Correct, Fast, Maintainable: Choose Any Three!

NICTA

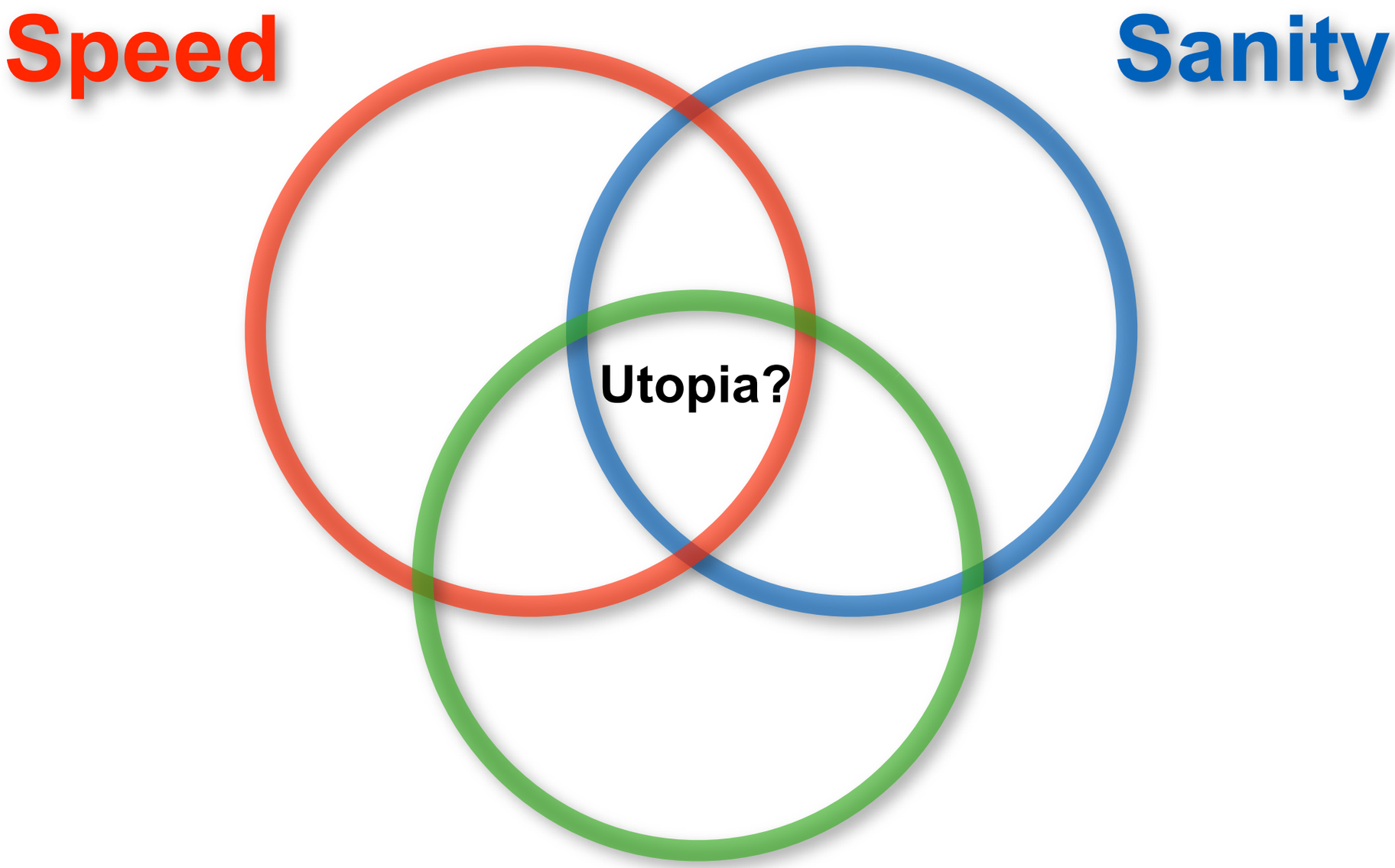
Bernard Blackham and Gernot Heiser

NICTA and University of New South Wales
Sydney, Australia

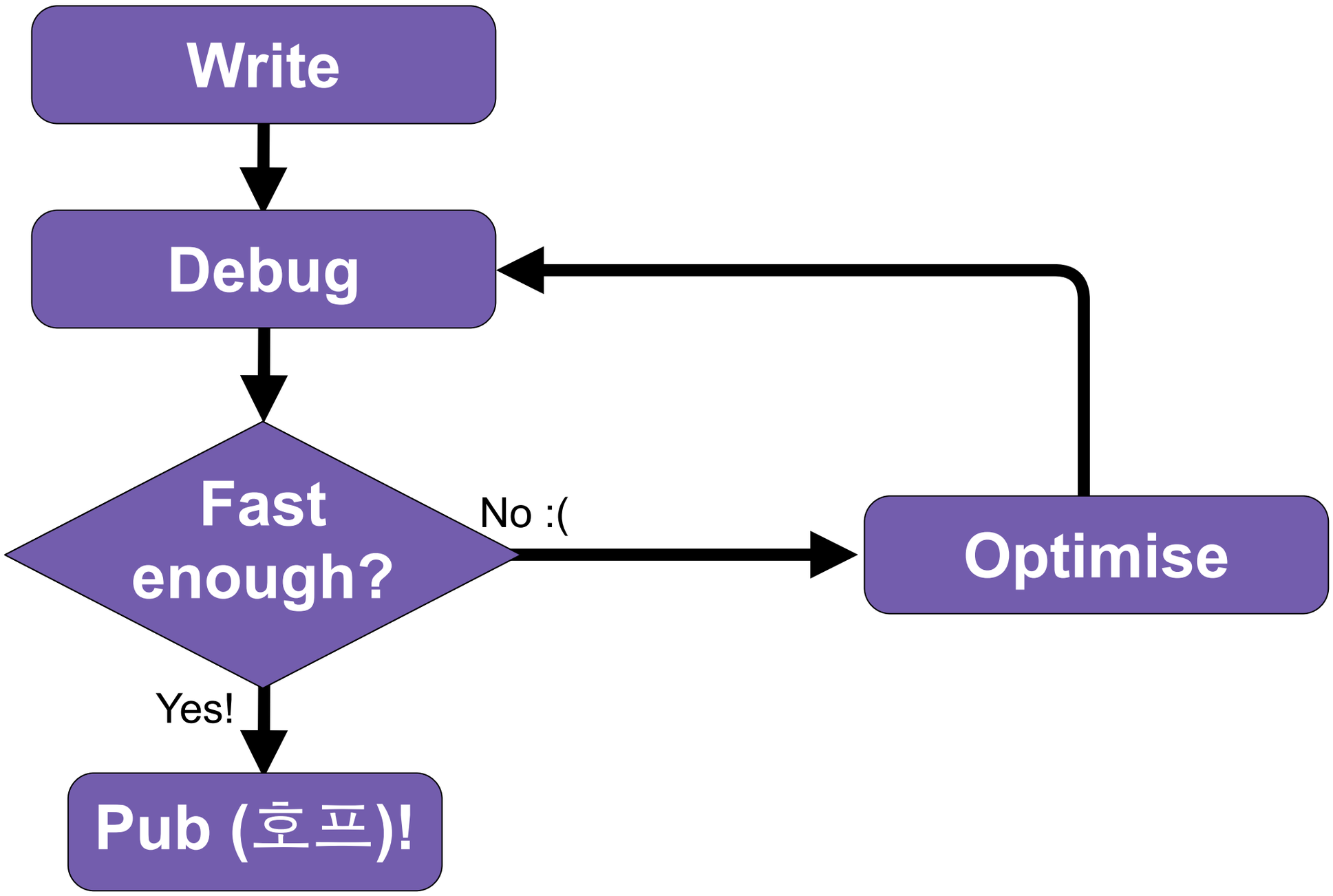
Australian Government
Department of Broadband, Communications and the Digital Economy
Australian Research Council

NICTA Funding and Supporting Members and Partners

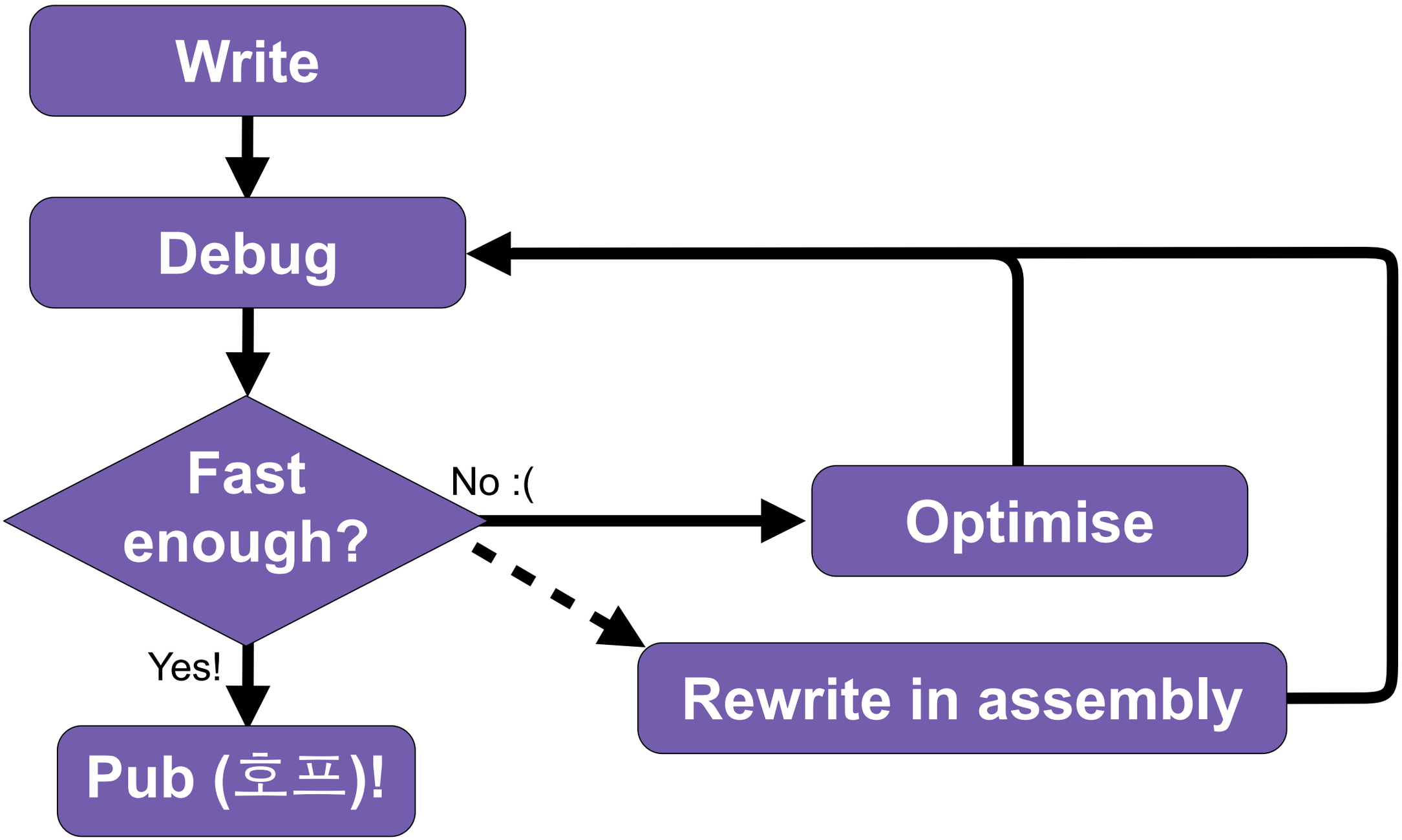
What programmers want



What programmers do



What programmers do

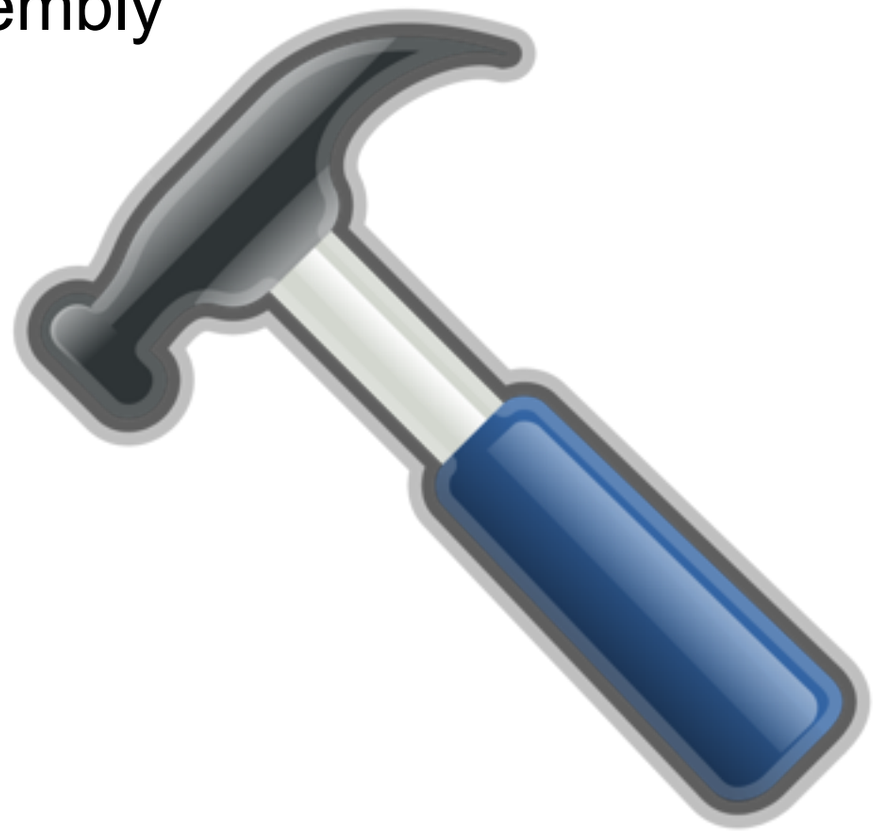


When to use assembly?

NICTA

Programmers (should) use assembly as as 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

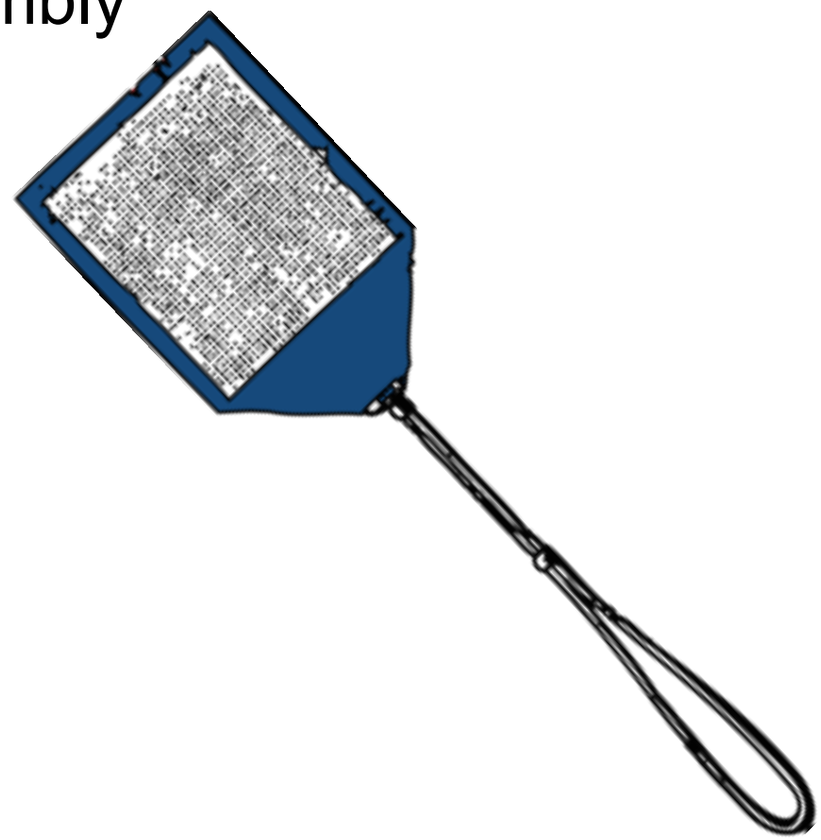


When to use assembly?

NICTA

Programmers (should) use assembly as as 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

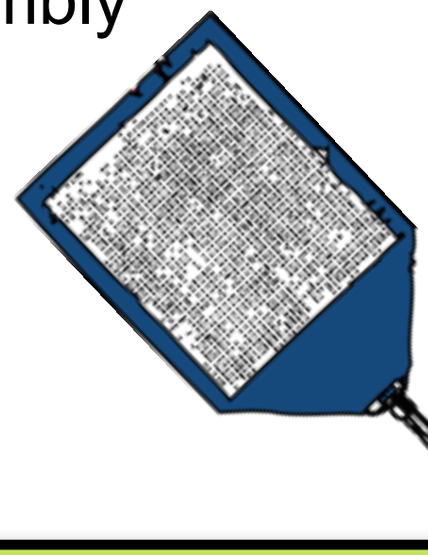


When to use assembly?

NICTA

Programmers (should) use assembly as as 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...

NICTA

- L4 microkernels emphasise ***fast*** IPC message-passing

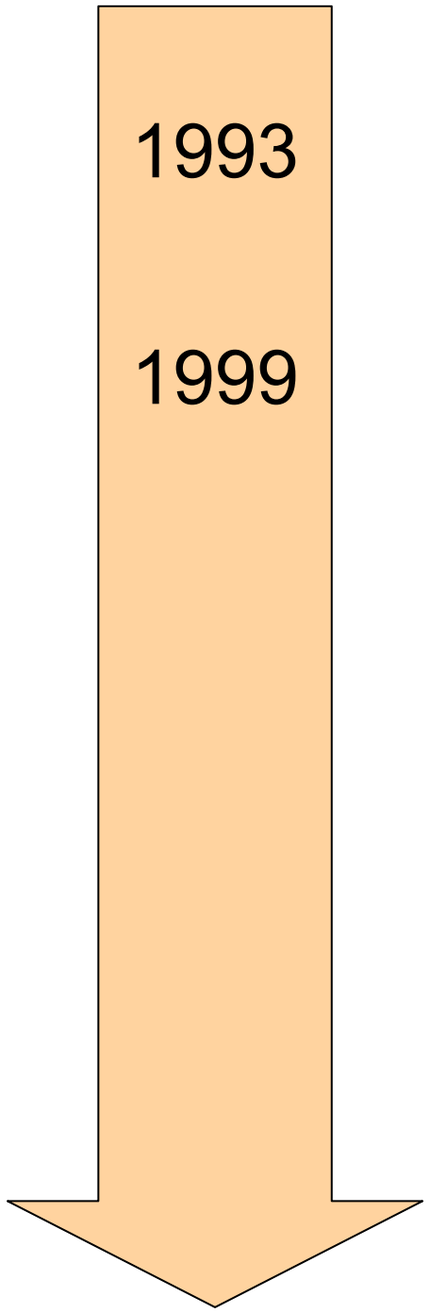
> - IPC performance is the Master
>
> Anything which may lead to higher IPC performance has to be discussed.
>
> In case of doubt, decisions in favour of IPC have to be taken.
>
> – Jochen Liedtke (1993)

A brief history of L4...

NICTA

1993 First L4 kernel written entirely in assembly

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



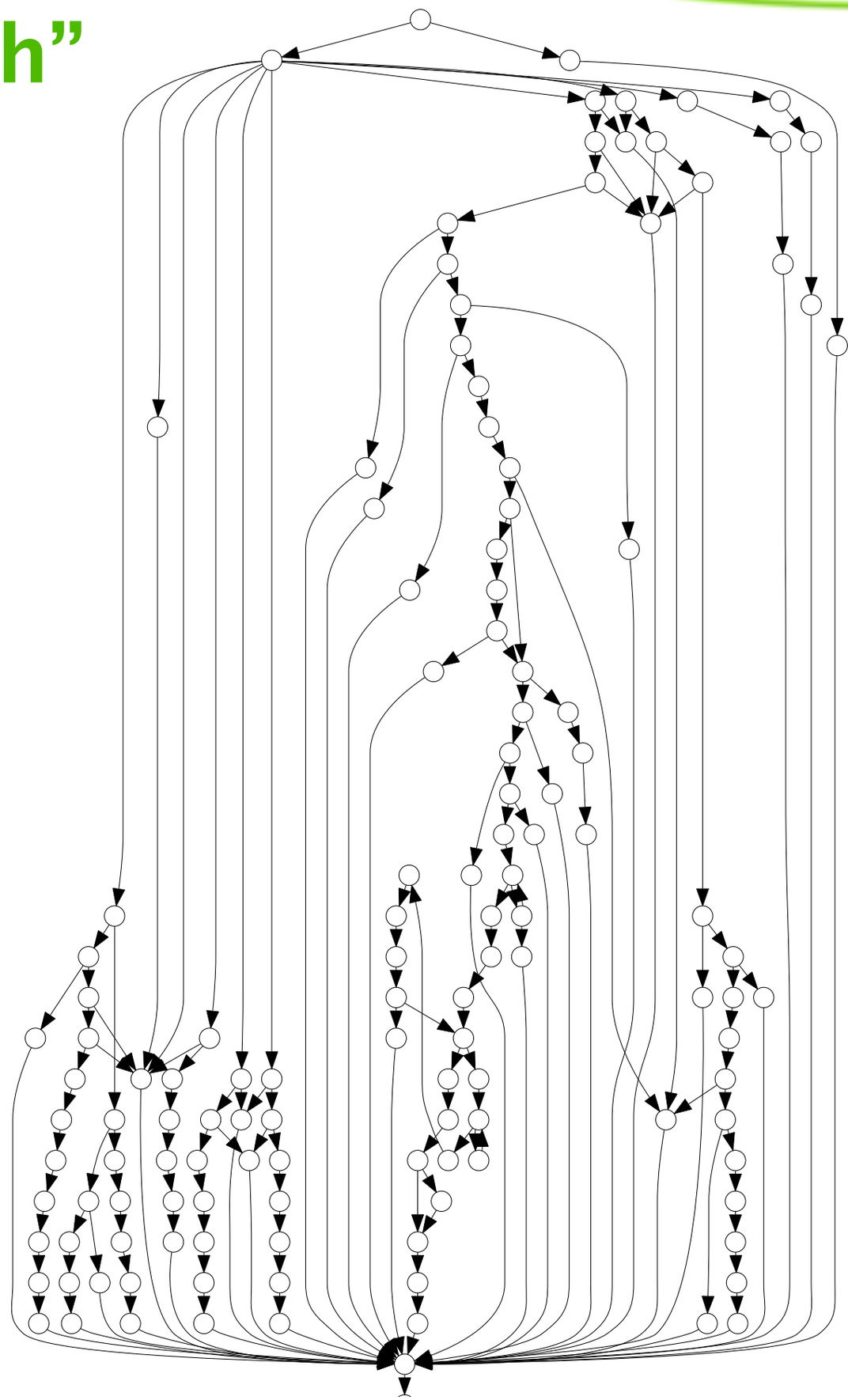
A brief history of L4...

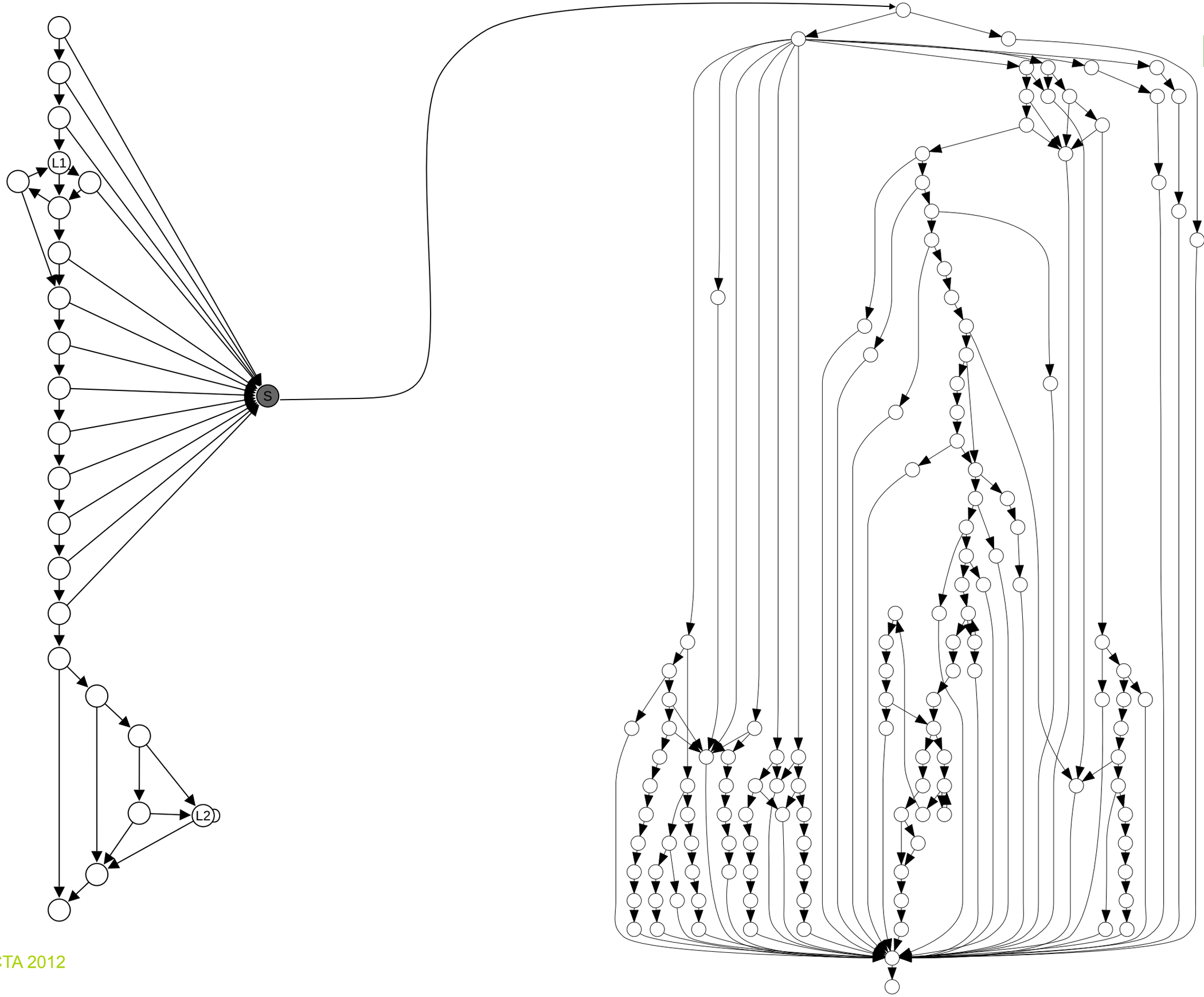
1993 First L4 kernel written entirely in assembly

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

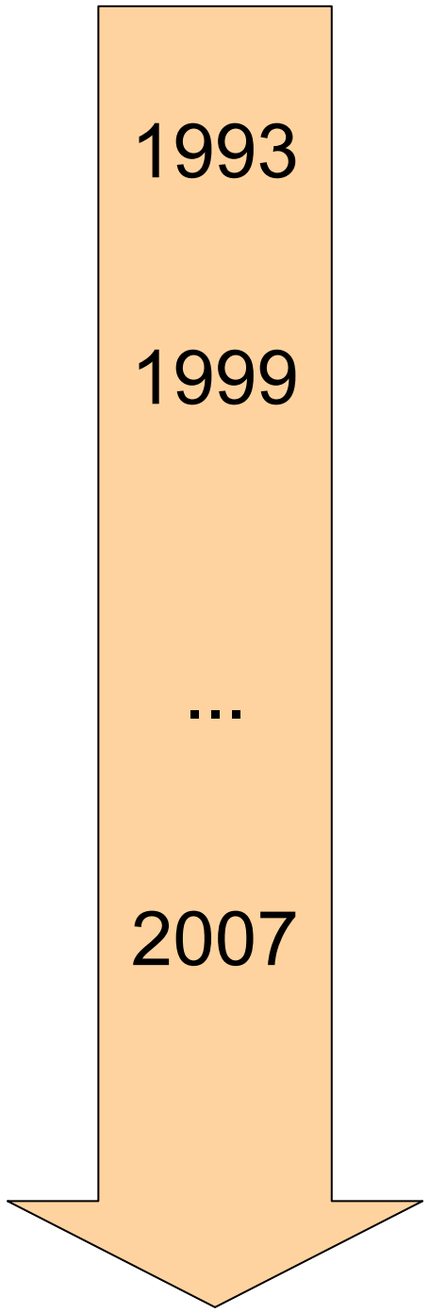
Fastpaths: dedicated code path that optimises the common-case

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

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

2007 — seL4 implementation written in C
IPC fastpaths written in C

A brief history of L4...

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?

NICTA

Determine correspondence between C and assembly:

```
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);
```

```
f001003c <fastpath_call>:
f001003c:   e59f21f4    ldr r2, [pc, #500]  ; f0010238 <fastpa
f0010040:   e1a03b81    lsl r3, r1, #23
f0010044:   e3530402    cmp r3, #33554432   ; 0x2000000
f0010048:   e592c000    ldr ip, [r2]
f001004c:   e92d4080    push    {r7, lr}
f0010050:   e59c3054    ldr r3, [ip, #84]   ; 0x54
f0010054:   8a000075    bhi f0010230 <fastpath_call+0x1f4>
f0010058:   e2133007    ands    r3, r3, #7
f001005c:   1a000073    bne f0010230 <fastpath_call+0x1f4>
f0010060:   e51c4100    ldr r4, [ip, #-256] ; 0xffffff00
f0010064:   e51ce0fc    ldr lr, [ip, #-252] ; 0xffffff04
f0010068:   e204500f    and r5, r4, #15
f001006c:   e3550005    cmp r5, #5
f0010070:   1a00006e    bne f0010230 <fastpath_call+0x1f4>
f0010074:   e20e653e    and r6, lr, #260046848  ; 0xf800000
f0010078:   e1a07310    lsl r7, r0, r3
f001007c:   e1b05ba6    lsrs    r5, r6, #23
f0010080:   0a000004    beq f0010098 <fastpath_call+0x5c>
f0010084:   e2658020    rsb r8, r5, #32
```

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

NICTA

- 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

NICTA

```
...

/* 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;

...
```

```
ldr r0, [r4]
and r3, r0, #3
cmp r3, #1
beq slowpath

ldr r3, [ip, #-208]
cmp r3, #0
beq slowpath
```

More examples: avoiding pipeline stalls

NICTA

stall x 2

```
...

/* 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;

...
```

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

stall x 2

More examples – avoiding pipeline stalls

NICTA

```
...

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;

...
```

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

More examples – avoiding pipeline stalls

NICTA

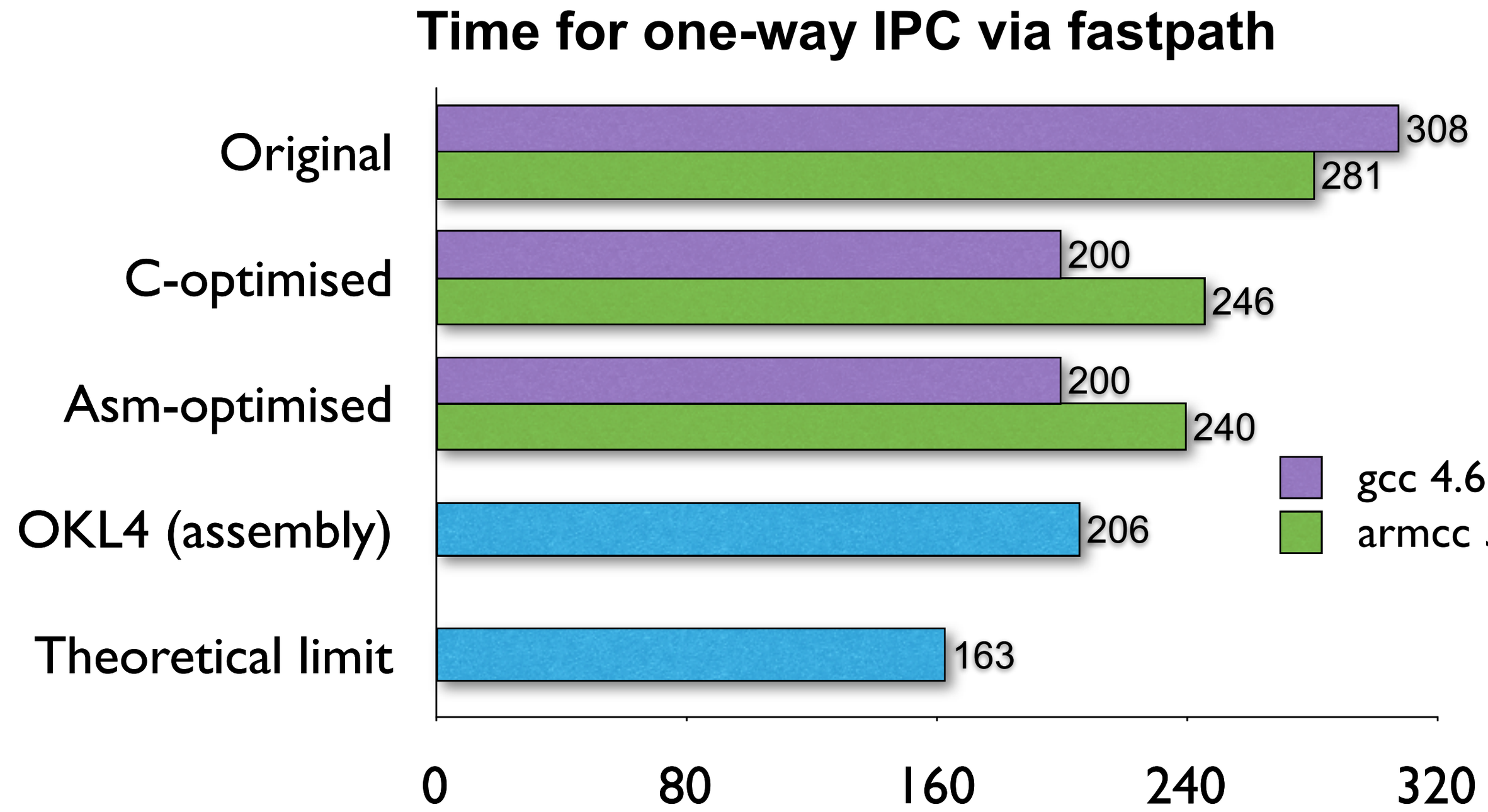
stall x 1

```
...

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;

...
```

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

Was it worth it?

NICTA



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*

NICTA

* 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