should reuse registers used for arguments for return values. But there are possibly other concerns – such optimising for programs written in the most likely userspace language and for virtualising another kernel.

9.3.1 Likely userspace language

As we need to choose some registers for system calls, we are imposing some policy so should impose the one that optimises most user applications. At the time of writing, most Blackfin user applications will use C/C++ and conform to the Blackfin gcc C calling convention. We first describe the L4 system call mechanism and then investigate whether it makes sense to base our system call register choice on this convention.

L4 stores the code that actually invokes the system call exception in the KIP. The userspace system call library jumps into the KIP to execute the system call. This permits the system call mechanism to be changed in an ABI-transparent manner. It also means that all system calls incur the penalty of an additional jump and return (minimum of 5 cycles in total on the Blackfin) but that is another story.

If we simply make this jump and return, a C function call, a straightforward implementation would make the system call convention identical to the C calling convention so that no argument or return value marshalling at userlevel needs to be done. Unfortunately, the number of callee saved registers may be large (on the Blackfin, there are 10) so preserving the convention may be expensive.

Furthermore, the C calling convention may be insane. For instance, even though the Blackfin has more than 40 registers, only the first 3 arguments to a function are register-backed. The rest are stored on the stack. The initial kernel had to walk the page table to check that the arguments on the stack were accessible, leading to a grossly inefficient and complex kernel [p55, Chapter 4.3.2]. As a result, we changed the kernel so that userspace is now forced to copy all arguments to registers.

Having established that using the function calling convention for the system call convention is unreasonable, there is no reason to wrap the system call invoking code, in the KIP, in a C function call. But the question remains: which registers should we use for system call arguments and return values?

One might argue that we should only use *callee* saved registers on the grounds that since in C, code is always executing in a function and if that function is using callee saved registers, it has already saved them. We could also argue the other point of view: we should only use *caller* saved registers on the grounds that the current function need not save them if we modify them.

But both arguments are flawed because regardless of whether a register is callee or caller saved, at the point before the system call invocation, the compiler might have been using either or both

types. After all, an efficient compiler will try to maximise register usage. As a result, the compiler would still be forced to save and restore the previous contents of any register we wish to use for system calls, before and after the system call (we assume the register will likely still be used after the system call due to the compiler maximising register usage for a non-trivial program).

We therefore conclude that deriving a system calling convention based on a particular language's calling convention is meaningless. The choice of registers seems arbitrary and independent of the likely userspace language.

9.3.2 Virtualising Linux

ucLinux is an obvious candidate for virtualisation on top of L4 in a similar fashion to Wombat. In this section, we analyse what effects this might have on the system call convention.

The kernel is not permitted to use the same system call mechanism as an operating system it is trying to virtualise – otherwise it would not be able to distinguish native system calls from the calls to be virtualised. We cannot change the guest operating system's calling convention as we wish to preserve ABI compatibility.

ucLinux uses system call exception 0 (`EXCPT 0`), register `P0` for the system call number (from 0-255), registers `R0-R4` for arguments and register `R0` for the return value [Bla06d uClinux-dist/uClibc/libc/sysdeps/linux/bfin/syscall.c].

If we do not use exception 0 for L4 system calls, virtualising ucLinux is straightforward and has no bearing on our choice of registers for system calls. However, if we insist on using exception 0 for our own system calls, then `P0` must be used for the system call number and our system call numbers must be greater than 255 (i.e. outside of ucLinux's range).

Another issue is that as L4 does not preserve all registers on an exception but Linux expects it to do [Joh97], on detecting an virtualised system call, we must ensure that the full user context has not been clobbered.

9.3.3 Summary

The choice of registers for the system call convention is arbitrary. But making it match the C calling convention is usually inefficient and too complicated for the kernel – especially if it requires that arguments be placed on the stack, as was the case on Blackfin. The kernel system call convention may need to be modified to distinguish native system calls from the calls to be virtualised.

9.4 Conclusion

L4 system calls do not preserve the full trapframe for efficiency. There is a concern that this creates an information leak.

Having decided that we do not need to save the full trapframe, we determined that the 3 categories of registers we do need to save are the instruction pointer, stack pointer and some "register room".

Lastly, we found that picking registers for the system call convention is – as long as we are reasonable – an arbitrary decision, except for when we need to virtualise an operating system. The initial Blackfin port was an example of something that was not within reason as it had to walk the page table to check that the arguments on the stack were accessible, leading to a grossly inefficient and complex kernel.

Chapter 10

10 The Single Stack Kernel

This chapter firstly describes the method we used to port to the single stack kernel.

It then discusses aspects of the implementation and raises warnings about certain error-prone mechanisms.

10.1 Conversion procedures

Warton [War05] provides a fleeting explanation – at best – of how to convert from the multi-stack kernel to the single stack kernel. We fill in this gap by describing the changes to context switching and trapping.

10.1.1 Changes to switching

Switching threads in the previous, multi-stack kernel required changing over to the stack stored in the TCB being switched to. The switch functionality was written in assembler and wrapped in a C function. The assembler consisted of:

1. Pushing onto the old TCB's stack the:
   a) registers saved by the compiler-specific C functional calling convention
   b) return address
2. Manipulating the stack pointer to point to where it was on the new TCB's stack from a previous switch.
3. Restoring state from the new TCB's stack:
   a) the registers saved for the calling convention
   b) jumping to the return address

As a result, code that called the C switch function could continue executing normally after context switches, without any special handling.

But in contrast to the multi-stack kernel, with a single stack shared between all TCBs, the call history, along with local variables, must be dropped on a context switch. As the execution context is lost, switches become more explicit.

Any code that could potentially switch stacks must be split at the potential switch point. Before the possible switch, all state necessary to recover from a switch (local variables etc.) must be saved to the TCB. In addition, a pointer to the function containing the code that follows the switch - called a *continuation* [War05]- must also be stored in the TCB so that execution can continue after the switch back.

However, the switch function is much simpler as it only needs to:

1. Drop the stack by moving the stack pointer to the top of it
2. Jump to the continuation stored in the new TCB

As a result, there is no longer a need for a switchframe.

10.1.2 Changes to trapping

As the single stack kernel drops the stack on a switch, if trapping from kernel mode (to service an MPU miss), one can now only switch at special *preemption points* that contain a pointer to a recovery function that can deal with the complete loss of kernel context [War05]. As the Blackfin port, for simplicity, does not allow kernel preemption [p75, Chapter 5.1], this is not currently an issue.

However, loss of the contents of the stack on a switch, means that usermode context must be stored outside of it – in the TCB. Therefore, the old multi-stack kernel trapping behaviour of simply pushing all registers onto the stack and then, servicing the exception in question, regardless of whether we trapped from usermode or kernel mode, no longer works - trapping becomes more complicated.

We consider two single stack kernel trap designs.

Design 1 - kernel stack pointer points to either the TCB or stack

This is the design described briefly by Warton [War05]. With both ARM and Blackfin, the stack pointer is banked i.e. there are separate hardware-maintained user and kernel mode stack pointers. In usermode, the kernel stack pointer will point to the top of the user context in the TCB. On a trap into the kernel, it pushes the context into the TCB, treating the area as if it was the stack. It then switches to the real kernel stack.

On a trap from kernel mode, the kernel stack pointer already points to the kernel stack. So regardless of whether we came from usermode or kernel mode, both kinds of traps can stack registers immediately.

Design 2 - kernel stack pointer always points to the stack

Warton does not discuss this perfectly valid alternative where the kernel stack pointer always points to the real kernel stack. On a trap into the kernel, this design would save a few registers sufficient to allow it to determine whether it came from usermode or kernel mode without clobbering context. This saving is needed for Blackfin since it has no banked registers other than IP and SP.

If it came from usermode, it saves the remaining context into the TCB. The initial few registers it saved also need to be copied in the TCB in case they are lost in a stack-dropping switch. Otherwise, if it came from kernel mode, it simply stacks the remaining context.

Verdict

There is no real trade-off here, as Design 1 is clearly better - Design 2 takes more cycles because of the extra copying of those initial registers if trapping from usermode and is more

complicated. However, it is worth considering these two designs given that Warton did not.

10.1.3 Summary of conversion procedures

Execution context is lost on a switch so any code that potentially switches needs to be split at the potential switch point. All state needed to recover from the switch should be stored into the TCB. A continuation function restores this state and continues execution after the possible switch.

The trapping behaviour has changed. If trapping from usermode, the kernel must still push all the context into the TCB (it does this by arranging that the kernel stack pointer will point to top of the user context on the trap) but the difference is that it must then switch to the single kernel stack. However, if trapping from kernel mode, the kernel can simply push the context as before.

10.2 Implementation

We implemented the single stack kernel using the techniques described in the previous section and verified Matthew Warton's 2 week time estimate.

A number of issues were found during the implementation. We will therefore consider the boot stack, size of the TCB, flawed – or at least, confusing – single stack kernel mechanisms and performance.

10.2.1 Boot stack

The Blackfin port now uses a single 2KB stack. However, many L4 ports still have an 8KB boot stack in addition for no reason. This is approximately 6% of the size of the kernel image so is needlessly wasteful and should be removed.

10.2.2 TCB size

In this section, we look at the size of the TCB on Blackfin and an advantage of physically-addressed TCBs.

Blackfin

TCBs no longer contain kernel stacks so their size has been reduced from 4KB to 1KB, of which 604 bytes are actually used (the rest is for "power-of-2" alignment):

Field	*Size (bytes)*
Architecture independent fields	326
Continuations for `L4_ThreadControl()` to unmap UTCBs when deleting threads [*]	20
Blackfin-specific	248
Trapframe	45 registers * 4 = 180
General continuations and miscellaneous	68
Total:	***604***

Table 9: L4/Blackfin TCB fields

As `1,024 – 604 = 420` bytes are being wasted for alignment, it would be useful to attempt to compact the TCBs to below 512 bytes (the next power of 2). However, this is difficult given

[*] UTCBs are normally deleted when the address space is deleted but this is wasteful [p102, Chapter 8.2]. The ARM port also deletes the UTCBs quickly but uses a less clean method of deleting them that does not require continuations.

that the size of architecture independent part plus the mandatory trapframe for interrupts is already `326 + 180 = 506` bytes, leaving little room for any additional bookkeeping. Aggressive use of C's `union` mechanism would have to be used and apart from making the kernel more error prone, it would require much work to squeeze the TCB down to 512 bytes.

A benefit of physically-addressed TCBs

Suppose for a particular architecture, we managed to squeeze the size of a TCB down to 512 bytes. Assume that the minimum page size of the architecture is greater than 512 bytes (which is true on most architectures).

Now, if the TCB array is virtually allocated, a pathological thread ID allocation that touched each page would nullify the effect of reducing the size of the TCB due to internal fragmentation.

Luckily, Blackfin is physically addressed and the L4 port allocates TCBs on demand, avoiding this problem. Other architectures could also use physical addressing to avoid this problem [Nou05].

10.2.3 Mechanisms

Warton devised a number of primitives in his implementations of the single stack kernel, which the Blackfin port also uses. However, there are a number of caveats he did not mention:

ACTIVATE_CONTINUATION()

Recall that at all points where the kernel *might* switch, the code has been split into two with a recovery continuation function containing the code of the latter half. Because after returning from a switch, the stack has been trashed, continuations assume nothing about the contents of the stack. They restore the state saved before a potential switch and continue execution. They never return as the stack may be gone and instead, call other continuations to "exit".

However, if the kernel decides not to switch at a switch point, it must still call the continuation to complete its work. As continuations can make no assumptions about the stack, the `ACTIVATE_CONTINUATION()` macro not only invokes the given continuation but also moves to the top of the stack.

Warton says that this provides greater data cache locality and reduces the change of running off the stack. At first, the cache argument seems valid given that, in normal circumstances, calculating the top of the stack requires no additional memory accesses.

Note that functions that never return but exit via a call to a continuation instead can also use `ACTIVATE_CONTINUATION()`. The Blackfin port for some time used this in its CPLB protection miss handlers.

Now, `ACTIVATE_CONTINUATION()` jumping to the top of the stack is safe if we trapped from usermode. However, if we came from kernel mode, we would be corrupting the previous

kernel context stored on the stack. The only way we could have come from kernel mode would be a CPLB miss. Architectures with hardware loaded CPLBs are therefore not affected by this issue. Software loaded architectures, such as the MIPS64 and Blackfin, must either:

1. On all code paths that might be traversed when trapped from kernel mode, waste a small amount of stack by simply performing a function call to the continuation, instead of calling `ACTIVATE_CONTINUATION()`.

OR

2. `ACTIVATION_CONTINUATION()` must calculate the highest safe address on the stack to move to.

Option 1 wastes some data cache by needless storing the previous function's context which will never be returned to.

Option 2 also requires a data cache line as the highest safe address is stored in the TCB, after being calculated on the trap in. Depending on the implementation, it also adds unnecessary complexity and instructions (both cycles and cache lines): an early Blackfin implementation, even had a pipeline-burning conditional jump based on whether it trapped from usermode or kernel mode.

Therefore, Option 1 makes far more sense based on simplicity alone. This puts into question whether `ACTIVATE_CONTINUATION()` should really have this error-prone "move to the top of the stack" behaviour at all. In the Blackfin port, the function call history is at most 5 deep so unless functions have a large number of local variables, the claimed data cache savings may not be worth the increased chance of bugs.

Current TCB

In the multi-stack kernel where the current kernel stack was inside the current TCB, the address of the current TCB could be calculated by aligning the stack pointer against the size of a TCB.

In the single stack kernel, there is no relationship between the stack pointer and the address of the current TCB. Therefore, Warton places a pointer to the current TCB at the top of the kernel stack, updated on switches. This is for cache locality with the ordinary contents of the stack and straightforward SMP generalisation, where there is one kernel stack per CPU.

However, this behaviour is not obvious and other port developers need to be careful not to clobber the absolute top of the stack when implementing `ACTIVATE_CONTINUATION()`.

System call functions

In this section, we discuss a poor implementation decision regarding system call functions and continuation arguments that leads to an unreliable kernel.

Functions that may switch are normally given a continuation argument so that they can call the continuation to resume work, after the potential switch:

```
// Note the continuation argument.
```

```
void function_that_might_switch (continuation_t continuation)
{
      // --- Do some work ---

      ACTIVATE_CONTINUATION (continuation);
}

void caller (void)
{
      // --- Do some work ---

      function_that_might_switch (caller_finish);

      // --- We never get here since "caller_finish" is called
}

void called_finish (void)
{
      // --- Do some more work ---
}
```

However, functions that implement system calls are not given continuation arguments. They return to the address after the call - just like a normal function call - but the contents of the stack have been trashed (due to call to `ACTIVATE_CONTINUATION()`):

```
void syscall_function_that_might_switch (void)
{
      // --- Do some work ---

      ACTIVATE_CONTINUATION (__builtin_return_address (0)/*address
after the instruction that called us*/);
}

void caller (void)
{
      // --- Do some work ---

      syscall_function_that_might_switch ();

      // --- We arrive back here but the stack context is gone!
}
```

What's worse is that, callee saved registers in `syscall_function_that_might_switch()` have not been restored because the function exited via calling a continuation – not through the normal compiler-generated function return path.

Warton's justification, for not passing a continuation argument, is that it avoids dealing with complicated calling function conventions should `caller()` be written in assembler. However, this is crazy because with a corrupted stack context and callee saved registers that haven't been saved, a huge number of potential bugs due to a violated C++ calling convention, are introduced. As a proof of this point, the following code was in an original version of the single stack kernel:

```
INLINE void tcb_t::do_ipc (threadid_t to_tid, threadid_t from_tid,
      continuation_t continuation)
{
```

```
        arch.do_ipc_continuation = continuation;

        // Trashes the stack pointer on return.
        sys_ipc(to_tid, from_tid);

        // The old stack context is gone!
        ACTIVATE_CONTINUATION(
              get_current_tcb()->arch.do_ipc_continuation);
}
```

Notice how it nobly attempts to avoid accessing the stack after it's gone:

- It recalculates the TCB by using `get_current_tcb()` instead of using the C++ `this` pointer
- It does not access the `continuation` argument

The code looks very convincing but sometimes fails. As `do_ipc()` is inlined, if there was a call to `get_current_tcb()` before `do_ipc()`, the optimising compiler cleverly realises that the `get_current_tcb()` in `do_ipc()` is redundant and can be served by retrieving it from a variable. Now, the variable may have been stored on the stack (in which case it's trashed) or stored in a callee saved register (which has also been trashed).

Even if `do_ipc()` were not inline, the C++ compiler still assumes that the C++ calling convention has been maintained after the call to `sys_ipc()`. It has every right to depend on the old stack context and callee saved registers for whatever reason. As a result, seemingly harmless changes like marking unrelated pointers as `const`, adding assertions, adding debug printing – even rearranging code unrelated to the inlined `sys_ipc()` call (but enough to vary the register allocation) – were enough to cause incorrect behaviour.

These bugs were only caught and worked around – several times and in different ways – because the Blackfin has an MPU that trapped bogus memory accesses after registers were clobbered. With the goal of eventually supporting CPUs without MPUs, such as the ARM7TDMI, which cannot trap these bugs, one wonders what the chance of writing a reliable kernel, on such CPU, is. And this is all in the name of saving a *single argument* in a function call. While it is true that some architectures have complicated function calling conventions, given a choice between reliability and apparently reduced function call complexity, it is obvious that we should choose reliability.

However, without control over the current L4 source base, one can only resort to workarounds and the following one is reliable for the time being (until the compiler changes behaviour etc.): after a call to a function that clobbers the stack and callee saved registers, *immediately* call a function that accepts no arguments and does not return. This function must not be `inline` nor `static` (as `static` functions may be inlined by the compiler). As it accesses no arguments, the compiler makes no assumptions about the stack. As it is a new function, the compiler will not access any callee-saved registers. Do not use `ACTIVATE_CONTINUATION()` to invoke this function as the additional code may be enough to cause bugs.

Summary on mechanisms

The single stack kernel's mechanisms are error-prone.

`ACTIVATE_CONTINUATION()` invokes a continuation but unnecessarily drops the contents of the stack for claimed cache benefits. This causes problems if it is accidentally used on a code path traversed by a trap from kernel mode.

The current UTCB is stored at the top of the kernel stack and is an invitation for it to be overwritten.

System call functions return after violating the calling convention, just to avoid passing a single continuation argument, potentially from assembler. A workaround is to switch to a non-inline function as quickly as possible.

10.2.4 Performance

On the Blackfin, the CPLB footprint of the single stack kernel is identical to the multi-stack kernel as the TCBs have always been physically allocated and the single kernel stack is locked into the CPLB as part of the kernel image. On other architectures with virtually addressed TCBs, the MPU footprint should be reduced as more of the smaller TCBs can fit into a page.

On a switch, the single stack kernel suffers the overhead of saving and restoring continuation state. But in its place, a multi-stack kernel must save and restore registers, to maintain the C calling convention. No general conclusion as to which is slower can be made as it depends on the particular code path and the C calling convention.

However, the main claim to fame of the single stack kernel is reduced cache footprint through the reuse of a single stack, which Warton's benchmarks show more than compensate for this kind of change in macro-benchmarks.

As there are no macrobenchmarks available for L4 on the Blackfin, all we could attempt was the `pingpong` microbenchmark. Attempts were make minimise the number of non-single-stack-kernel related changes between the multi-stack and single stack kernels e.g. disabling of some newly-added CPU-consuming assertions. However, given the number of structural changes indirectly caused by the single stack kernel, it was very difficult to create a fair comparison. And at that time, the kernel was in early development, so the IPC time was meaningless as it was dominated by system call deferral and CPLB refill costs.

Therefore, the cost of entering and exiting the kernel was measured and found to be 4,412.68 cycles, almost 10% slower than the multi-stack kernel. But this should be taken with a grain of salt as again, the number of indirect changes between the kernels was significant and difficult to minimise. And also, a macrobenchmark, had one been available, may have led to a different conclusion.

10.2.5 Implementation summary

The boot stack is a waste of space and should be merged with the single stack. The size of TCB has been reduced from 4KB to just over 600 bytes, thanks to the removal of the kernel stacks, and we believe it would be difficult to reduce the size even further.

The mechanisms in the single stack kernel are error-prone and are not suitable for reliable

kernels.

We experienced difficulty in trying to measure the performance difference due to the state of the kernel and the lack of macrobenchmarks.

10.3 Conclusion

We described the changes necessary to port the single stack kernel:

- execution context is lost on a switch so any code that potentially switches needs to be split at the potential switch point and its state stored in the TCB for recovery by a continuation function
- if trapping from usermode, one needs to switch to the kernel stack after pushing the user context into the TCB.

We implemented the single kernel stack and reduced the size of the TCB from 4KB to just over 600 bytes.

We reflected on our experience and determined that the single stack kernel's mechanisms are error-prone:

- `ACTIVATE_CONTINUATION()` unnecessarily drops the contents of the stack
- the current TCB pointer is stored at a vulnerable position – the top of the kernel stack
- system call functions return after violating the calling convention, just to avoid passing a single continuation argument.

Chapter 11

11 Physically Addressed TCBs

The Blackfin port previously used a static array of 256 TCBs. With the desire to support a larger thread ID space, we changed this to an array of 65536 TCB pointers (each 32-bit) with TCBs allocated on demand. Nourai called this scheme "phys_4b" and found that it gave comparable performance to the other physically-addressed TCB schemes. We chose it because it was the simplest.

In porting the Blackfin to this scheme, we found two important details in his implementation that he did not mention in his thesis report. We now describe those details – namely, how to convert from an address to a TCB pointer and how to handle TCB allocation – to make it possible for others to migrate to the physically-addressed TCB scheme.

11.1 Address to TCB Conversion

The KDB kernel debugger's `print_tid()` function calls `tcb_t::addr_to_tcb()` which returns the TCB an address corresponds to. With a simple array of TCBs (virtual or otherwise) it was easy to test whether an address was pointing to a TCB – check if the address was inside the array. But with an array of TCB pointers, TCBs can be allocated anywhere inside the kernel heap.

The method used by Nourai to check if the address points to a TCB is to access the thread ID field as if the address pointed to a TCB. The thread ID can then index into the TCB pointer array. If the pointer in the array is identical to the candidate address, then the address is a TCB.

11.2 TCB Allocation

In this section, we compare implementation-level TCB allocation issues using virtual and physical TCB arrays. We then build on this and progress to discussing thread allocation in the context of a physical TCB pointer array, which is our target data structure.

11.2.1 Virtual TCB array

In an L4 implementation with a virtual array of TCBs, there is a trick to test if a TCB is valid – access the thread ID field of the TCB and see if it is non-zero. On an access to an invalid TCB, a mapping from the virtual address to the *dummy TCB* is added. The dummy TCB has a zero thread ID and is readonly.

The `tcb_t::allocate()` method initialises a TCB, given a pointer to a virtual TCB. If there is already a mapping from the virtual TCB address to the dummy TCB, it is erased. A mapping is then made from the virtual address to the physical backing for the TCB. Finally, the thread ID field is set appropriately and will be non-zero, denoting a valid TCB.

11.2.2 Physical TCB array

If the TCBs are stored in a physically-addressed array of adjacent TCBs, as the initial Blackfin port did, aside from wastefully reserving memory for non-existent threads and limiting the thread ID space as a result, everything is still straightforward.

As MPU misses never occur on the locked TCBs and as we have no virtual memory on the Blackfin anyway, the dummy TCB is not used. All TCBs are simply zero initialised (i.e. their thread ID fields are zero) to start with so in effect, all TCBs are separate dummy TCBs. There is no need to make them readonly since regardless of the method of storing TCBs, the kernel never writes to TCBs before calling `tcb_t::allocate()` - the readonly marking of the dummy TCB mappings is simply to catch kernel bugs.

And without virtual memory, `tcb_t::allocate()` need not play with mappings and simply initialises the thread ID field to create a new TCB.

11.2.3 Physical TCB pointer array

But if the structure is changed to a physically-addressed array of TCB *pointers*, things become more complicated if we want to keep the current method of checking TCB validity by dereferencing (to reduce the number of changes that have to be made to the kernel). Firstly, all pointers must be initialised to point to a dummy TCB, not NULL, else dereferencing cannot work. Consider the following code fragment:

```
tcb_t *virtual_tcb_address = get_current_space ()->get_tcb (tid);
```

```
// Is "tid" invalid (using TCB deference trick)?
if (tid != virtual_tcb_address->tid)
{
      // "tid" is invalid, a mapping from "virtual_tcb_address" to
      // the dummy TCB was added by the above dereference.
      virtual_tcb_address->allocate ();
}
```

For the virtual TCB array case, the comments reflect exactly what is occurring. For the physical TCB array case, all invalid TCBs act as if they were all separate dummy TCBs i.e. just like the virtual TCB array case, `get_current_space ()->get_tcb (tid)` returns a different address for each different `tid`.

But for the physical *TCB pointer array* case, if `tid` is invalid, `get_current_space ()->get_tcb (tid)` always returns the *same* pointer – a pointer to the dummy TCB. `virtual_tcb_address->allocate()` cannot work because that would modify the dummy TCB. This is a consequence of not having virtual memory – our virtual addresses are not mapped to physical addresses but rather, our virtual addresses are equal to physical addresses and in this case, that address is the dummy TCB for *all* invalid TCBs.

The solution that was used is to drop `virtual_tcb_address->allocate()` and use the following code:

```
virtual_tcb_address = tcb_t::create_new (tid);
```

11.3 Conclusion

We increased the thread ID space from 8-bit to 16-bit by changing from an array of TCBs to Nourai's physically-addressed array of TCB pointers. We described some undocumented, low-level implementations issues that are required for migrating to this scheme.

Chapter 12

12 IPC

In this chapter, we discuss a small set of implementation-specific issues regarding IPC.

We firstly discuss a caveat with the IPC system call and the user context. The rest of the chapter concentrates on our attempts to back some IPC message registers in real CPU registers.

Other IPC-related issues are dealt with in more general chapters, including Protected Kernel Addressing, CPLB Optimisations and Micro-optimisations.

12.1 User Context

Functions implementing system calls in L4 generally modify the user context to return a value to userland.

However, the function implementing IPC cannot safely do this. This is because a CPLB pagefault handler invokes the IPC function in order to contact the faulting thread's pager. The faulting thread's context should not be modified by this IPC function as exceptions are supposed to be transparent (a thread should not know that it has experienced an exception, except for a time difference).

Instead, only the code, that invokes the function that implements IPC, should modify the user context. This is a general problem and was previously undocumented, leading to the Blackfin port corrupting the `R0` register (return value register for the IPC system call) on a CPLB fault.

12.2 Register-Backed Message Registers

In this section, we start by discussing the motivation for register-backing message registers. We then explain why an IDL compiler is essential for taking advantage of the performance improvement provided by registers. Finally, we discuss our attempts to use registers on Blackfin.

12.2.1 In theory

The motivation is that if messages are small enough to fit completely into registers, then only a context switch is required to perform an IPC. No copying to and from memory-backed message registers would be required and therefore, no data cache lines would be touched for the message itself. Apart from saving a few cycles on the copy itself, this cache friendliness is why L4's IPCs can be so lightweight.

On protected kernel addressing architectures without virtual memory, this can also save a protection unit miss and refill on the UTCB [p87, Chapter 7.1].

12.2.2 In practice – the need for an IDL compiler

In this section, we show why `libl4` is insufficient for taking advantage of register-backed message registers and motivate the need for an IDL compiler.

The userspace C++ system call library `libl4` stipulates that messages be sent like this:

```
L4_Msg_t message;

// Construct “message”.
L4_MsgClear (&message);
L4_SetMsgLabel (&message, label);
L4_AppendWord (&message, word_to_be_sent);

// Copy “message” into the message registers.
L4_MsgLoad (&message);

// Send the contents of the message registers to “destination”.
L4_Send (destination);
```

Ideally, `L4_MsgLoad()` would copy as many words as possible into register-backed message registers.

However, `L4_MsgLoad()` actually copies all of the words into the UTCB, stored in memory. `L4_Send()` then reads the words back out of the UTCB and then loads as many as possible into registers before invoking the IPC system call sequence.

This is because `L4_MsgLoad()` cannot write to registers if the C++ calling convention is to be maintained. It cannot simply use callee saved registers because there is no guarantee that the compiler has not decided to use them for storing variables. Marking registers as clobbered does not work as gcc will restore them before the function exits.

Contrary to popular belief [Kuz05], an optimising compiler would not help us here because some words (the ones that should be register-backed) are still needlessly stored into the UTCB by `L4_MsgLoad()` and memory writes cannot be optimised out. These memory writes touch data lines and completely defeat the purpose of having register-backed message registers.

The situation is actually worse than that. If the message is not going to be reused, as is the common case, there is no point constructing the message (`L4_MsgAppendWord()`) in memory and touching data cache lines.

Furthermore, when `L4_Send()` actually copies words out of the UTCB and into registers, the number of words it copies it equal to the number of register-backed message registers, which may be larger than the size of the message. This is because doing a checks such as:

```
if (number of words >= 1)
{
      Copy Word into Register for Message Register #1
      if (number of words >= 2)
      {
            Copy Word into Register for Message Register #2
            if (number of words >= 3)
            {
                  [...]
```

with its conditional jumps would probably exceed the cost of unnecessary copies from the UTCB to registers, especially given that the words it copies are likely to be in the same data cache line. Nevertheless, these unnecessary copies do not just waste CPU but also clobber registers that the compiler will have to spend more time restoring. If there is a shortage of registers, the contents of the to-be-clobbered registers may have to be pushed onto, and later popped off, the stack resulting in more data cache lines being used.

Therefore, the only way to actually exploit L4's register-backed message registers is to handcraft assembler or use an IDL compiler, such as Magpie [Nic06c], to generate such assembler.

12.2.3 Attempts on Blackfin

We attempted to support register-back message registers for the Blackfin port but ran into several issues – GCC, difficulties in trying to be efficient and the realisation that we would not actually gain anything, given the state of gcc and the rest of the port.

GCC bugs

GCC appears to ignore register assignments and clobbering, making it extremely difficult to write code that manages registers correctly.

Ignores register assignments

```
// Assign val to Message Register 0 i.e. register "P2".
register MR0 asm ("P2") = val;
```

GCC ignored this line and simply did not produce any code for it. It should be clear to gcc, because we specified a *specific register*, that the `register` keyword is not just a hint in this situation. Using `asm volatile` did not make a difference.

Ignores register clobbering

Our code was of this form:

```
// Message Register 0 is stored in register "P2" ... [1]
register MR0 asm ("P2");

asm volatile
(
      // ... System call invoking code ...

      : "+r" (MR0)  // System call wrote to MR0 i.e. P2 ... [2]
);

// ... Some C++ code ... [3]

// ... Some C++ code that copies P2 to memory ... [4]
```

Register `P2` was clobbered by the code at [3] even though we have indicated that we are using `P2` for the duration of the function:

1. As we described above in *Ignores register assignments*, because we specified a specific register at [1], the `register` keyword is not just a hint and should not be ignored by gcc.
2. Furthermore, the line at [2] explicitly states that the assembler reads and writes `P2` so the register should not be used by C++ until after the end of the function and should be untouched at [4].

This is evidently a gcc bug.

Efficiency of hand-optimised assembler

Given the above unreliability of C++ code after assembler that had clobbered some registers, we decided to resort to implement `L4_Ipc()` (the back-end to `L4_Send()`) almost entirely in assembler. The idea was that the clobbered registers would be restored by the C++ calling convention when the function exited.

Unfortunately, when the compiler inlined `L4_Ipc()`, the registers stayed clobbered for a longer period of time and were evidently, only restored when the code's parent non-inline function exited. As a result, it reduced the number of available registers to surrounding C++ code – so much so that gcc refused to compile certain C++ segments that invoked `L4_Ipc()` on the

grounds that it had run out of registers for register allocation!

Furthermore, by writing so much code in assembler, we had forced copying from registers containing C++ variables (such as the words to be copied in the UTCBs) to registers we had arbitrarily chosen. Had it been possible to write the code in C++, the compiler would almost certainly have been able to avoid this and provide for a more optimal register allocation. In other words, assembler actually *reduced* performance in this situation.

In any case, given that the registers stayed clobbered for longer than expected and gcc is known to be buggy in this situation, this simply did not work. On top of this, the code was far more complex, being assembler. At this point we decided to give up.

Futility of the exercise

Even if we had succeeded in implementing register-backed message registers (and this was impossible since gcc is buggy), any gain would have been outweighed by the:

1. unnecessary storing of message words to memory in the absence of an IDL compiler
2. the lack of a fastpath – the slowpath still accesses the message words stored in the UTCB, not the registers

Benchmarks of any improvement would only be meaningful if the above two problems were solved.

Summary of attempts on Blackfin

GCC ignored our attempts to mark registers as precious or reserved. Even if could have implemented the fastpath correctly, it would only have been meaningful if we had used an IDL compiler and written a fastpath, neither of which we found time to do.

12.3 Conclusion

Functions implementing system calls in L4 generally modify the user context to return a value to userland. However, the function implementing IPC cannot safely do this.

Register-backed message registers can result in zero-copy IPC and reduce D-cache footprint. Libl4 is insufficient for exploiting register-backed message registers – an IDL compiler is required. GCC is seemingly uncooperative with register specifications and this is one reason why we abandoned implementing register-backing.

Chapter 13

13 Micro-optimisations

This chapter is a collection of techniques that we found to give a very large combined performance boost.

Our micro-optimisations were guided by 2 general principles:

1. **Frequency of Execution**: Certain sections of the kernel, such as the trap code and IPC path, may be executed millions of times a second. A few wasted cycles on these paths add up to a few million wasted cycles per second. Hence, we spent much time optimising these sections of code.

2. **How Fast is Fast?**: A particular section of code – perhaps a loop – may take a millionth of a second to execute or less – faster than a blink of the eye. But on a 500 MHz processor, this is 500 cycles, which is the ballpark speed for an *entire IPC*.

The topics we will cover are assertions, volatile operations, inlining, avoiding jumps, code simplicity, pointer arithmetic and always choosing hardware routines over software ones.

13.1 Assertions

In this section, we discuss disabling assertions in release mode.

There is an undocumented constant called `CONFIG_KDB_NO_ASSERTS` which disables all assertions. By disabling this unnecessary code, the time to perform a trap into and out of the kernel more than halved – it fell from 1970.41 cycles to 898.24 cycles. The size of the kernel decreased by a similar amount.

With increasing numbers of consistency checks in the kernel, the importance of disabling assertions becomes ever so more increasing.

13.2 Volatile Operations

The C `volatile` keyword is useful for preventing the compiler from optimising. However, careless use of this can result in suboptimal performance. In this section, we provide 2 examples – an `SSYNC` pipeline flush and reading registers.

13.2.1 SSYNC pipeline flush

Reads to some system memory-mapped registers need to be preceded by an instruction barrier to resolve any speculative states in the pipeline. This is normally done using an `SSYNC` pipeline flush, which takes a minimum of 10 cycles. So a typical code sequence might be:

```
asm volatile (“ssync;”);
ASSERT (NORMAL, *memory_mapped_register == 0x1234);
```

In debug mode, we ordinarily require the `volatile` as gcc does not understand instruction side effects. This prevents gcc from:

1. simply optimising out the instruction because “it doesn't affect any variables”
2. moving the barrier instruction to somewhere else

However, in release mode, the `ASSERT` is not executed at all. Therefore, the `SSYNC` is not required either and is merely wasting cycles. Therefore, it should be marked for conditional compilation using `#if !defined (CONFIG_KDB_NO_ASSERTS)`.

13.2.2 Register reads

It's a similar story for ordinary register reads as well, where the `volatile` is needed because the register read must occur at a particular location in the code:

```
u32_t val;
asm volatile (“%0 = retx;” : “=r” (val));
ASSERT (NORMAL, val != 0xabadcafe);
```

In this case, assigning the value of the `retx` register to `val` should not be done in release mode, when assertions are disabled. We now consider caching of `volatile` register reads.

Caching Volatile Reads

Sometimes we do not want to remove a `volatile` statement but might still be executing it more often than necessary.

For instance, we used to have a function `l4bfin_interrupt_level()` that read the `IPEND` memory mapped register in a loop:

```
for (int i = 0; i < 16; i++)
{
      if (*BFIN_IPEND & (1 << i))  // Memory Mapped Register
      {
            // Do something.
      }
}
```

The `BFIN_IPEND` pointer was marked as `volatile` because it is changed by the processor on traps and untraps. As a result, the compiler ensured that it would be reloaded – at the cost of 4 cycles – on every iteration of the loop, instead of simply caching it in a register like a normal variable. But we knew that the `IPEND` would not change inside the loop so by simply moving it out, we saved 60 cycles (`4 * 16 – 4 = 60`, where the `– 4` is because we still need to load `IPEND` the first time):

```
const u32_t ipend = *BFIN_IPEND;  // Memory Mapped Register
for (int i = 0; i < 16; i++)
{
      if (ipend & (1 << i))
      {
            // Do something.
      }
}
```

As it was a very commonly called function, these wasted cycles alone was greater the cost of an entire IPC on other architectures!

13.3 Inlining Functions

In this section, we discuss how carefully inlining some functions, instead of calling them, can result in dramatic performance improvements.

Recall that the direct cost of a call to an empty function is at least 18 cycles [p17, Chapter 2.1]. Indirect costs include the C calling convention which forces:

1. Marshalling of arguments into the correct registers.
2. Saving or avoidance of registers that may be clobbered by the call function. With only 10 registers (excluding the stack and instruction pointers) preserved by the Blackfin gcc calling convention, more than 30 registers cannot be relied upon after a function call.

We can avoid these costs through inlining. An additional advantage of inlining is that breaking the function call barrier enables more optimisations. An example is this:

```
void f (bool input)
{
      if (input)
            // ... do this ...
      else
            // ... do that ...
}

void g (void)
{
      f (true);
}

void h (void)
{
      f (false);
}
```

`f` is not inlined so the `if input` check is always performed. However, if we marked `f` as `inline` (or `static`), the check can be optimised out because the compiler knows which way the branch is going to go. There is actually a gcc inefficiency here in that even though the inlined `if` is optimised out in `g()`, `true` is still placed into the `R0` register before the inlined test that is actually no longer compiled in. Similarly, in `h()`, `false` is placed into a register before the missing branch.

Of course, inlining functions has the following disadvantages from both a performance and maintenance point of view:

1. If the function is large enough and/or called frequently enough, the size of the kernel will increase and furthermore, increases in instruction cache capacity misses (which cost around 100 cycles each [p24, Chapter 2.3.2]) may outweigh any savings in function calling convention overheads.
2. Changes to inline functions require time-consuming recompilation of all files that call the function. Inline functions in libraries that maintain ABI compatibility cannot change

(but this is not a problem with the kernel).

3. Inline functions (which are implemented in .h headers) generally require inclusion of other headers to perform work. Header dependencies may become circular and require re-engineering.

So a general rule is to only inline small functions (as a rule of rough, several C statements as long as they do not call non-inline functions) that are called frequently on any critical path.

For instance, L4's generic page table walker (`space_t::lookup_mapping()`) calls our `pgent_t::is_subtree()` method up to 4 times for the page table implementation (which is a 2 level page table that is essentially 4 levels deep [p65, Chapter 4.5.4]). Our pgent_t implementation looks like this:

```
bool is_subtree (space_t *s, pgsize_e pgsize)
{
      return is_tree (s, pgsize);
}

bool is_tree (space_t *s, pgsize_e pgsize)
{
      bool ret;

      if (pgsize == size_4m)
            ret = tree.is_tree;
      else
            ret = ((pgsize_e) leaf.pgsize < pgsize);

      return ret;
}
```

As `is_subtree()` (which is called 4 times) calls `is_tree()` each time, there are potentially 8 function calls for each call to `space_t::lookup_mapping()`, which in itself is called on the critical CPLB refill path. The direct cost is `8 * 18 = 144` cycles lost to function call overhead alone! And then there are the indirect costs with argument marshalling and register clobbering. As these functions are small, there is no performance benefit in making them full functions. And these are not the only functions being called on the refill path.

We inlined many such commonly called functions. Note that the following presented cycle counts are quite high as they represent the kernel at different, early stages of development. Here are examples of the performance improvements we got *for free*:

Patch	*Main Inlined Functions*	*Benchmark*	*Before (cycles)*	*After (cycles)*	*Saving (cycles)*
d155	* Trapframe accessors * Current interrupt level * Whether we came from kernel mode * Exception type accessor	Trap in and out of kernel	894.24	651.15	243.09
d163	* Page table entry accessors (above text – the "pgent_t" example)	DCPLB refill	4130.30	3269.93	860.37
r34	* TCB accessor * CPLB flush functions * CPLB tactic function (i.e. should I flush multiple CPLB entries of the whole CPLB)	DCPLB refill	2805.98	2767.77	38.21

Table 10: Performance benefits of inlining functions

Notice how we managed to reduce the DCPLB refill cost by close to 900 cycles (out of more than 4000) simply by inlining!

13.4 Avoid Jumps

Pipelined architectures begin executing future instructions to keep the pipeline full. Unfortunately, jumps create problems for this scheme and so trying to avoid jumps will be the focus of this section.

The target of a computed jump cannot be determined until the computation has been completed so the inability to fetch ahead results in pipeline stalls. Similarly, the processor cannot know whether conditional branches will be taken or not so does not know which set of future instructions to fetch in advance - we later discuss the solution, branch prediction, in depth [p147, Chapter 13.4.5].

The following are the minimum costs for branches and assume no I-cache misses (note that they are all 1 cycle more than implied by the CPU manual [Ana05]):

Operation	*Cycles*
Unconditional Jump	5
User Exception (`EXCPT`) i.e. system calls	10
Return from Exception (`RTX`) [*]	5
Conditional Jump Mispredicted	9
Conditional Jump Predicted	4
Conditional Jump Not Taken, Predicted	1

Table 11: Blackfin jump and branch prediction costs

Note that the conditional jump costs exclude the cost of the tests and any register loading required. Branches are statically predicted by the programmer using the assembler `(bp)` suffix.

So our mission is to firstly, avoid jumps but if that is not possible, attempt to optimise the common case by prediction. We will discuss 5 optimisations methods related to branches. We later also discuss loop unrolling as another way to avoid jumps [p162, Chapter 14.2.1].

13.4.1 Do not allow special cases

Testing for special cases on a commonly executed path is a waste of cycles.

For example, the L4 boot process fakes an initial context switch in order to initialise some state and establish the current TCB pointer. This initial switch is indicated by a NULL `old_tcb` pointer. So instead of this test:

```
if (old_tcb->get_space () != new_tcb->get_space ())
```

we needed an extra test (and potentially, a conditional jump depending on how it was

[*] This figure is derived because we could not measure it directly – `EXCPT` cost 10 cycles and `EXCPT`, followed by `RTX`, cost 15 cycles so we concluded that `RTX` takes `15 - 10 = 5` cycles.

implemented by the compiler):

```
if (!old_tcb || old_tcb->get_space () != new_tcb->get_space ())
```

There were several other checks and complexities for the initial switch. But in *all* other uses of this context switching function, `old_tcb` is never NULL. This check is needlessly being executed in the common case. The solution was to split this context function into 2 – one function for the initial switch and one function for the common case.

13.4.2 Do not provide optional return arguments

Optional return arguments are also a performance hit because of extra checking.

`l4bfin_interrupt_level()` was once a commonly called function. It returned the current interrupt level but also, optionally returned whether we had trapped from kernel mode:

```
int l4bfin_interrupt_level (bool *was_in_kernel_mode = 0)
{
      // ... Some calculations ...

      if (was_in_kernel_mode)
            *was_in_kernel_mode = [expression];

      return [expression];
}
```

Notice that we are checking if `was_in_kernel_mode` points to NULL, in case the user of the function does not care about the relevant value. This is a common C++ API idiom but is woefully inefficient. In the common case, `was_in_kernel_mode` is non-NULL so the overhead of this check is added unnecessarily. There are 2 solutions:

1. Split the function into two

We can split the function into two to handle each situation – one where the caller cares about the value in question and the other, does not even calculate the value in question. In this way, we have pushed the "do we care about this value" check from runtime to compile-time. This technique has to be used with caution as it risks increasing I-cache capacity misses if the function is large enough.

2. Make argument compulsory

If the caller guarantees that `was_in_kernel_mode` always points to a valid variable, then we save a jump by never needing to check that it points to non-NULL. However, the callers who do not care about the return value will still be forced to have the value written somewhere in memory, potentially touching another D-cache line. But if the function is inlined, there is a hope that the compiler will realise that the value is not used and not even bother calculating it.

13.4.3 Do not generalise

Generalising code is sometimes a bad idea as we show with this example. As the code for flushing the DCPLB was identical to the respective ICPLB code, except for some pointers and the way in which the respective protection unit was disabled and enabled, we thought that the *Extract Method* refactoring would be a good way to reduce code duplication:

```
// Prototype for our back-end containing the actual code for flushing
// [DI]CPLBs.  Notice that that all arguments except "space" are
// different for flushing a DCPLB as opposed to an ICPLB.  This is
// where all the shared code is implemented.
inline
void
l4bfin_cplb_flush_unlocked_entries (space_t *space,
        volatile const u32_t *addr_regs, volatile u32_t *data_regs,
        int num_unlocked_regs,
        void (*disable_func) (void), void (*enable_func) (void));

// Front-end function for flushing the ICPLB.  The DCPLB function
// is identical except the I's change to D's.
inline
void
l4bfin_icplb_flush_unlocked_entries (space_t *space)
{
       l4bfin_cplb_flush_unlocked_entries (space,
               BFIN_ICPLB_ADDR, BFIN_ICPLB_DATA,
               L4BFIN_ICPLB_NUM_UNLOCKED_ENTRIES,
               l4bfin_icplb_disable, l4bfin_icplb_enable);
}
```

However, the use of function pointers (`disable_func` and `enable_func`) for passing the functions for disabling and enabling the respective CPLB (e.g. `l4bfin_icplb_disable()` and `l4bfin_icplb_enable()`) was sufficiently complex enough to prevent gcc from inlining:

1. these flushing functions (`l4bfin_cplb_flush_unlocked_entries()` and `l4bfin_[di]cplb_flush_unlocked_entries()`)
2. the calls to the disabling/enabling functions passed as function pointers (`l4bfin_[di]cplb_disable()` and `l4bfin_[di]cplb_enable()`)

What's worse is that these are called on the address space switching path with performance is critical.

Given how short the functions were, it made better performance sense to simply duplicate the code of `l4bfin_cplb_flush_unlocked_entries()` in implementing the two functions, `l4bfin_[di]cplb_flush_unlocked_entries()`. The reduced abstraction also benefited readability.

We conclude that duplicating small sections of code is simpler and more efficient than attempting to extract the shared code into a generalised function and forcing branches.

13.4.4 Use sensible data structure invariants

Assumptions about data structures can save us from performing unnecessary tests.

Previously, the code for flushing the CPLBs on an address space could not simply set all the CPLB entries to 0 – it needed to check, for every entry, that it was not flushing a locked entry (e.g. for a kernel superpage).

We could simply avoid these tests and conditional jumps, for each of the 16 entries of the 2 CPLBs, by using data structure invariants. As the locked entries are determined at boot time and are never changed, we simply place all locked entries in the last few entries of each CPLB and our loops will be set to only go up to a predefined constant, containing the number of unlocked entries. No checks nor conditional jumps will need to be made.

13.4.5 Use branch prediction

As we stated previously, the processor does not know which way a conditional branch will be taken, in advance. The approach taken by Blackfin is for the programmer to provide static prediction hints. The assembler is annotated to specify whether the branch is expected to be taken. If this is correct, it permits instruction readahead but if it is incorrect, the pipeline must be flushed of the instructions that came from the branch that was not actually taken.

By default, the compiler specifies the branch prediction bit by using well-known results from code analysis. The conditional jump at the end of each loop is always predicted to be taken, as it is expected that most loops execute for several iterations.

All programmer specified `if`'s are assumed to not be taken as they are expected to cover special cases. As a result, simply inverting the check and swapping code between the `if` and else clauses is sufficient to affect performance. The `if` and `else` clauses may have been written in a particular order for readability reasons. Therefore, gcc provides the `__builtin_expect()` compiler directive to allow the programmer to specify branch prediction. We should also specify the branch prediction bit when writing assembler manually.

We must be careful to specify the correct prediction as a mispredicted branch costs 9 cycles. Furthermore, the common path should not branch so that the processor can fetch the instructions without waiting for the jump address to be computed: a predicted untaken branch costs 1 cycle – the same as a basic instruction – but a predicted taken branch costs 5 cycles.

13.5 Simplicity

For frequently called functions such as `get_current_tcb()` and previously, `l4bfin_interrupt_level()`, we should make them as simple as possible so that they execute in as a few cycles as possible and so that inlining does not bloat the I-cache footprint. Recall that `l4bfin_interrupt_level()` used to loop 16 times and perform bit manipulations when it could be written to simply read from a cacheable variable [p52, Chapter 4.2.3].

13.6 In Loops, Use Pointer Arithmetic - Not Array Indexing

We initially found that it cost at least 15 cycles to copy each message register (a single 32-bit word). We found that this figure was due to the compiler continually performing array subscript calculations:

```
// 404c86:       17 32           P2=R7;
// 404c88:       9d ac           P5=[P3+0x8];
// 404c8a:       91 ac           P1=[P2+0x8];
// 404c8c:       42 44           P2=P0<<2;
// 404c8e:       4a 5a           P1=P2+P1;
// 404c90:       08 6c           P0+=0x1;
// 404c92:       08 e4 10 00     R0=[P1+0x40];
// 404c96:       aa 5a           P2=P2+P5;
// 404c98:       44 0a           CC=P4<=P0(IU);
// 404c9a:       10 e6 10 00     [P2+0x40]=R0;
// 404c9e:       f4 17           IF ! CC JUMP 404c86
for (word_t i = start; i < start + count; i++)
      dest->utcb->mr [i] = this->utcb->mr [i];
```

Simply re-expressing this indexed array loop into a pointer loop [p161, Chapter 14.2.1] reduced the number of cycles per message register down to 8:

```
const word_t *src_mr = this->utcb->mr + start;
const word_t *src_mr_end = src_mr + count;

word_t *dest_mr = dest->utcb->mr + start;

// 404c90:       10 90           R0=[P2++];
// 404c92:       08 92           [P1++]=R0;
// 404c94:       42 08           CC=P2==P0;
// 404c96:       fd 17           IF ! CC JUMP 404c90
while (src_mr != src_mr_end)
      *dest_mr++ = *src_mr++;
```

Putting this into perspective, the cost of copying 60 message registers has dropped from `60 * 15 = 900` to `60 * 8 = 480` cycles – a saving of more than 400 cycles, when a 0MR Inter-AS IPC costs 1567.86 cycles [p177, Chapter 17.1].

13.7 Avoid Software-Based Primitives

Software routines can be very slow. As our motivating example, when optimising our DCPLB refill path, we found that an innocent looking statement for calculating the next index in the FIFO replacement policy took more than *250 cycles*:

```
upto = (upto + 1) % num_unlocked_regs;
```

Instead, we defied our wisdom of avoiding jumps and found the following, less straightforward code executed in less than 10 cycles:

```
upto++;
if (EXPECT_FALSE (upto >= num_unlocked_regs))
      upto = 0;
```

The reason for this anomaly is that the Blackfin does not support divide nor modulus with a simple instruction. Instead, `libgcc` implements modulus as a function, `___modsi3`:

```
    17c8:       00 e3 70 09     CALL  2aa8 <___umodsi3>;

00002ab4 <___modsi3>:
    2ab4:       67 01           [--SP] = RETS;
    2ab6:       08 32           P1=R0;
    2ab8:       11 32           P2=R1;
    2aba:       ff e3 d3 ff     CALL  2a60 <___divsi3>;
    2abe:       49 30           R1=P1;
    2ac0:       52 30           R2=P2;
    2ac2:       c2 40           R2*=R0;
    2ac4:       11 52           R0=R1-R2;
    2ac6:       27 01           RETS = [SP++];
    2ac8:       10 00           RTS;
        ...

00002a60 <___divsi3>:
    2a60:       67 01           [--SP] = RETS;
    2a62:       47 01           [--SP] = R7;
    2a64:       82 43           R2=-R0;
    2a66:       80 0c           CC=R0<0x0;
    2a68:       02 07           IF CC R0 = R2;
    2a6a:       07 02           R7=CC;
    2a6c:       8a 43           R2=-R1;
    2a6e:       81 0c           CC=R1<0x0;
    2a70:       0a 07           IF CC R1 = R2;
    2a72:       02 02           R2=CC;
    2a74:       d7 59           R7=R7^R2;
    2a76:       00 e3 09 00     CALL  2a88 <___udivsi3>;
    2a7a:       0f 02           CC=R7;
    2a7c:       81 43           R1=-R0;
    2a7e:       01 07           IF CC R0 = R1;
    2a80:       37 90           R7=[SP++];
    2a82:       27 01           RETS = [SP++];
    2a84:       10 00           RTS;
        ...
00002a88 <___udivsi3>:
    2a88:       00 69           P0=0x20;
```

```
    2a8a:       a3 e0 09 00     LSETUP(2a90 <___udivsi3+0x8>,2a9c
<___udivsi3+0x14>)LC0=P0;
    2a8e:       03 60           R3=0x0(x);
    2a90:       82 c6 08 c0     R0= ROT R0 BY 0x1;
    2a94:       82 c6 0b c6     R3= ROT R3 BY 0x1;
    2a98:       8b 52           R2=R3-R1;
    2a9a:       8b 09           CC=R3<R1(IU);
    2a9c:       1a 06           IF ! CC R3 = R2;
    2a9e:       82 c6 08 c0     R0= ROT R0 BY 0x1;
    2aa2:       c0 43           R0=~R0;
    2aa4:       10 00           RTS;
        ...
```

To perform modulus, 3 non-inlined functions are called (the `CALL`'s) with additional processing on top. Finally, the deepest function in the call chain, `___udivsi3`, performs a loop (`LSETUP` creates a hardware loop) that executes `0x20 = 32` times.

So in summary, avoid modulus when using gcc on the Blackfin and in general, find ways of restructuring code to avoid software routines.

13.8 Conclusion

It is worth optimising code at an extreme level if it is invoked very frequently like the trap and IPC paths. In this chapter we described a range of effective techniques for this purpose:

- Disabling assertions more than doubled performance for free, on an ordinary kernel
- The C `volatile` keyword disables optimisation so we should minimise, or carefully structure, its usage
- Inlining small, commonly called functions gave us visible performance differences for free by avoiding the function call convention
- Jumps are hazards for the pipeline – we should make the flow of execution are linear as possible:
  - Special case tests should never been executed in the common path
  - Sometimes, it is better to duplicate code to avoid jumps, especially if it is short
  - Data structure invariants can remove unnecessary conditional jumps
  - Branch prediction can be used in the cases where we cannot avoid jumps
- Commonly-called functions should be simple so that they are fast and can be inlined, making them even faster again
- We should avoid array subscripts in loops and use pointer arithmetic instead; GCC is not smart enough to convert the former into the latter
- We should avoid software routines at all costs, such as modulus on Blackfin

Chapter 14

14 CPLB Optimisations

This chapter is concerned with implementation issues – specifically, we will run experiments for optimising CPLB protection unit related activities.

We firstly compare ways of switching address spaces, without leaking protection.

We then look at the fastest way to change the protection entries of an entire CPLB. The complication is that changing data protection must be done with data caching disabled and instruction protection changes require instruction caching to be disabled. So the trade-offs for working for the DCPLB do not apply to the ICPLB and vice-versa.

14.1 Switching Address Spaces

For, an untagged protection unit (i.e. with no address space identifiers), when switching to a different address space, one normally flushes all unlocked entries. However, as soon as we return to userspace after the switch, we are guaranteed to experience an ICPLB miss for loading the next instruction. We then almost certainly expect DCPLB misses on data accesses to the stack and global variables.

Therefore, we investigate saving and restoring the ICPLB and DCPLB, to and from the address space data structure, on switches. This would eliminate CPLB misses on context switches entirely.

We describe the methods we will be comparing and then present performance benchmarks, followed by an analysis.

14.1.1 Methods

We describe 3 methods for switching address spaces:

Method 1. Flush CPLBs

With this method, we zero out both the instruction and data CPLBs to prevent protection leaking to the new address space. This is the orthodox address switching method for untagged MPUs, such as Blackfin's CPLBs. It is also very easy to implement.

The direct cost is the zeroing. The high, indirect costs are the compulsory user CPLB misses after the switch.

Method 2. Save and restore CPLBs

We can avoid these compulsory CPLB misses, on an address space switch. Firstly, we save the contents of the CPLBs to the kernel data structure for the address space being switched away from. Secondly, we then restore the contents of the CPLB from the kernel data structure for the address space being switched to. Therefore, address spaces maintain their CPLB entries even after a switch.

The direct cycle costs (measured later) are greater than with the previous method because we are not just zeroing CPLB entries anymore – we are them copying to and from memory. We are using the following amount of extra memory for storing CPLB entries:

`2 * 16 * 8 = 256` bytes per address space

where:

`2` = number of protection units (DCPLB and ICPLB)

`16` = number of entries per protection unit [*]
`8` = entry size (address word and attributes word)

This extra use of memory translates to touching `256 / 32 = 8` D-cache lines for each of the old and new address spaces. Of course, the increased code complexity and size also results in a greater I-cache footprint. However, we expect that all this will be easily outweighed by the elimination of compulsory CPLB misses, after an address space switch.

Method 3. Save CPLBs on refills

We wish to avoid the overhead of saving the contents of the CPLBs on each address space switch. If every time we updated a CPLB, we updated its cached value stored in the address space data structure, we could avoid saving any CPLB entries on a switch. Therefore, we would only need to *restore* the CPLBs for the address space being switched to, on an address space switch.

Over the previous method, we decrease the cost of a switch but increase the CPLB refill overhead.

14.1.2 Performance results

We ran the pingpong microbenchmark due to lack of available macrobenchmarks [p48, Chapter 4.1]. We compared the 3 address space switching strategies in terms of DCPLB refill and IPC performance.

Address Space Switching Strategy	*DCPLB Refill (cycles)*	*Intra-AS IPC (cycles)*	*Inter-AS IPC (cycles)*
1. Flush CPLBs	1,405.5	911.20	12,228.24
2. Save and Restore CPLBs	1,400.8	910.80	1,782.41
3. Save CPLBs on Refills	1,425.26	910.71	**1,567.30**

Table 12: Address space switching strategies

The DCPLB refill cost is largely unchanged across all 3 strategies and is in the range of noise e.g. the differing placement of the kernel code creating a slightly more expensive cache conflict refill cost. We later show that even if we could optimise the DCPLB refill path aggressively, Flush CPLBs would still not be fast enough to outperform Save CPLBs on Refills.

The Intra-AS IPC column shows that the baseline overhead of an IPC. Notice that it is virtually the same across all 3 address space switching strategies as it does not involve any address space switching. The small differences (a cycle at most) are noise. This figure provides the lower bound for the Inter-AS IPC cost.

The difference between the Intra-AS IPC and Inter-AS IPC is composed of the direct costs (e.g. flushing the CPLBs) and indirect costs (e.g. CPLB misses) of the address space switch.

[*] Strictly speaking, we only actually need to store 15 since both the DCPLB and ICPLB lock 1 entry each, for the kernel superpage, and they never modify this entry.

Notice that the Inter-AS IPC cost for a CPLB flush strategy is an order magnitude higher than the other 2 methods.

We urge caution in interpreting these results due to the unrealistically, small working set of pingpong.

14.1.3 Performance analysis

We now examine the performance figures, we just presented, in detail.

Method 1. Flush CPLBs

We previously noted that there were 3 DCPLB misses per Inter-AS IPC [p57, Chapter 4.4]. Things have gotten worse since then because of the introduction of ICPLB protection and additional misses due to an unlocked KIP. The cost breakdown is as follows:

Item	*Approximate Cost (cycles)*
Common Intra-IPC and Inter-IPC Cost	911.2
0. Flushing the CPLBs (the only direct cost)	200 * 2 CPLBs = 400 [*]
1. Kernel-mode data miss on destination UTCB message registers	1,405.5
2. Kernel-mode data miss on KIP for setting the UTCB pointer to the destination thread [#]	1,405.5
3. Usermode instruction miss on the KIP, when returning from the system call wrapper [#]	1,200 [*]
4. Usermode instruction miss on user code	1,200 [*]
5. Usermode data miss on global variables	1,405.5
6. Usermode data protection violation on KIP for accessing UTCB pointer of current thread (as 2. only gave the data page kernel permissions) [#]	1,500 [*]
7. Usermode data miss on UTCB message registers tag [A]	1,405.5
8. Usermode data miss on stack	1,405.5
Total	12,238.7

Table 13: CPLB misses after an address space switch (and flushed CPLBs)

The estimated total of 12,238.7 is very similar to the measured cost of 12,228.24 so we can assume our cost breakdown is sound.

[*] We did not get time in the thesis to run systematic benchmarks for these values so the numbers shown are from rough benchmarks.
[#] Could be avoided by locking or on each switch, pre-filling KIP page into the DCPLB and ICPLB.
[A] Why this did not show up in the DCPLB misses before the thesis began requires further investigation.

As one can see, the direct cost of a context switch is only 400 cycles. However, the indirect costs of a flushed CPLBs – CPLB refill misses – is 10,927.5, almost 12 times the inherent cost of an IPC!

Perhaps our slow C++ CPLB refill path, which takes 1,405.5 cycles to execute, is to blame? Now if we were to avoid all CPLB misses on the KIP, by pre-filling [p89, Chapter 7.2.2] (steps 2, 3 and 6), and all UTCBs misses, by using a global UTCB array [p93, Chapter 7.3.3] (steps 1 and 7), we would be left with just 3 misses (steps 4, 5 and 8). Then, if we could develop a fast enough CPLB refill assembler path, this solution may still be feasible as long as:

```
911.2 + 400 + 100 + 3x < 1,567.30
x = 52.03
```

where:

`911.2` = IPC cost
`400` = CPLB flush cost
`100` = UTCB scheme trap and untrap cost [p93, Chapter 7.3.3]
`3` = minimum number of CPLB misses possible with a flush
`x` = maximum cost of our optimised CPLB refill path to make flushing feasible
`1567.30` = cost of fastest address space switching method identified (Save CPLBs on refills)

The calculation shows that our DCPLB refill path we have to be optimised down to 52.03 cycles, which is unattainable as changing the protection on Blackfin alone takes 50 cycles [p93, Chapter 7.3.3].

Therefore, flushing CPLBs, on address space switches, on protected kernel addressing architectures with software loaded CPLBs, is infeasible due to the high indirect CPLB refill costs, even if we managed to rewrite the CPLB refill path in assembler. But, this may well be feasible on the ARM1156T2-S, which is similar to the Blackfin, except that it has faster hardware-refilled protection units.

Method 2. Save and restore CPLBs

As we can see, this is far faster than flushing the CPLB due to the elimination of CPLB misses. The cost of saving and restoring both CPLBs is `1782.41 – 910.80 = 871.61` in total. Given that merely restoring both CPLBs costs `270 + 207 = 477` cycles [p163, Chapter 14.2.2], this seems reasonable.

Method 3. Save CPLBs on refills

Both the CPLB refill and IPC paths are critical to performance so increasing one at the expense of the other requires justification. We note that the CPLB refill overhead only increases slightly by `1425.26 – 1400.8 = 24.46` cycles, as compared to the previous method. But this difference is quite close to the noise level and is therefore an acceptable penalty since we reduce the context switching time from 1782.41 cycles to 1567.30 cycles. `1567.30 – 910.71 = 656.59` is therefore the time spent switching address space (and restoring CPLBs) and can be considered as the architecture lower bound on context switch performance.

14.1.4 Experimental findings

On address space switches, flushing the CPLBs results in very high indirect costs, due to compulsory CPLB refills. Saving and restoring CPLBs is faster by an order of magnitude. Even if the CPLB refill path was highly optimised, flushing the CPLBs could not be faster than saving and restoring CPLBs.

14.1.5 Restoring CPLBs creates problems

The last 2 address space switching methods saved cycles by avoiding CPLB misses by restoring saved CPLB entries. These entries were saved in address space data structures. What we did not mention is that this actually introduces a major problem in an L4 optimisation for page fault handling. In this section, we describe the problem followed by 2 possible solutions.

The problem - page fault optimisation

On a page fault (i.e. the page is not in the page table, let alone the CPLB), L4 sends an IPC to the faulting thread's pager. The pager is expected to use the `L4_MapControl()` system call to map a page into the faulting thread's address space's page table. It then returns an IPC to the faulting thread.

The kernel could now return to userspace and allow the CPLB miss on the same address to occur again. The kernel would then be re-entered, the CPLB refill path re-executed but as this time the page is in the page table, the refill would succeed.

However, most kernel ports avoid this unnecessary extra trap (almost 400 cycles on the final Blackfin port [p177, Chapter 17.1]), by refilling the CPLB immediately using the new page table entry before returning to userspace. However, if we restore CPLB entries on address space switches, this refill may have already been performed by another thread in the address space, leading to complications. Consider the following sequence of events where Thread A & B are in the same Address Space S and Pager Thread P is in a different address space.

1. Thread A experiences a DCPLB miss on page 0x1234. The kernel is entered, does not find the entry in Address Space S' page table and IPCs Pager Thread P.
2. Pager Thread P uses `L4_MapControl()` to add page 0x1234 to Address Space S' page table.
3. *A timer interrupt interrupts the pager*. The scheduler invokes Thread B.
4. Thread B also experiences a DCPLB miss on the same page 0x1234. The kernel is entered, sees the page in the page table and performs the DCPLB refill. *We add page 0x1234 to the current Address Space S' saved DCPLB entries*.
5. Eventually, the scheduler returns control to Pager Thread P.
6. Pager Thread P returns an IPC to Thread A, the original faulting thread.

7. On the address space switch from Pager Thread P's space to Thread A's Address Space S, we restore the saved CPLB entries from Address Space S. *This includes page 0x1234 due to Step 4.* What would normally happen here is that the kernel completes the resolution of the original Thread A pagefault by refilling the CPLB by looking up the page table. However, because of Step 4, *page 0x1234 is already in the CPLB*.

If page 0x1234 *might* already be in the CPLB due to this problem, we need to probe the entire CPLB to prevent the addition of a duplicate entry into the CPLB. A probe is slow because it involves a loop read the 16 memory-mapped registers in a CPLB (in the order of a hundred cycles). However, it is still faster than attempting to bypass the potential problem by not performing the refill on pagefaults and returning to usermode straight away because in the common case, a miss will occur on the same address and waste time trapping straight back into the kernel.

So we wish to have a more efficient means of determining if a page has already been inserted into the CPLB by another thread, in the same address space, than probing.

Solutions

In this section, we present 2 solutions for solving this pagefault optimisation problem.

L4_MapControl() updating saved CPLBs

We could bypass the problem by changing `L4_MapControl()` to add the page to an address space's saved CPLB entries, so that when we switch back to that address space, that new CPLB entry is loaded. As `L4_MapControl()` now effectively performs a CPLB insertion, there is no need for the kernel to perform the troublesome pagefault optimisation in question (i.e. refilling the CPLB after an IPC to the pager, to avoid an unnecessary trap back in).

This would work on a system with a unified data and instruction CPLB. However, with Blackfin's split data and instruction CPLBs, it is unclear which CPLB `L4_MapControl()` should insert into, for pages with both data and instruction permissions.

Inserting into both CPLBs may be suboptimal if, for example, the page is currently only being used for instructions and not in a data context until a long while later, as it would replace a DCPLB entry that is likely to be currently in use. Therefore, the `L4_MapControl()` API could be modified to indicate whether the the page is being inserted for data, instructions or both.

A further problem is that if a pager were to map a large number of pages into an address space using one or more calls to `L4_MapControl()` - more than can fit into the 16 entry CPLBs – time would be wasted inserting certain pages into the CPLB as they would be evicted by others before they have even been used.

Refill code marking page as “in CPLB”

Let us consider a different solution where the kernel still attempts the L4 pagefault optimisation of refilling after the IPC to the pager returns.

Every time we insert a page into a CPLB, we modify the inserted page's page table entry (which we have a pointer to, after a page table lookup) to indicate that the page is already in the ICPLB and/or DCPLB, by storing 2 bits.

Now when the kernel attempts the refill after the IPC to the pager, it can determine whether the page is already in a CPLB simply by analysing the page table entry, which it would have anyway from the page table lookup.

Of course, if one's page table entries are already full, it would be quite wasteful to increase the size of the entries just to accommodate these 2 bits. Furthermore, on architectures whose hardware mandates a particular page table format, this would also require extra memory for a shadow page table. Remember that our ultimate goal was to avoid CPLB probing on the pagefault path. However, the pagefault path is not executed commonly enough to justify extra memory usage.

As a final implementation detail, if we wish to invalidate a CPLB, we would like to be able to find its page table entry, without a lookup, to clear our “in CPLB” bits. To allow this, we simplify store page table entry pointers when we set save CPLBs in address space data structures.

Conclusions on CPLB restoration problems

After a user pager returns control to the kernel to resolve a pagefault, the kernel normally refills the CPLB by looking up the new page table entry. However, with an unfortunate sequence of events, restoring CPLBs on address space switches may lead to the refill being unexpectedly performed by another thread in the address space.

We could probe the CPLB to check for this situation but this slow. We could simply not do the refill and return to usermode. The processor will cause an immediate trap back into the kernel, in the common case where the refill still needs to be performed. This trap is unnecessary overhead.

Another approach is to essentially fill the CPLBs when a page is mapped into the page table and remove the need for a refill. Without API changes, this does not work well for split instruction and data protection units.

Yet another approach is to store bits, in the page table entries, indicating whether a particular page is in a particular CPLB. There is a concern that this might waste a lot of memory on enlarged page table entries or the need for a shadow page table.

14.2 Restoring CPLBs Without Caching

Having decided, in the previous section, to restore CPLBs from memory instead of flushing the CPLBs on an address space switch, we compare the performance of different methods for restoring the contents of the Data CPLB and Instruction CPLB protection units. Recall that each unit contains 16 entries but only 15 of them are unlocked and need to be restored.

In order to add a page to a CPLB, or modify an existing page's protection, one must disable the relevant protection unit [p29, Chapter 2.3.3]. Therefore, disabling the data protection unit disables data caching and similarly, disabling instruction protection disables instruction caching.

This section will show that the DCPLB must be manipulated in a different way to the ICPLB for optimal performance. We expect that the DCPLB restore code will be sensitive to data accesses as data cache is disabled during DCPLB modifications. On the other hand, we expect that the ICPLB code will be highly sensitive to code size, as instruction caching is disabled during ICPLB modifications.

We describe the methods we will be comparing and then present performance benchmarks, followed by an analysis.

14.2.1 Methods

For each of the 4 methods, we provide the C++, assembler and machine code, where available for restoring the instruction CPLBs. The data ICPLB code is identical except for different pointer targets so is not provided.

Method 1. Indexed Array Loop

This an ordinary C++ loop which restores each of the 15 CPLB protection entries using array subscripts:

```
l4bfin_icplb_disable ();
{

        // 40a0ac:       4b e1 e0 ff     P3.H=ffe0;
        // 40a0b0:       22 e1 09 f0     R2=-4087 (X);
        // 40a0b4:       05 68           P5=0x0;
        // 40a0b6:       0b e1 00 11     P3.L=1100;
        // 40a0ba:       4a 4f           R2<<=0x9;

        // Loop body
        // 40a0bc:       21 32           P4=R1;
        // 40a0be:       6a 44           P2=P5<<2;
        // 40a0c0:       1a 5a           P0=P2+P3; [*]
        // 40a0c2:       0d 6c           P5+=0x1;
        // 40a0c4:       62 5a           P1=P2+P4; [*]
```

[*] pointer calculation based on offset from the start of the array in bytes (`P2`)

```
        // 40a0c6:       08 e4 26 04     R0=[P1+0x1098];
        // 40a0ca:       00 93           [P0]=R0;
        // 40a0cc:       02 32           P0=R2;
        // 40a0ce:       08 e4 36 04     R0=[P1+0x10d8];
        // 40a0d2:       82 5a           P2=P2+P0; [*]
        // 40a0d4:       10 93           [P2]=R0;
        // 40a0d6:       72 68           P2=0xe;
        // 40a0d8:       55 09           CC=P5<=P2;
        // 40a0da:       f1 1f           IF CC JUMP 40a0bc;
        for (int i = 0; i < L4BFIN_ICPLB_NUM_UNLOCKED_ENTRIES; i++)
        {
                BFIN_ICPLB_ADDR [i] = space->saved_icplb_addr [i];
                BFIN_ICPLB_DATA [i] = space->saved_icplb_data [i];
        }
}
l4bfin_icplb_enable ();
```

Notice from the generated assembler how gcc is continually recalculating the source and destination pointers at the lines marked [*] (i.e. continually adding `i`).

Method 2. Pointer Loop

As gcc could not optimise out the array arithmetic in the previous method, we rewrite the code to explicitly avoid the continual array subscript calculations and also, move the loop setup outside of the region where the ICPLB (and instruction caching) is disabled:

```
register volatile u32_t *addr_mmr = BFIN_ICPLB_ADDR;
register volatile u32_t *addr_mmr_end = BFIN_ICPLB_ADDR +
        L4BFIN_ICPLB_NUM_UNLOCKED_ENTRIES;
register volatile u32_t *data_mmr = BFIN_ICPLB_DATA;

register u32_t *addr_src = space->saved_icplb_addr;
register u32_t *data_src = space->saved_icplb_data;

l4bfin_icplb_disable ();
{
        // 40a0cc:       28 90           R0=[P5++];
        // 40a0ce:       4a e1 e0 ff     P2.H=ffe0;
        // 40a0d2:       08 92           [P1++]=R0;
        // 40a0d4:       0a e1 3c 11     P2.L=113c;
        // 40a0d8:       00 90           R0=[P0++];
        // 40a0da:       51 08           CC=P1==P2;
        // 40a0dc:       20 92           [P4++]=R0;
        // 40a0de:       f7 17           IF ! CC JUMP 40a0cc;
        while (addr_mmr != addr_mmr_end)
        {
                *addr_mmr++ = *addr_src++;
                *data_mmr++ = *data_src++;
        }
}
l4bfin_icplb_enable ();
```

Note that gcc is ignoring the `register` hint for the `addr_mmr_end` variable so `P2` is being continually reloaded.

Method 3: Unrolled Pointer Loop

Loop unrolling avoids loop overhead – the conditional branch on each iteration. However, the size of the code is expanded proportional to the number of iterations – in this case, 15:

```
register volatile u32_t *addr_mmr = BFIN_ICPLB_ADDR;
register volatile u32_t *data_mmr = BFIN_ICPLB_DATA;

register u32_t *addr_src = space->saved_icplb_addr;
register u32_t *data_src = space->saved_icplb_data;

l4bfin_icplb_disable ();
{
        // [Sequence repeated 15 times using macros not shown]
        // 40a15e:       08 90           R0=[P1++];
        // 40a160:       24 6c           P4+=0x4;
        // 40a162:       20 93           [P4]=R0;
        // 40a164:       10 90           R0=[P2++];
        // 40a166:       25 6c           P5+=0x4;
        // 40a168:       28 93           [P5]=R0;
        *addr_mmr++ = *addr_src++;
        *data_mmr++ = *data_src++;
}
l4bfin_icplb_enable ();
```

Method 4: Assembler Unrolled Pointer Loop

This code is the same as the Unrolled Pointer Loop except that the repeated section is hand-optimised and contain fewer instructions:

```
r0 = [p2++];
r1 = [p3++];

[p0++] = r0;
[p1++] = r1;
```

14.2.2 Performance results

The following table shows the speeds of the 4 different methods for restoring 15 ICPLB entries, with instruction caching disabled, and 15 DCPLB entries, with data caching disabled:

Method	Size of code executed with respective protection & caching disabled (bytes)		Restore with instr. protection & caching disabled (cycles)	Restore with data protection and caching disabled (cycles)
	Per CPLB entry	Total		
1. Indexed Array Loop	32	48 [*]	414	414
2. Pointer Loop	20	**20**	**270**	270
3. Unrolled Pointer Loop	12	180 [#]	920	**207**
4. ASM Unrolled Pointer Loop	**8**	120 [#]	542	214

Table 14: Methods for restoring CPLB entries

The *Total* figure under *Size of code executed with respective protection & caching disabled* is the size of the machine code placed after the point where the protection unit (and caching) was disabled, up to just before where that protection unit is re-enabled. *Per CPLB entry* is always smaller than or equal to *Total* in size as it contains only part of *Total's* code – the part that is executed for each CPLB entry.

For instance, for the Indexed Array Loop, the loop setup code of 16 bytes executed while the protection was disabled. For each of the 15 CPLB entries copied, 32 bytes of code was executed – this is called *Per CPLB entry*. The *Total* figure is therefore `16 + 32 = 48`, not `16 + 15 * 32 = 496` as this figure measures the *size* of the code, not the total number of cycles. Notice that the *Pointer Loop* appears to have no setup cost (`20 – 20 = 0`) as we managed to execute this while protection was still enabled so it does not appear in our figures. This is a feature – not a bug – because we are trying to determine the effect of code size with disabled instruction or data caching.

Restore with instr. protection & caching disabled (cycles) is the number of cycles that each of the 4 methods took to restore 15 ICPLB entries, while instruction caching was off. *Restore with data protection & caching disabled (cycles)* is the number of cycles that each of the 4 methods took to restore 15 DCPLB entries. These values include all overhead, including loop setup time regardless of whether that was done before or after caching was disabled. Note that these values are inclusive of disabling and re-enabling the relevant protection (and by implication, caching) to avoid executing instrumentation code when caching was disabled. Currently disabling or enabling either takes approximately 18 cycles:

```
p0.h = DMEM_CONTROL or IMEM_CONTROL;           // 1 cycle
p0.l = DMEM_CONTROL or IMEM_CONTROL;           // 1 cycle
r0.h = mask for disabling/enabling protection; // 1 cycle
r0.l = mask for disabling/enabling protection; // 1 cycle
[p0] = r0;                          // 4 cycles (MMR access)
ssync;                              // min. 10 cycles (pipeline flush)
```

[*] The loop setup code (`48 – 32 = 16` bytes), in the other 3 methods, managed to be executed when protection was still enabled.

[#] 15 CPLB entries are restored (the remaining one is locked and not restored) and the loop has been unrolled.

14.2.3 Performance analysis

We now examine the performance figures, we just presented, in detail.

Pointer Loop compared to Indexed Array Loop

With either kind of caching disabled, pointer arithmetic (Pointer Loop) beats array dereferencing (Indexed Array Loop) convincingly. There is no surprise here as pointer arithmetic results in not just smaller code size but also fewer cycles. GCC should be able to perform this conversion automatically as an optimisation but fails to if there is more than 1 array in a loop (there are 4 in our case).

Unrolled Pointer Loop

With instruction caching disabled, the main objective is to reduce the size of the code as each instruction fetch becomes extremely expensive. As a result, at 920 cycles compared to 270 for the Pointer Loop, unrolling the loop, with instruction caching off, backfires spectacularly – the cost of loading instructions becomes higher than the loop overhead saved.

With data caching disabled, but instruction caching still enabled, the size of the code is not of a concern, if we assume the loop unrolling does not result in code so large that the number of instruction cache misses become unpalatable. The objective is to instead avoid uncached accesses to main memory i.e. the DCPLB entries saved in the address space data structure. But we cannot avoid these data accesses as we must restore the 15 DCPLB entries. The number of data accesses is constant for all approaches, assuming all local and loop variables are maintained in registers (which they are). Therefore, we can see from the 207 cycles, instead of 270, that unrolled loops make sense when data caching is disabled.

One might argue that this is an unfair comparison because the Pointer Loop's generated assembler involved unnecessary reloads of the `P2` register, which do not occur here. However, it is not of a concern because:

1. Each reload costs 2 cycles. With 15 iterations, the total cost is `15 * 2 = 30` cycles, assuming no caching effects. Even if we subtracted 30 cycles from the Pointer Loop's measurements, the rank order of the techniques remains unchanged.
2. With instruction caching disabled, the extra 8 bytes of code for each `P2` reload in the Pointer Loop, would actually penalise the Pointer Loop. Yet, we find that the Unrolled Loop is still more inefficient.

Assembler Unrolled Pointer Loop

With instruction caching disabled, this performed better than the Unrolled Pointer Loop due to more compact code (120 bytes instead of 180 bytes).

It is unclear why the Assembler Unrolled Loop performed slightly worse than the Unrolled

Loop when data caching was disabled, when it executes fewer instructions and the machine code is more compact.

14.2.4 Experimental findings

We tried to determine the fastest way to restore the DCPLB registers, where data caching must be disabled, and the fastest way to restore the ICPLB registers, where instruction caching must be disabled.

Instruction protection is highly sensitive to increased code size so loop unrolling is a bad idea. With data protection, loop unrolling improves performance by discarding loop overhead.

GCC is not always smart enough to convert indexed array loops into pointer loops resulting in unnecessary array subscript overhead. Always write pointer loops manually for performance.

14.3 Conclusion

On address space switches, saving and restoring CPLBs is faster than flushing the CPLBs, as it avoids compulsory CPLB misses. Unfortunately, this adds some pagefault optimisation complications.

Blackfin protection changes require disabling the respective protection unit and therefore, the respective type of caching. The fastest way to restore the DCPLB is to use loop unrolling. But the fastest way to restore the ICPLB is a simple loop.

Chapter 15

15 Kernel Development Strategies

This chapter describes some general, high-level techniques for developing reliable kernels. They are important, yet peripheral, observations that we made during the development of the kernel port.

We firstly discuss why kernels should be developed on real hardware. Secondly, we describe how to debug a kernel without any printing. Finally, we introduce the *debug-driven development methodology*.

15.1 Real Hardware

A popular school of thought is that one should primarily develop systems code in simulators, emulators or virtualisers due to stronger debugging capabilities [KDC05].

However, for CPUs which enforces memory protection even in kernel mode (which catches kernel bugs), there are overwhelming reasons to instead develop primarily on real hardware:

1. The virtual environment may be buggy. Hardware is buggy enough already. Software can only be expected to be worse.
2. The virtual CPU may act differently to the real CPU and especially on non-cycle-accurate simulators, differences in cache and pipeline behaviour may result in bugs that only appear on real hardware. Developing on a virtual – rather than a real – platform would be like reading the wrong book.
3. Even if the virtual CPU simulation is completely accurate (cycle accurate, even), it is unlikely that the functionality and interfaces of external devices, such as 3D graphics cards, are cloned identically by the virtual environment.
4. It encourages the development of stronger in-kernel KDB debugging capabilities, which after all, is all one has when a bug only manifests itself in hardware. Furthermore, bugs in production systems running on real hardware can be debugged *when they occur* – gone is the problem of needing – and being unable – to reproduce the bug.

This is not to say that we should avoid using simulators completely as they can assist in debugging – instead, we are suggesting that one should avoid using them as the primary development platform.

However, for CPUs without memory protection, such as the ARM7TDMI, the importance of a simulator is increased as it is the only way to trace kernel memory accesses to find memory bugs.

15.2 Debugging Without Visible Output

It possible, without a simulator, to debug delicate code where even debug printing is not allowed – an arbitrary test can be performed by:

1. Infinite looping if the condition is true
2. Rebooting the CPU if the condition is false

With this binary and lightweight debugging trick, almost any bug can be tracked down without any heavy-weight printing, that might perturb timing.

15.3 Debug-Driven Development

The Blackfin port was only ever executed on real hardware, we needed strong debugging support. As a result, we used a *debug-driven development* methodology, in which the guiding principle is that code always has bugs and/or will be continually changed, so needs to be readily debuggable:

1. The KDB can perform both an in-kernel backtrace and a userland backtrace!
2. Every code path has detailed debug printing that can be turned on at compile time. As there is only one serial port, user debug is distinguished from kernel debug by highlighting the former in green.
3. Assembler is avoided as much as possible on the slowpath as it is difficult to debug. Even the code, that invokes the functions that implement system calls, is written in C++. All other L4 ports resort to assembler.
4. There are plenty of assertions.

We believe that this methodology is a far more reliable way to write software, in general – not just kernels.

15.4 Conclusion

If with memory protection is available, we should develop primarily on real hardware as simulated development environments almost certainly cannot be as faithful as real ones. Simulators should only be used for elusive bugs or if the processor does not support memory protection.

A trick for determining the result of a test without being able to print, is to make one branch to spin forever and the other branch reboot.

The debug-driven development methodology assumes that we will always need to debug so code is written to make this easy. It is believed to be a way of writing very reliable kernels and software in general.

Chapter 16

16 Flaws with L4

In this chapter, we discuss problems with L4 design and the L4-embedded implementation. We analyse the fixed size kernel memory heap, followed by ways to exhaust kernel memory.

16.1 Fixed Kernel Memory Heap

In a traditional operating system, such as Linux, all memory pages can be used by userspace or the kernel. However in L4, the kmem memory heap is a reserved area in kernel space, whose size is fixed at compile time.

If little kernel memory is in use, the rest cannot be used by userspace. Worse still, if more kernel memory is required than is available in the kmem heap, the kernel will start reporting out of memory errors and cannot grab unused pages from userspace. The Sel4 kernel aims to solve this problem using userspace management of kernel memory [EDE06].

Forced to specify a fixed heap size, we unfortunately cannot determine a "correct" size for the kmem heap. We would like to claim something like "a 4MB kmem heap supports `4 * 1048576 / 512 = 8192` threads" (where `512` bytes is the approximate size of a TCB under the single stack kernel). However, we cannot even guarantee support for even a single thread as all kernel memory might be used by that single thread through page table entries.

16.2 Ways to Exhaust Kernel Memory

The section considers various ways that user tasks may attempt to consume all kernel memory and especially take advantage of the fact that the kmem heap is of a fixed size, as discussed in the previous section.

16.2.1 Creating threads and address spaces

There is a potential for denial-of-service attacks by creating large numbers of threads or address spaces, to consume kernel memory. However, L4's design avoids this by making the relevant system calls, `L4_ThreadControl()` and `L4_SpaceControl()` privileged and therefore, only available to the root task.

16.2.2 Touching memory

Similarly, should a userspace thread touch every page in the virtual address space, it does not matter since it is a privileged pager that ultimately makes the decision as to whether to create a mapping and consume kernel memory.

So, in both of these cases, the root task asks as a resource manager that accepts or denies requests to prevent kernel memory from being exhausted. But this is suboptimal as again, the root task must be trusted [p77, Chapter 5.2] so there should be an ability to delegate thread creation and paging privileges onto other threads – this would also avoid unnecessary IPCs (and expensive address space switches) to the root task. Resource containment would then have to be added to L4.

16.2.3 TCB allocation

When a thread is deleted, its TCB's Thread ID field is merely set to 0 to mark it as invalid. However, the physical backing for it is *never* deallocated.

If threads are continually being created and deleted and thread IDs recycled, this saves a lot of memory allocation overhead. However, if thread IDs are not reused, kernel memory is simply lost. Nevertheless as only the privileged root task can create threads, one can prevent kernel memory from being exhausted in this way.

16.2.4 Dummy TCB mappings

In kernels with virtual TCB arrays, the dummy TCB mapping created on accesses to invalid threads, opens up another denial-of-service opportunity: an unprivileged thread can make L4 run out of, or at least waste, kernel memory by IPC'ing to all invalid threads in the thread ID space. As each IPC incurs a TCB validity test, additional dummy TCB mappings are created

and memory is wasted through page table entries.

At first, we would imagine that redirectors, which intercept IPCs and send them to another thread for checking, could be used to audit all IPCs. However, this is infeasible as if all IPC calls are to be monitored, performance suffers with a doubling of IPC calls. Also, redirectors must not use the same TCB validity test (as that would create the dummy TCB mappings we are trying to avoid) so must somehow maintain a list of valid threads. But worst of all, this does not work because the current L4 implementation performs the thread validity checking (and therefore, adds the dummy TCB mapping) *before* passing on the message to be checked to a redirector.

16.3 Conclusions

Like almost all software, L4 has flaws – some of which we analysed:

The fixed size of the kernel heap means that it is too small at times and too big at others. With this design, it is unclear what the size of the heap should be.

The user may attempt to waste kernel memory through creating large numbers of threads or address spaces, touching pages and never recycling thread IDs. However, the trusted root task is involved in all these so can thwart these attempts.

However, an attack that IPCs all invalid threads in the thread ID space will succeed by consuming kernel memory with page table mappings for the dummy TCB.

Chapter 17

17 Global Evaluation

In previous chapters, we evaluated qualitatively and quantitatively aspects of the design and implementation. However, we have not measured the conglomerate of all this work.

We again use the pingpong microbenchmark [p48, Chapter 4.1], with the addition of the *DCPLB refill* test for measuring the cost of a data CPLB miss and refill. Recall that microbenchmarks are dangerous for making sweeping claims about performance but we do not have a choice here due to the lack of *serious* macrobenchmarks for our platform. In any case, at least we know that we are presenting the best case performance figures.

We use these figures to compare the performance of the initial and final kernel ports, then examine the performance of the final port in detail.

17.1 Performance Improvement

In this section, we compare the speed of the initial kernel to the final kernel to validate the effectiveness of the design and implementation techniques we discussed throughout the thesis.

The following table compares the speed of initial kernel to the final kernel. All figures are with caching on.

Operation	*Initial Kernel (cycles)*	*Final Kernel (cycles)*	*Improvement (times)*
Kernel entry and exit	4,049.2	252.16	16.1
System call [*]	7,775.3	394.35	19.7
DCPLB refill	- [#]	1,425.31	- [#]
Intra-AS IPC			
0 MRs	17,566.3	910.5	19.3
4 MRs	17,218.7	967.10	17.8
8 MRs	17,272.2	1,003.20	17.2
60 MRs	17,890.7	1,471.71	12.2
Inter-AS IPC			
0 MRs	39,203.1	1,567.86	25.0
4 MRs	38,907.2	1,625.4	23.9
8 MRs	38,962.8	1,660.45	23.5
60 MRs	39,582.0	2,163.77	18.3

Table 15: L4/Blackfin benchmarks (initial and final kernels)

All measurements indicate an order of magnitude improvement in performance. The change in Kernel entry and exit, System call and Intra-AS IPC times can be attributed to the micro-optimisations we performed [p137, Chapter 13]. The Inter-AS IPC improvement is even greater than that enjoyed by Intra-AS IPC as we also eliminated the majority of indirect address space switching costs – we no longer do a simple flush of the CPLBs, which had resulted in mandatory CPLB refills on address space switches [p153, Chapter 14.1].

Notice how unstable the IPC measurements for the *initial* kernel are (the 0MRs case is slower than the 4MRs case, which theoretically does more work) due to cache effects of an overly large cache footprint.

As the number of message registers to be transferred increases, the performance differential between the final and initial kernels, for the Intra-AS IPC and Inter-AS IPC, decreases. This is because we did not manage to improve the message register copying time to the extent that we

[*] The system call was the Blackfin-specific `L4_KDB_SystemClock()` which merely looks up a 64-bit value from the scheduler class.
[#] Figure unavailable due to time constraints.

managed to speed up the trap code and general IPC logic, using micro-optimisations.

In conclusion, the work we did in this thesis sped the `pingpong` microbenchmark by more than 12 times, by all measures.

17.2 Final Kernel Port Performance

We now discuss each component of the final kernel's performance, from the previous `pingpong` benchmark results [p177].

The Inter-AS IPC performance, even in the case where 60 words are transferred, outperforms ucLinux's best case context switch time of approximately 2,485 cycles [Hen06]. Although no fastpath has yet been written (so we are measuring a predominately C++ kernel), all figures are still higher than would be expected from L4 slowpaths. Therefore, this section will look at each component in detail.

But firstly, for illustrative purposes, the following graph compares each component of kernel performance:

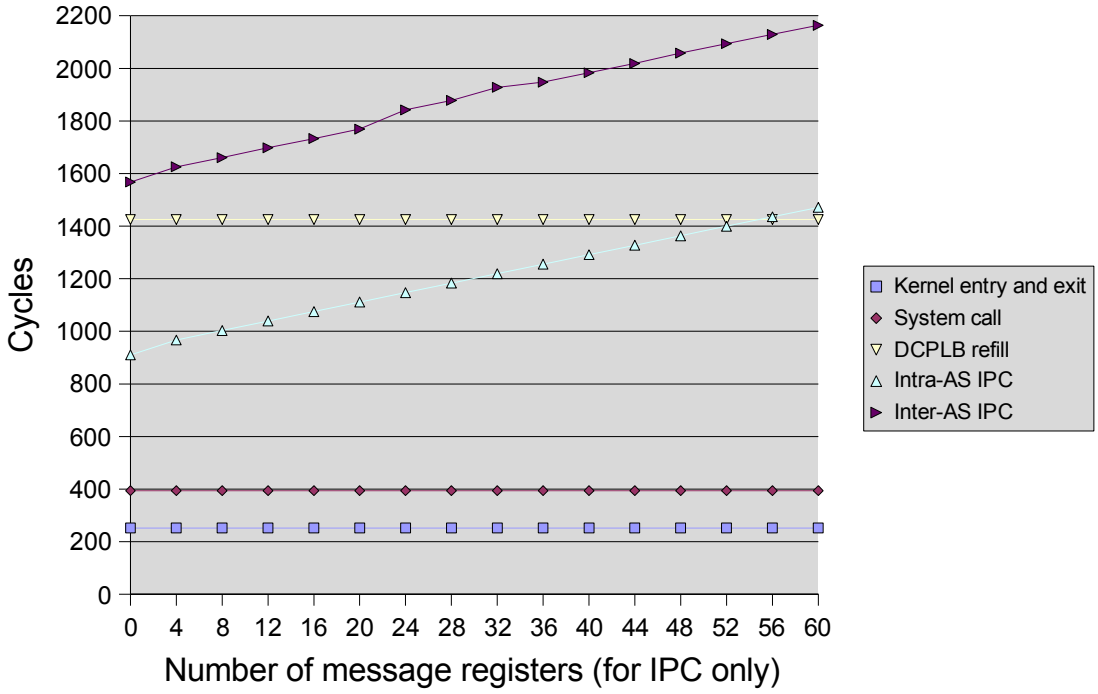

Illustration 1: Final L4/Blackfin kernel's performance

The time taken by the IPC operations seems to increase linearly with the number of message registers, so this gives us extra confidence that the numbers are correct.

Now, let us look at each component in detail:

17.2.1 Kernel entry and exit

We appear to have a very high cost of 252.16 cycles for an assembler path, especially when we

are not actually preserving the whole trapframe. However, this is not particularly surprising since basic programming constructs on the Blackfin appear to be very slow. For instance, as we found previously, a call to an empty function takes at least 18 cycles [p17, Chapter 2.1]. Of course, we are not suggesting that we should use functions in assembler paths, but rather that this illustrates how even a no-operation programming construct on the Blackfin is expensive, so that one can imagine the cost of constructs that actually perform computation.

Furthermore, our kernel entry and exit code is shared between all paths – system calls, MPU refills, MPU page faults and interrupts. As a result, it is too general and extremely unoptimised. For instance, in order to untrap from an exception, we spend 26 cycles to determine that we came from an exception and then jump to the appropriate code path.

```
// Everything, except where noted, execute in 1 cycle
// (assuming no cache misses so the reality is likely far worse).
p0.h = BFIN_IPEND_HIGH;
p0.l = BFIN_IPEND_LOW;
r0 = [p0];  // [4 cycle MMR read]

// In "emulation" mode?
r2 = 1;  // 1 << 0
r1 = r0 & r2;
cc = r1 == 0;
if ! cc jump untrap_rte_label;

// Handling NMI?
r2 = 4;  // 1 << 2
r1 = r0 & r2;
cc = r1 == 0;
if ! cc jump untrap_rtn_label;

// Handling an exception?
r2 = 8;  // 1 << 3
r1 = r0 & r2;
cc = r1 == 0;
if ! cc jump untrap_rtx_label;  // [9 cycles mispredicted branch]
```

However, if we had avoided using this lazy general path and coded a specialised untrap into the exception-handling path, none of these 26 cycles would have been needed. The rest of the trap code is just as inefficient, explaining the 252.16 cycle cost. When writing the fastpath, this kind of overhead should be avoided.

17.2.2 System call

System calls take approximately an extra 142.19 cycles, on top of a kernel entry and exit, for two reasons. Firstly, system calls, which are always implemented as exceptions, must defer their work to an interrupt handler, to handle the case of a nested MPU miss exception [p88, Chapter 7.2.1]. Secondly, we enter the C++ path for determining which system call to invoke. Recall that basic C++ constructs on the Blackfin are expensive – a call to an empty function takes at least 18 cycles [p17, Chapter 2.1]. Also, the calling convention results in constant saving and restoring of registers and for instance, the constant recalculation of the current TCB pointer in different functions. A fastpath would avoid this C++ overhead.

17.2.3 DCPLB Refill

The fully C++ DCPLB refill handler enters the kernel through the system call path but takes an extra 1,030.96 cycles to execute. Here is the cost breakdown:

Component	*Approximate cycle count*
System call overhead	400
Full trapframe preservation	Approx. 45 words * 2 = 90 cycles
Two level page table walk	400
Additional C++ overhead and DCPLB manipulation	400
Benchmark limitation	100
Total:	1,390

Table 16: DCPLB refill cost breakdown

Unlike a system call, we must preserve the full trapframe of forty-five 32-bit words. Secondly, we must walk a somewhat complicated two level page table structure that is actually 4 levels deep to handle the 4 page sizes and the code that currently does this is too general as it actually walks an *n-level* page table [p65, Chapter 4.5.4]. Thirdly, there is the additional C++ overhead for the whole refill path and adding the entry to the DCPLB. Again, recall that basic C++ constructs on the Blackfin are expensive – a call to an empty function takes at least 18 cycles [p17, Chapter 2.1] - so these hundred cycle figures are reasonable.

Lastly, 100 cycles are inadvertently lost to extra DCPLB refills caused by the eviction of `pingpong` DCPLB pages due to a limitation in the benchmark. Such an eviction occurs approximately every 15 tests and `1425.31 / 15 = 95.0` is approximately the 100 cycles we list above.

An assembler fastpath would certainly perform far better by not saving registers it does not need to use, using a non-general page table walk and avoiding C++ overhead.

17.2.4 Intra-AS IPC

The 910.5 cycle cost for Intra-AS IPC is composed of the following:

Component	*Approximate cycle count*
System call overhead	400
`sys_ipc()` C++ IPC implementation	500
Total:	900

Table 17: Intra-AS IPC cost breakdown

It is clear that the C++ overhead of `sys_ipc()` – which we did not write – is unbearable. After all, all Intra-AS IPC needs to do is context switch to a thread in the *same* address space and

copy message registers. The latter does not even have to happen if message registers are register-backed and the message is small enough.

We can expect that an assembler fastpath that saves few registers, avoids general paths and avoids C++ overhead, would perform almost an order of magnitude better.

17.2.5 Inter-AS IPC

The Inter-AS IPC cost is composed of the cost of Intra-AS IPC (approximately 900 cycles) and the Blackfin architecture's lower bound cost for an address space switch – the cost of restoring CPLBs (approximately 600 cycles) [p156, Chapter 14.1.3].

Compared to other operating systems, claiming that this is our context switching time is "cheating" slightly because L4's IPC calls do not preserve the full trapframe on the switch. However, even if we add the overhead of saving and restoring up to forty-five 32-bit words – 90 cycles – to our 0MR Inter-AS IPC cost of 1,567.86 cycles, we still outperform Linux's lower bound context switching time of approximately 2,485 cycles [Hen06]. And this is the performance of our slowpath – the fastpath, could be expected to be more than several hundred cycles faster by avoiding C++ overhead.

17.2.6 IPC message register copying costs

Notice from the graph [p179, Illustration 1: Final L4/Blackfin kernel's performance], that on average, every 32-bit message register copied costs an additional 10 cycles. This is fairly close to the theoretical minimum of 8 cycles for an iteration of the message-register-copying loop:

```
loop_top:
      r0 = [p2++];    // Read source message register [1 cycle]
      [p1++] = r0;    // Write destination message register [1 cycle]
      cc = p2 == p0;  // Test for end of loop [1 cycle]
      if ! cc jump loop_top (bp);  // Next iteration [5 cycles]
```

The missing 2 cycles probably come from a pipeline dependency between the consecutive accesses to R0 but this requires further investigation.

17.3 Conclusion

The effectiveness of the design and implementation techniques we discussed throughout the thesis are exemplified by an order of magnitude performance difference between the initial and final kernels.

We only have a slowpath yet our context switches already perform far better than ucLinux. However compared to other L4 ports, performance is still disappointing.

We believe that fastpath DCPLB refill and IPC paths, written in assembler, are the answer. We have already shown that C++ is very inefficient. Hand-crafted assembler would also avoid the overhead of the C++ calling convention – the constant saving and restoring of registers and the recalculation of items such as the current TCB. Furthermore, we would not need to save registers we do not use and we can avoid slow generalised code. Such a fastpath is expected to be an order of magnitude faster for all benchmarked components except Inter-AS IPC, which suffers from an architecture lower-bound address space switching cost of approximately 600 cycles.

Chapter 18

18 Conclusion

In this final chapter, we compare our achievements to the goals we stated in the introductory chapter. We also discuss future directions.

18.1 Achievements

We have discussed the design issues in supporting L4 on MPU-based processors. Such issues include no virtual memory, protected kernel addressing and high-performance address space switching with untagged MPUs. Our findings are not Blackfin-specific and generalise to other chips, such as the ARM1156T2-S and even the MPU-less ARM7TDMI.

We also analysed additional general design issues such as reducing the memory used by UTCBs, factors affecting the system call convention and register-backing of message registers. We also documented kernel development strategies and observations about L4. We hope all of our findings will be useful to future kernel developers.

We have constructed a port of L4 to Blackfin based on our analyses. The port also implements recent related work – the single kernel stack and physical TCB arrays. We have described the micro-optimisations used to speed up the Blackfin port, most of which generalises to other architectures.

Our context switching time of 1,567.86 cycles already outperforms ucLinux's 2,485 cycles. Once a fastpath is written, this gap will widen even further in our favour and we will approach L4's traditional hundred-cycle address space switching time.

There is still work to be done on the Blackfin port before it is deployable. However, most of the port has been completed and as our findings allow L4 to support an additional category of processors – those without virtual memory – the future looks bright.

18.2 Future Work

In this section, we describe the remaining work we believe should be done.

18.2.1 ARM ports

The findings in this thesis generalise to ARM processors without virtual memory. Given the popularity of ARM, we should make ports to the MPU-based ARM1156T2-S and protection-less ARM7TDMI (used in the iPod).

18.2.2 Tweaking the Blackfin kernel

This thesis has concentrated on the difficult architectural, L4 API and optimisation issues. The remaining issues are comparatively easy – but time consuming – coding exercises that can be done later:

Cache API

Flushing the D-caches and I-caches using the `DMEM_CONTROL` and `IMEM_CONTROL` registers as suggested by the Blackfin manual [Ana05] does not work. An investigation should be made into the feasibility of looping through all cache lines and flushing each individually – although this is expected to be very slow.

The port should respect `L4_MapControl()`'s page cacheability attributes. Currently, the Blackfin port assumes that all addresses below the Blackfin's physical memory limit of 128MB should be write-back cached and that all other addresses – potentially, but not always, device memory – should be uncached. This behaviour always produces correct results but some memory above 128MB need not be uncached and the lack of caching is wasting cycles.

More efficient DCPLB replacement policy

We should also maintain a low overhead queue of free CPLB entries. Only when this queue is empty should we invoke our replacement policy [p64, Chapter 4.5.3].

Once the kernel is very stable, we should switch to a psuedorandom replacement policy to avoid the cyclical worst case of the current FIFO policy.

Eliminate DCPLB misses in kernel mode

We previously suggested a global UTCB array scheme [p92, Chapter 7.3.3] and a KIP entry pre-filling scheme [p89, Chapter 7.2.2]. Implementing these would eliminate expensive in-

kernel DCPLB misses and also simplify the kernel.

We also suggested alternative and pessimistic approaches: pre-filling the DCPLB for UTCB accesses [p89, Chapter 7.2.2] and disabling the protection [p90, Chapter 7.2.2].

A thorough comparison of these methods should be performed.

Locking of TCBs in L1

There is 4KB – 36KB in total of Data SRAM on Blackfin 53x models. It would be interesting to measure the performance increase of placing TCBs here to reduce D-cache pressure.

Should there be an insufficient amount of Data SRAM to store all of the system's current TCBs, LRU-based replacement could be used. The real cost of this method are the cycles and D-cache footprint of (hopefully, occasionally) copying TCBs to and from L1 and SDRAM. If this is too expensive, L4 could either limit the maximum number of threads supported so that all will fit into SRAM or state arbitrarily that the first *n* threads will be placed into SRAM.

An issue would be whether this SRAM should be a dedicated kernel resource for simplicity or shared with userspace. If it is to be shared with userspace, the ordinary page-based protection mechanism could be used. However, one might wonder what performance gains could be achieved by giving userspace access to SRAM but the end-to-end argument [SRC84] suggests that such a decision should be pushed as high as possible up the application stack, which in this case means leaving the decision up to userspace, as only the application knows its resource needs best.

Implement fastpath

Implement the IPC, DCPLB/ICPLB refill and interrupt handlers in assembler. These form the most critical parts of kernel performance. Use the performance monitoring unit extensively to hand optimise the code. Load this code into the instruction SRAM to reduce I-cache pressure.

The Skyeye simulator [Sky06] can not be used as it is not cycle accurate.

VisualDSP++ (Windows) [Ana06b] provides a cycle accurate simulator. Porting L4 to the Analog Devices toolchain is difficult given the number of gcc-isms in L4 but should be considered.

Another option is to convert the ELF generated by gcc to a format readable by VisualDSP++. It is possible to do this using the `elf2elf` tool if one uses an older version of VisualDSP++, 3.5 [Get06].

Support the watchdog timer

An investigation should be made into whether the watchdog timer is used in any real production environment. It may only be used for emulators which is a different topic entirely. However, it may be useful in realtime systems for flagging tasks that are running for longer than they should

be.

If the watchdog timer is to be supported, NMI kernel entry and exits will need to be supported.

Paravirtualise ucLinux

For a start, ucLinux's register usage and system call convention must be analysed in greater detail so that we can potentially virtualise it. As no one has previously written an L4 port for an MPU-based processor, no one has yet ported ucLinux to L4. There are surely other design and implementation issues that one will discover when embarking on this "Wombat/Blackfin" project.

If we do this, we immediately gain access to Linux macrobenchmarks. We should then re-evaluate the trade-offs made in this thesis as a result of the `pingpong` microbenchmark.

Work around hardware anomalies

The Blackfin processor suffers from numerous documented glitches from revision to revision that can compromise the kernel [Ana06a]. The solutions to these problems are well documented inside that document. An analysis should be made as to which revisions of the Blackfin chip are popular so that the number of revisions to be supported can be limited, minimising the number of hacks required.

Serial port in release mode

The STAMP board only has 1 serial port which the kernel uses for the KDB debugger. In release mode, the serial port should be available to userspace. However, this has never been tested.

After everything is implemented and debugging of the kernel finished, debug statements will not need to be tweaked. Therefore, it would then be a good time to ensure the serial port is not used by the kernel in release mode.

18.2.3 After tweaking the Blackfin kernel

With the above changes, the kernel should be in a very mature state and should be released to the public as a beta for community feedback.

The Iguana operating system should be ported to Blackfin so that eventually, the whole ERTOS stack will run on the Blackfin in direct competition with GNU/Linux. Any effects of physical memory and protected kernel addressing on Iguana should be analysed, as they were for L4.

Other versions of the Blackfin such as the 535 should be supported. Currently, GNU toolchain support does not exist for the 535. Another objective would be to port L4 to Analog Device's VisualDSP++. This would force not only the L4/Blackfin combination to be more portable but

also L4 itself.

Bibliography

[Ana05] Analog Devices, "ADSP-BF53x/BF56x Blackfin Processor Programming Reference - Revision 1.0, June 2005," 2005; http://www.analog.com/processors/blackfin/technicalLibrary/manuals/index.html

[Ana05b] Analog Devices, "ADSP-BF531/ADSP-BF532/ADSP-BF533: Blackfin Embedded Processor," 2005; http://www.analog.com/processors/blackfin/technicalLibrary/manuals/index.html

[Ana05c] Analog Devices, "ADSP-BF533 Blackfin Processor Hardware Reference - Revision 3.1, June 2005," 2005; http://www.analog.com/processors/blackfin/technicalLibrary/manuals/index.html

[Ana06a] Analog Devices, "ADSP-BF533 Blackfin Anomaly List for Revisions 0.3, 0.4, 0.5 (Rev V, 09-18-2006)," 2006; http://www.analog.com/UploadedFiles/REDESIGN_IC_Anomalies/691920698IC_Anomaly_BF533.pdf

[Ana06b] Analog Devices, "VISUALDSPBF - VisualDSP++ for Blackfin Processors," 2006; http://www.analog.com/en/epProd/0,2878,VISUALDSPBF,00.html

[Arm04] Arm Limited, "ARM7TDMI (Rev 4) Technical Reference Manual," 2004; http://www.arm.com/pdfs/DDI0210B_7TDMI_R4.pdf

[Arm05] ARM Limited, "ARM1156T2-S r0p0 Technical Reference Manual," 2005; http://www.arm.com/pdfs/DDI0338C_arm1156t2s_r0p0_trm.pdf

[Bla05] Blackfin ucLinux Project, "bfin-elf-gcc Calling Convention," 2006; http://docs.blackfin.uclinux.org/doku.php?id=using_inline_assembly_calling_external_assembly_from_c

[Bla06] Blackfin ucLinux Project, "Cache Management," 2006; http://docs.blackfin.uclinux.org/doku.php?id=cache_management

[Bla06c] Blackfin ucLinux Project, "Blackfin: Stamp Development Platforms and Modules," 2006; http://blackfin.uclinux.org/projects/stamp

[Bla06d] Blackfin ucLinux Project, "Blackfin ucLinux Project," 2006; http://blackfin.uclinux.org/

[Bla06e] T. Blattmann, "Porting L4Ka::Pistachio to Mips32," 2006; http://i30www.ira.uka.de/teaching/thesisdocuments/l4ka/2006/blattmann_study_porting-l4ka-pistachio-to-mips32.ps

[Car90] D. Carta, "Two fast implementations of the 'minimal standard' random number generator," *Comm. ACM vol. 33 no. 1*, ACM Press, 1990, pp. 87-88

[CNC00] C. Chen, G. Novick, K. Shimano, "Pipelining," 2000; http://cse.stanford.edu/class/sophomore-college/projects-00/risc/about/index.html

[EDE06] D. Elkaduwe, P. Derrin, K. Elphinstone, "Kernel data – first class citizens of the system," *Proc. 2nd. Int'l Workshop Object Systems and Software Architectures*, 2006

[Get06] R. Getz, "RE: linking VisualDSP libraries into uClinux," 2006; http://blackfin.uclinux.org/forum/forum.php?thread_id=1633&forum_id=39

[Hen06] M. Hennerich, "uClinux on the Blackfin DSP Architecture: Part 3," 2006; http://www.eet.com/esc/showArticle.jhtml;jsessionid=4GAY1UC503N5IQSNDBNCKHSCJUMEKJVN?articleID=185302712

[Hen06b] M. Hennerich, "Blackfin ucLinux Applications," 2006; http://www.analog.com.ru/Public/uClinux on Blackfin/[5]-uClinux Applications.pdf

[HLS02] D. Holland, A. Lim, M. Seltzer, "A new instructional operating system," *ACM SIGCSE Bull., Proc. 33rd SIGCSE technical symp. computer science education vol. 34 no. 1*, ACM Press, 2002, pp. 111-115

[Ipo06] iPod Linux Project, "iPod Linux - Linux for your iPod," 2006; http://www.ipodlinux.org/

[Joh97] M. Johnson, "Linux Kernel Hackers' Guide: How System Calls Work on Linux/i86," 1997; http://tldp.org/LDP/khg/HyperNews/get/syscall/syscall86.html

[KDC05] S. King, G. Dunlap, P. Chen, "Debugging operating systems with time-traveling virtual machines," *Proc. USENIX 2005 Ann. Technical Conf.*, USENIX Association, 2005, pp. 1-15

[Kuz05] I. Kuz, "L4 User Manual NICTA L4-embedded API Document Version 1.11 of October 5, 2005," 2005; http://www.ertos.nicta.com.au/software/kenge/pistachio/latest/userman.pdf

[Lie96] J. Liedtke, "Toward real microkernels," *Comm. ACM vol. 39 no. 9*, ACM Press, 1996, pp. 70-77

[Nic05a] National ICT Australia, "NICTA L4-embedded Kernel Reference Manual Version NICTA N1," 2005; http://www.ertos.nicta.com.au/software/kenge/pistachio/latest/refman.pdf

[Nic05b] National ICT Australia, "NICTA L4 Microkernel to be Utilised in Select QUALCOMM Chipset Solutions," 2005; http://www.ertos.nicta.com.au/press/051124_L4_Qualcomm_vfinal.pdf

[Nic06a] National ICT Australia, "The L4 Microkernel," ; http://www.ertos.nicta.com.au/research/l4/

[Nic06b] National ICT Australia, "L4-Embedded," 2006; http://www.ertos.nicta.com.au/research/l4/embedded.pml

[Nic06c] National ICT Australia, "Tool: magpie," 2006; http://www.ertos.nicta.com.au/software/kenge/magpie/latest/

[Nou05] A. Nourai, "A Physically-Addressed L4 Kernel," 2005; http://www.disy.cse.unsw.edu.au/theses_public/05/anourai.pdf

[Pan68] R. Pankhurst, "Operating Systems: Program overlay techniques," *Communications of*

the ACM vol. 11 no. 2, ACM Press, 1968, pp. 119-125

[Pot99] D. Potts, "L4 on Uni- and Multiprocessor Alpha," 1999; http://www.ertos.nicta.com.au/publications/

[RC01] A. Rubini, J. Corbet, *Linux Device Drivers 2nd ed.*, O'Reilly & Associates, Inc., 2001

[Sch96] S. Schonberg, "The L4 Microkernel on Alpha - Design and Implementation," 1996; http://os.inf.tu-dresden.de/drops/doc.html

[Sky06] SkyEye Project, "SkyEye – Open Source Simulator," 2006; http://www.skyeye.org/

[SLF+94] J. Chase, H. Levy, M. Feeley, E. Lazowska, "Sharing and protection in a single-address-space operating system," *ACM Trans. Computer Systems (TOCS) vol. 12 no. 4*, ACM Press, 1994, pp. 271-301

[SML+02] M. Swift, S. Martin, H. Levy, S. Eggers, "Nooks: an architecture for reliable device drivers," *Proc. 10th ACM SIGOPS European Workshop*, ACM Press, 2002, pp. 102-107

[SRC84] J. Saltzer, D. Reed, D. Clark, "End-to-end arguments in system design," *ACM Trans. Computer Systems (TOCS) vol. 2 no. 4*, ACM Press, 1984, pp. 277-288

[Uns03] University of Karlsruhe, "L4 eXperimental Kernel Reference Manual Version X.2," 2003; http://l4ka.org/projects/pistachio/l4-x2-r2.pdf

[Uns05] University of New South Wales, "Caching From an OS Perspective: COMP9242 2005/S2 Week 3," 2005; http://www.cse.unsw.edu.au/~cs9242/05/lectures/03-cache.pdf

[War05] M. Warton, "Single Kernel Stack L4," 2005; http://www.disy.cse.unsw.edu.au/theses~/05/mwarton.pdf

[Wig99] A. Wiggins, "The Design and Implementation of the L4 Microkernel on the StrongARM SA-1100," 1999; http://www.disy.cse.unsw.edu.au/theses_public/99/awiggins.ps.gz

[WW94] C. Waldspurger, W. Weihl, "Lottery Scheduling: Flexible Proportional-Share Resource Management," *Proc. 1st Symp. Operating System Design and Implementation*, Usenix Assocation, 1994