Correct, Fast, Maintainable – Choose Any Three!

Bernard Blackham and Gernot Heiser
NICTA and University of New South Wales, Sydney, Australia
{bernard.blackham,gernot}@nicta.com.au

Abstract

The common-case IPC handler in microkernels, referred to as the *fastpath*, is performance-critical and thus is often optimised using hand-written assembly. However, compiler technology has advanced significantly in the past decade, which suggests that we should re-evaluate this approach.

We present a case study of optimising the IPC fastpath in the seL4 microkernel. This fastpath is written in C and relies on an optimising C compiler for good performance. We present our techniques in modifying the C sources to assist with compiler optimisation. We compare our results with a hand-optimised assembly implementation, which gains no extra benefit from hand-tuning.

1 Introduction

Focusing on the common case is the mantra of optimisation. For microkernels this is message-passing inter-process communication (IPC), and as a result, there has been a strong focus on improving the performance of IPC [LES+97, GCC+05].

Many microkernels achieve good performance by providing *fastpaths* for IPC, which improve the performance of IPC operations by an order of magnitude. Fastpaths are created in order to perform a specific operation for the most common set of conditions. If any of these conditions do not hold, a fastpath reverts back to the standard code path through the kernel (referred to as the *slowpath*). A key property of a fastpath is that the kernel's behaviour should be functionally identical with or without it.

The first L4 microkernels were coded entirely in assembly, in order achieve the best possible performance from the hardware [L4Impl]. Later versions such as Pistachio and Fiasco were written in C or C++, however performance-critical sections such as the IPC fastpaths were still in assembly.

As assembly does not need to comply with any ABIs (other than at the system-call interface), more opportunities for optimisation are available. By hand-crafting fastpaths for specific CPUs, authors can minimise pipeline stalls by careful instruction scheduling, avoid cache misses using strategic prefetching, and craft the control flow to minimise costly branches. Using these techniques, impressive IPC times have been achieved – e.g. 151 cycles on the Intel XScale PXA255 (ARMv5) [L4H] and 36 cycles on Itanium [GCC+05].

However, these results challenge maintainability. Assembly code is generally more difficult to read, write and maintain; fastpaths are extremely fragile, requiring full knowledge of the system to confidently make any modifications.

seL4 [KEH+09] is the world's first general-purpose microkernel to have a complete machine-checked proof of correctness. The seL4 code base is written almost entirely in C, with only a few hundred lines of assembly code where necessary. The formal verification currently applies only to the C code, not assembly.

seL4 similarly has a fastpath which improves the performance of IPC by an order of magnitude. However the fastpath is written in C, as our verification infrastructure could only formally verify C code, not assembly. As a result, we expected a performance penalty, but estimated it to be less than 10 %.

We have observed that modern C compilers for RISC architectures are becoming competitive with assembly crafted by a talented and skillful programmer. Using the gcc compiler on ARM, we have been able to optimise the compiled machine code by modifying only the C source code.

We have improved the speed of the fastpath by 35 % through tuning of the C code to aid compiler optimisations. Our fastpath is competitive with hand-crafted assembly, and with other microkernels on the same architecture. In this paper, we explore various techniques for optimising the assembly output of the compiler by refining the C code used for IPC. We show that modern compilers can obtain almost all of the gains attainable with hand-optimised assembly, even for low-level kernel code.

2 Background

The debate over the merits of optimisation in assembly code vs C (or other higher-level languages) dates back to the 1970s and continues on today [Hyd].

Clearly, a knowledgeable and talented assembly developer, given enough time and resources, can outperform a compiler. However there is the secondary issue of maintainability. One can argue that highly-optimised C code requires just as careful maintenance as an equivalent assembly implementation in order to preserve correctness and performance. We contend that a greater level of skill, knowledge and care is required to maintain an assembly version.

Whether compilers would ever catch up to the level of a talented assembly developer is an old question. Massalin investigated the idea of finding not just an optimised sequence of instructions, but an *optimal* sequence for a given loop-free construct [Mas87]. He wrote a brute-force compiler called a *superoptimiser* to exhaustively search for the optimal sequence. Given the exponential nature of the super-optimiser, it is limited to short sequences of code (a dozen instructions in Massalin's work).

Liedtke states that a microkernel should place emphasis on IPC performance above all other considerations [Lie93]. For the Pentium, MIPS and Alpha architectures, the inherent architectural costs of IPC have been shown to range between 45 and 121 cycles [LES+97]. On the ARM architecture, IPC times as low as 151 cycles have been demonstrated on the ARMv5 Intel XScale PXA255 [L4H]. Our work focuses on the ARM11 (ARMv6) CPU core, which has a deeper 8-stage pipeline. On this core, L4 kernels have achieved 206 cycles for a one-way IPC [KEH+09].

Gray et. al. describe their experience optimising the IPC fastpath for the Itanium processor [GCC+05], with a complex VLIW pipeline. They could reduce their IPC time from 508 cycles to 36 cycles by hand-optimising their assembly. We believe that on simpler pipelines common to RISC architectures, modern compilers can achieve closer to optimal output.

3 Microkernel IPC

IPC is often provided in both synchronous and asynchronous forms, and seL4 is no exception. Synchronous IPC transfers a message only when both the sender and receiver are at a "rendezvous" point – specifically, the sender must be ready to send and the receiver must be ready to receive. This allows for a direct transfer of data from sender to receiver. On the other hand, asynchronous IPC does not require the coordination of threads, as messages are buffered in the kernel until the receiver is ready receive.

In L4-derived microkernels, synchronous IPC is typically provided by five basic primitives:

- *Send* is a one-way message transfer to another thread. The send will block until the recipient thread is ready to receive.
- *Wait* receives a message from any thread that is ready to send data to it, or blocks if no threads are ready to send.
- *Call* is a combined send and receive operation to another thread, and will run to completion or fail with an error. It is often used by a client to request an operation be performed by a server. These semantics guarantee to the server that it can respond without waiting or needing to buffer the response.
- *Reply* is a non-blocking send, used to a respond to a message received with *Wait*. If the sender had used *Call*, then it is guaranteed to be blocked and ready to receive the response.
- *ReplyWait* combines the effects of *Reply* and *Wait* together. This sequence is frequently used by servers, and in most cases allows for a direct context switch.

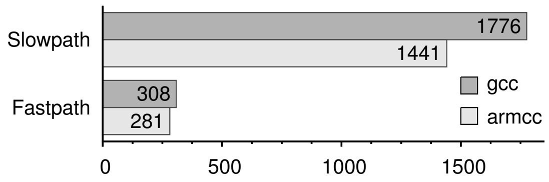

Figure 1: One-way cycle counts of the IPC slowpath compared to the original IPC fastpath.

This synchronous IPC model is best suited to a priority hierarchy such that servers always have a higher priority than their clients. This allows both *Call* and *ReplyWait* operations to directly transfer data and control flow from clients to servers and vice-versa, significantly improving IPC efficiency. As a result, optimising these two operations offers the largest benefit to real systems. The remainder of this paper focuses on optimising these two synchronous IPC operations within the seL4 microkernel.

Anatomy of an IPC fastpath In many performance-critical applications, the focus of optimisation is commonly on tight loops found in computation kernels or memory copying. However, in a microkernel-based system, the IPC path, unlike typical hotspots, has very few tight loops and is largely composed of conditional branches. This precludes many of the usual optimisation techniques.

Although there are a large number of branches, the majority of these handle exceptional circumstances. The most common scenario for a synchronous IPC, and the one that reaps the most benefit in optimising for, is where a thread A sends a message to a thread B, and:

- thread B is ready and waiting for a message;
- thread B can begin executing immediately under the current scheduling discipline; and
- no error conditions occur.

In such a case, we can pass control flow directly from thread A to thread B, copying the message directly between the two threads with no buffering.

There are numerous steps and checks involved in handling a common IPC operation in seL4, such as testing that all objects involved in the operation are valid and in the correct state for the fastpath. If any of the checks fail seL4 reverts to the slowpath. Assuming all the checks succeed, the fastpath can proceed to transfer the data and control to the new thread.

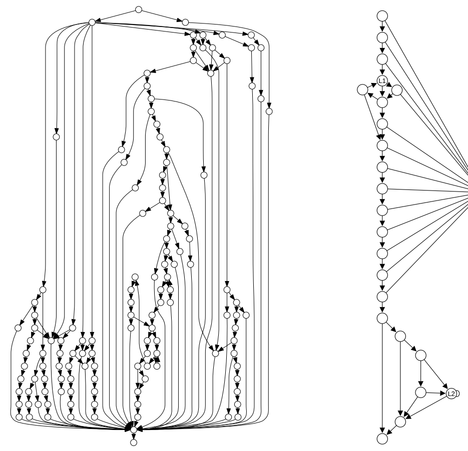

Figure 2: Control flow graphs of the slowpath (left) and fastpath (right) of seL4. Each node in the graph is a basic-block or a call to a function. The shaded node in the fastpath leads into the slowpath.

Figure 1 shows the performance difference between the slowpath and the fastpath in seL4 – the fastpath is 5.6 times faster. Figure 2 (left) shows the control flow graph of the slowpath and offers some insight into why it takes so much time to perform precisely the same operation as the fastpath: it must handle all exceptional circumstances, which results in a larger cache footprint as well as incurring many branches and potential branch mispredictions.

Focusing on the common case, we obtain a fastpath with a control flow graph shown on the right of Figure 2. The shaded node represents a call out to the IPC slowpath. There are two loops to decode the address of the destination and to transfer the message data. A typical IPC in this fastpath would have almost no branching, except in the message copy. Even decoding the address requires only a single loop iteration in most systems. We use this fastpath as our starting point for further optimisation.

4 Optimisation techniques

There is a plethora of "collective wisdom" for optimising C code, often given as simple tips or rules which may allow a compiler to generate better performing code. These techniques should obviously not be applied blindly, as what performs better on some architectures may be detrimental on others. Additionally, many of these techniques, such as loop reversal and unrolling, are performed automatically by modern compilers.

Our improvements focused on using the gcc compiler (4.6.1), from Mentor Graphics's CodeBench Lite 2011.09-69. We also evaluated our improvements with ARM's own compiler (armcc 5.01).

The essence of our fastpath optimisation work is to analyse the compiled machine code from the C compiler and search for missed optimisation opportunities. Opportunities may arise in the form of pipeline stalls, redundant calculations, or sub-optimal data packing. We found that almost all of these can be resolved by modifications to the C code which give the compiler more scope for optimisation.

In doing this step-by-step comparison, we performed several simple optimisations which are commonly a part of the collective wisdom of optimisation, including:

- avoiding unnecessary use of `char`, `short`, and signed types to avoid superfluous sign-extension and zero-extension;
- avoiding unnecessary bit-masking, e.g. when it is known that unused bits will be zero;
- giving branch hints to achieve straight-line code for the common case;
- avoiding complex expressions which may result in many live registers, leading to stack spilling.

We also employed some lesser known techniques, described in the following sections, in order to assist the compiler. We note that even a "smarter" compiler could not have performed most of these optimisations automatically, as they depend on code invariants which cannot be detected by static code analysis.

4.1 Avoiding pipeline stalls

The IPC fastpath is heavily control-oriented – there are many conditional branches to ensure the conditions for the fastpath are satisfied. Many of these branches depend on values loaded from memory, and form load-test sequences that create pipeline stalls.

For example, Figure 3(a) lists a portion of the seL4 fastpath that contains consecutive branches. Each test depends on loading a value from memory. The generated assembly code is shown in Figure 3(c). On the ARM1136, each load instruction (`LDR`) has a 3-cycle latency for its result. As the results here in `r0` and `r3` are required immediately, the CPU is stalled for two cycles after both loads.

With knowledge of the pipeline and the intent of the C code, a human can observe that the second load can be issued earlier. This speculative loading cannot be performed by the compiler, as there is no hint to suggest that it is safe to do so – the validity of the second pointer could depend on the result of the branch.

By "lifting" the load for the second memory access above the first branch, we tell the compiler that it is safe to issue the memory access earlier. This is shown in Figure 3(b). The compiler may still choose to defer the load until it is required. However in this case, the compiler can see the optimisation opportunity to avoid the pipeline stall, and has sufficient spare registers which can be utilised. The resulting assembly code in Figure 3(d) has only one stall cycle, saving three cycles off the execution of the fastpath.

We discovered many places where this simple optimisation could be used in the fastpath, giving the compiler more flexibility to schedule instructions.

It should be noted that lifting accesses may not always offer better optimisation opportunities to the compiler. Due to the possibility of pointer aliasing, a compiler is not always able to safely reorder a read and a write to memory, even if the developer knows it to be safe.[1] Therefore, lifting a memory load may actually result in increased register pressure, so this optimisation should be used with care.

4.2 Expressing memory layout

In seL4, the thread object is composite, formed of two smaller objects which are positioned adjacently in memory – a *CNode* and a *thread control block* (TCB), each 256 bytes. Thread objects are always aligned to their size (512 bytes). Often, seL4 is required to access the CNode given a pointer to the TCB. The address for the CNode was computed by clearing the 8th bit of the TCB address.

This can be optimised by instead of clearing the 8th bit, simply subtracting 2^8. By doing so, the compiler is made aware of the actual memory layout of these objects, which it could not infer when we only cleared the bit (it is possible that the bit may not have been set in the first place).

[1]Many compilers support *strict aliasing*, which guarantees that pointers of different types will never overlap, however the possibility for aliases of pointers of the same type still exists. The *restrict* keyword can assist in excluding aliases in this case.

```
...

/* Check endpoint is not in send state. */
endpoint = *endpointPtr;
if ((endpoint & 0x3) == 0x1) goto slowpath;

/* Check that the caller cap is valid. */
callerCap = *callerCapPtr;
if (callerCap == 0) goto slowpath;
...
```

(a) A sample of C code without lifting optimisation.

```
...
endpoint = *endpointPtr;
callerCap = *callerCapPtr;

/* Check endpoint is not in send state. */
if ((endpoint & 0x3) == 0x1) goto slowpath;

/* Check that the caller cap is valid. */
if (callerCap == 0) goto slowpath;
...
```

(b) Code with `callerCap` load lifted.

```
ldr r0, [r4]
and r3, r0, #3
cmp r3, #1
beq slowpath
ldr r3, [ip, #-208]
cmp r3, #0
beq slowpath
```

(c) Generated assembly without lifting optimisation.

```
ldr r0, [r4]
ldr r3, [ip, #-208]
and r5, r0, #3
cmp r5, #1
beq slowpath
cmp r3, #0
beq slowpath
```

(d) Generated assembly with `callerCap` load lifted.

Figure 3: A sample code snippet where explicitly lifting memory accesses for the compiler aids optimisation.

In particular, it can optimise memory loads from the CNode given the TCB address by negatively-indexing the TCB address when performing the memory load. In ARM assembly, this can be expressed as `LDR r0, [r1, #-256]`, where `r1` is the TCB address. Although this only saves one cycle, it also reduces register pressure, allowing the compiler to use the register for other optimisations.

4.3 Usage of inline assembly

We use assembly code *only* for hardware-specific operations which cannot be expressed in C. As the IPC fastpath entails a context switch, it requires accessing CPU-specific registers which have no C equivalent. By utilising *inline* assembly instead of function calls to external assembly routines, the compiler is not constrained to use the C ABI at these boundaries. This allows for better register allocation, stack usage and alias analysis. As we inline all assembly routines, including the return to userspace, the resulting code has no branches in the case of a 0-length IPC.

4.4 Limitations of compiled C

Almost all of the optimisations that we were able to identify could be expressed equivalently at the C level. Some further optimisations required using assembly, yet did not give a measurable gain to the IPC fastpath. In particular, we were able to remove the need for a valid stack. This potentially saves several cycles, by avoiding unnecessary register loads and memory accesses, and reduces register pressure. However, despite removing all these superfluous instructions, the overall cycle count of the IPC fastpath was not reduced, as removing these instructions left bubbles in the pipeline where it was already stalled.

The extra register was not useful in the fastpath either, as there were already two registers going unutilised. For code with more register pressure, the results may be quite different.

All attempts to further reduce the cycle count of the IPC fastpath resulted in changes which could easily be expressed in C. There still remained 14 cycles in which the pipeline was stalled due to data dependencies, however it became increasingly difficult to eliminate these stalls without significantly penalising non-fastpath IPC operations. Although we do not claim our final assembly to be optimal, the time spent optimising it further had well and truly reached the point of diminishing returns.

5 Evaluation

We evaluated the results of our optimisations by measuring the execution time on an ARM11 core on the Freescale i.MX31 processor. We used the performance monitoring unit to measure precise cycle counts for 160 000 iterations of a ping-pong bench-

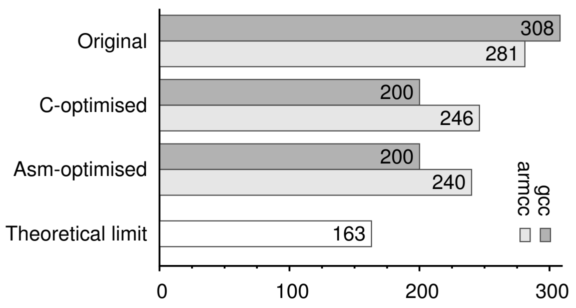

Figure 4: Cycle counts of a one-way IPC via the fastpath for a 0-length IPC message between threads in different address spaces.

mark between two address spaces. We computed the one-way IPC as half the average round-trip time. The results are shown in Figure 4.

The "theoretical limit" is what we would achieve if we could eliminate all unnecessary pipeline stalls in the existing assembly, assuming it is otherwise optimal. There are practical limitations to achieving this, but this number is a lower bound, given the design of seL4. The difference between "C-optimised" and "Asm-optimised" reflects optimisations which could not be performed at the C level (e.g. discarding the stack and repurposing the stack pointer).

The best results were obtained using the gcc compiler. Although armcc often generates better optimised code than gcc, it was unable to optimise our fastpath as effectively. We found that armcc did not order code optimally, despite hints using `__builtin_expect()`. As a result, there were 7 more branch mispredictions in the armcc version compared to the result from gcc. These mispredictions account for over 90 % of the difference between gcc and armcc fastpaths.

We also measured the effect of compiling seL4 for ARM's size-optimised Thumb instruction set. Thumb reduced the size of the compiled fastpath (in bytes) by 20 %, but increased the cycle count by over 80 %, as significantly more instructions were needed.

6 Discussion

Using optimisations at the C level has allowed us to reduce the one-way IPC times by 35 %. Using modifications to the assembly code, we were able to remove superfluous instructions, but we were unable to reduce the execution time further. Given our experiences, we claim that for heavily control-oriented code such as the fastpath, and for our register-rich RISC platform, human-guided compilers can achieve almost as good a result as hand-optimised assembly.

Our experiments were performed on a single-issue pipeline, common to many embedded systems. Multiple-issue (e.g. superscalar or VLIW) pipelines pose interesting optimisation challenges both for compilers and humans.

There may be situations where the compilers are not aware of (or are unable to generate) instructions that would lead to more optimised output. Inline assembly may be able to assist in some of these cases, whilst keeping the majority of the code in C.

Optimisation effort We dedicated around two person-weeks of work to optimising the C fastpath. Like many optimisation efforts, we achieved the majority of our gains within the first 30 % of the work, rapidly reaching the point of diminishing returns. We estimate that optimising an assembly implementation would require at least twice as much effort, and significantly increase the subsequent maintenance burden.

Maintenance There are two distinct aspects to maintaining code such as the fastpath: performance and correctness. One disadvantage of a C-optimised fastpath is that the performance is highly sensitive to changes in the compiler. Vigilant performance regression testing is required to ensure that the optimisation efforts do not bit rot.

However, we claim that C code is significantly easier to understand, and to maintain correctness of, than the equivalent assembly implementation. Changes to data structures are much easier to incorporate into C than in assembly.

Portability The majority of our optimisations do not target a specific architecture or compiler – they simply present further optimisation opportunities for the compiler. For example, if there existed a super-optimiser capable of scaling to produce optimal machine code for the fastpath, it too would be assisted by our optimisations.

Verification The IPC fastpath is a verified part of the seL4 microkernel – i.e., there is a machine-checked formal proof that it adheres to the functional specification and preserves all invariants within the kernel. By limiting our optimisations to the C code, we retain the ability to verify the fastpath.

Cache layout The majority of IPC benchmarks are focused on hot-cache performance. Ideally, all memory accesses are in the L1 cache, which on our platform can be accessed in a single cycle. However, even for hot-cache performance, pathological memory layouts may lead to conflict misses and significantly degrade performance.

There are approximately 120 unique instructions executed by a one-way IPC, giving an instruction-cache footprint of 18 cache lines. Due to the compactness of the IPC fastpath, and the ease with which code placement in the microkernel can be manipulated, it is simple to avoid instruction-cache conflicts.

7 Conclusion

We have demonstrated that for heavily control-oriented code, it is possible to perform significant optimisations at the C level. Modern compilers are sufficiently advanced that they can recognise many optimisation opportunities. Where optimisations are missed by the compiler, it is often due to a lack of insight available to the compiler, and can be resolved through modifications to the C sources.

Using carefully guided optimisations on the IPC fastpath, we were able to attain all of the possible optimisations we could conceive without resorting to hand-crafting assembly code. On RISC platforms such as ours, the need to write fastpaths in assembly to get maximum performance is questionable when modern compilers can achieve similar results in C.

Although these are preliminary results on two specific code paths, we believe that the techniques here can generalise to hot code paths in both kernels and general userspace code.

In order to achieve these optimisations, we have had to gain an intricate understanding of the details of the microarchitecture and closely inspect the generated assembly code. This is no different to the depth of knowledge required to optimise an assembly implementation. However, by performing optimisations at the C level, we retain expressiveness, reduce maintenance overhead, and can make use of current formal verification techniques.

Acknowledgements

We thank James Wilmot for his assistance with benchmarking seL4's IPC fastpaths.

NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

References

[GCC+05] Charles Gray, Matthew Chapman, Peter Chubb, David Mosberger-Tang, and Gernot Heiser. Itanium — a system implementor's tale. In *2005 USENIX*, pages 264–278, Anaheim, CA, USA, Apr 2005.

[Hyd] Randall Hyde. The great debate. http://webster.cs.ucr.edu/Page_TechDocs/GreatDebate/.

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *22nd SOSP*, pages 207–220, Big Sky, MT, USA, Oct 2009. ACM.

[L4H] The L4 headquarters. http://l4hq.org.

[L4Impl] Implementations of the L4 μ-kernel interface. http://os.inf.tu-dresden.de/L4/impl.html.

[LES+97] Jochen Liedtke, Kevin Elphinstone, Sebastian Schönberg, Herrman Härtig, Gernot Heiser, Nayeem Islam, and Trent Jaeger. Achieved IPC performance (still the foundation for extensibility). In *6th HotOS*, pages 28–31, Cape Cod, MA, USA, May 1997.

[Lie93] Jochen Liedtke. Improving IPC by kernel design. In *14th SOSP*, pages 175–188, Asheville, NC, USA, Dec 1993.

[Mas87] Henry Massalin. Superoptimizer: a look at the smallest program. *SIGARCH Computer Architecture News*, Oct 1987.