Correct, Fast, Maintainable: Choose Any Three!

Bernard Blackham and Gernot Heiser

NICTA and University of New South Wales
Sydney, Australia
What programmers want

- Speed
- Sanity
- Correctness

Utopia?
What programmers do

Write

Debug

Fast enough?

Yes!

Pub (ᄒ포)!}

No :(

Optimise
What programmers do

- Write
- Debug
- Fast enough?
  - Yes!
    - Pub (').'!
  - No :(
    - Optimise
    - Rewrite in assembly
When to use assembly?

Programmers (should) use assembly as a last resort when:

- higher level languages are not sufficient
- they have the know-how to do better in assembly
- they have little mercy for those who follow
When to use assembly?

Programmers (should) use assembly as a last resort when:
- higher level languages are not sufficient
- they have the know-how to do better in assembly
- they have little mercy for those who follow
When to use assembly?

Programmers (should) use assembly as a last resort when:

- higher level languages are not sufficient
- they have the know-how to do better in assembly
- they have little mercy for those who follow

We claim:

- Modern compilers are often smarter than we think
- Many assembly optimisation techniques can be performed in C
A brief history of L4...

• L4 microkernels emphasise fast IPC message-passing

• IPC performance is the Master

  Anything which may lead to higher IPC performance has to be discussed.

  In case of doubt, decisions in favour of IPC have to be taken.

A brief history of L4...

1993
First L4 kernel written entirely in assembly

1999
L4Ka::Hazelnut written in C/C++
IPC fastpaths hand-coded in assembly
A brief history of L4...

1993
First L4 kernel written entirely in assembly

1999
L4Ka::Hazelnut written in C/C++
IPC *fastpaths* hand-coded in assembly

**Fastpaths**: dedicated code path that optimises the common-case
The “slowpath”
A brief history of L4...

1993
First L4 kernel written entirely in assembly

1999
L4Ka::Hazelnut written in C/C++
IPC *fastpaths* hand-coded in assembly

...  

2007
More L4 μ-kernels, all C/C++ with fastpaths in assembly

seL4 implementation written in C
IPC fastpaths written in C
How much performance does a fastpath in C compromise over a fastpath in assembly?

2007

seL4 implementation written in C
IPC fastpaths written in C
How to optimise C code?

Determine correspondence between C and assembly:

```c
void
fastpath_call(word_t cptr, word_t msgInfo)
{
    message_info_t info = message_info_set_msgCapsUnwrapped(
        messageInfoFromWord_nolencheck(msgInfo), 0);
    uint32_t length = message_info_get_msgLength(info);
    uint32_t fault_type = fault_get_faultType(curThread->fault);

    /* Check there's no extra caps, the length is ok and
    * there's no saved fault. */
    if (unlikely(fastpath_mi_check(msgInfo) ||
        fault_type != fault_null_fault)) {
        slowpath(SysCall);
    }

    /* Lookup the cap */
    cap_t ep_cap = lookup_fp(
        TCB_PTR_CTE_PTR(ksCurThread, tcbCTable)->cap, cptr);

    /* Check it's an endpoint */
    if (unlikely(cap_get_capType(ep_cap) != cap_endpoint_cap ||
        !cap_endpoint_cap_get_capCanSend(ep_cap))) {
        slowpath(SysCall);
    }

    /* Check correspondence between C and assembly */
    f001003c <fastpath_call>:
        ldr r2, [pc, #500] ; f0010238 <fastpath_call+0x1fc>
        lsl r3, r1, #23
        cmp r3, #33554432   ; 0x2000000
        ldr ip, [r2]
        push   {r7, lr}
        ldr r3, [ip, #84]   ; 0x54
        bhi f0010230 <fastpath_call+0x1f4>
        ands    r3, r3, #7
        bne f0010230 <fastpath_call+0x1f4>
        ldr r4, [ip, #-256] ; 0xffffff00
        ldr lr, [ip, #-252] ; 0xffffff04
        and r5, r4, #15
        bne f0010230 <fastpath_call+0x1f4>

    /* Look up the fault */
   _fault_t fault = fault_lookup(tcbCTable, fault_type);

    /* Look up the next cap */
    ep_cap = lookup_fp(fault->cap);

    /* Check it's an endpoint */
    if (unlikely(cap_get_capType(ep_cap) != cap_endpoint_cap ||
        !cap_endpoint_cap_get_capCanSend(ep_cap))) {
        slowpath(SysCall);
    }

    /* Check correspondence between C and assembly */
    f001003c <fastpath_call>:
        ldr r2, [pc, #500] ; f0010238 <fastpath_call+0x1fc>
        lsl r3, r1, #23
        cmp r3, #33554432   ; 0x2000000
        ldr ip, [r2]
        push   {r7, lr}
        ldr r3, [ip, #84]   ; 0x54
        bhi f0010230 <fastpath_call+0x1f4>
        ands    r3, r3, #7
        bne f0010230 <fastpath_call+0x1f4>
        ldr r4, [ip, #-256] ; 0xffffff00
        ldr lr, [ip, #-252] ; 0xffffff04
        and r5, r4, #15
        bne f0010230 <fastpath_call+0x1f4>
```

For each instruction, ask:

- Why is it there? (what C code generated it?)
- Is it needed? (either superfluous or redundant?)
- Is it causing any pipeline stalls?
- Can the same effect be achieved faster?
- What prevents the compiler from doing so?
Simple examples

• Unnecessary sign-extension, zero-extension
  – Caused by using char, short and signed types unnecessarily

• Unnecessary bit-masking

• Unnecessary branches
  – Use compiler branch hints to optimise code placement

• Stack spilling
  – Re-write complex expressions
    (help with common subexpression elimination)

• Function calls (e.g. for machine-specific assembly)
More examples: avoiding pipeline stalls

... 

/* 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;

...
More examples: avoiding pipeline stalls

... 
/* 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;

...
More examples - avoiding pipeline stalls

... 

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;

...
More examples – avoiding pipeline stalls

... 

```c
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;
```

...
Was it worth it?

Time for one-way IPC via fastpath

- Original: 308 cycles (gcc 4.6.1) vs 281 cycles (armcc 5.01)
- C-optimised: 246 cycles (gcc 4.6.1) vs 240 cycles (armcc 5.01)
- Asm-optimised: 206 cycles (gcc 4.6.1) vs 200 cycles (armcc 5.01)
- OKL4 (assembly): 163 cycles
- Theoretical limit: 163 cycles
Was it worth it?

• Optimisation effort comparable, if not easier

• Performance is compiler dependent
  – Optimisations can be used by all compilers

• Correctness is obscured
  – Still better than assembly

• Source is less maintainable?
  – comments are good!
Some limitations apply*

* Our claims hold for:

- RISC instruction sets
- Single-issue pipeline
- Register-rich architecture
- Heavily control-oriented code

Room for more optimisation:

- repurpose sacred registers
- discarding stack completely
Lessons learnt

• Modern compilers are awesome!

• Verification and performance are not necessarily at odds

• Applying assembly optimisation techniques to C sources can achieve near-optimal results*

bernard.blackham@nicta.com.au