Improving Interrupt Response Time in a Verifiable Protected Microkernel

NICTA

Bernard Blackham
Yao Shi
Gernot Heiser

The University of New South Wales & NICTA, Sydney, Australia

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

NICTA Funding and Supporting Members and Partners

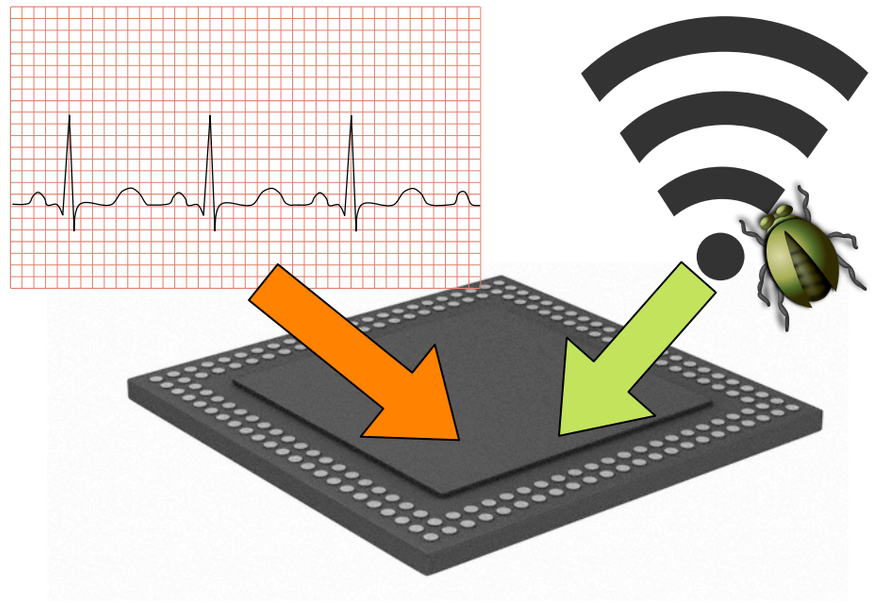
EuroSys 2012

Motivation

NICTA

The desire to build systems which are:

- **hard real-time**
  - because application domains demand it
- **mixed-criticality**
  - necessary to remain competitive
- **trustworthy!**
  - bugs cost money, embarrassment and possibly life.



e.g. medical implants, industrial automation, some automotive systems

Motivation

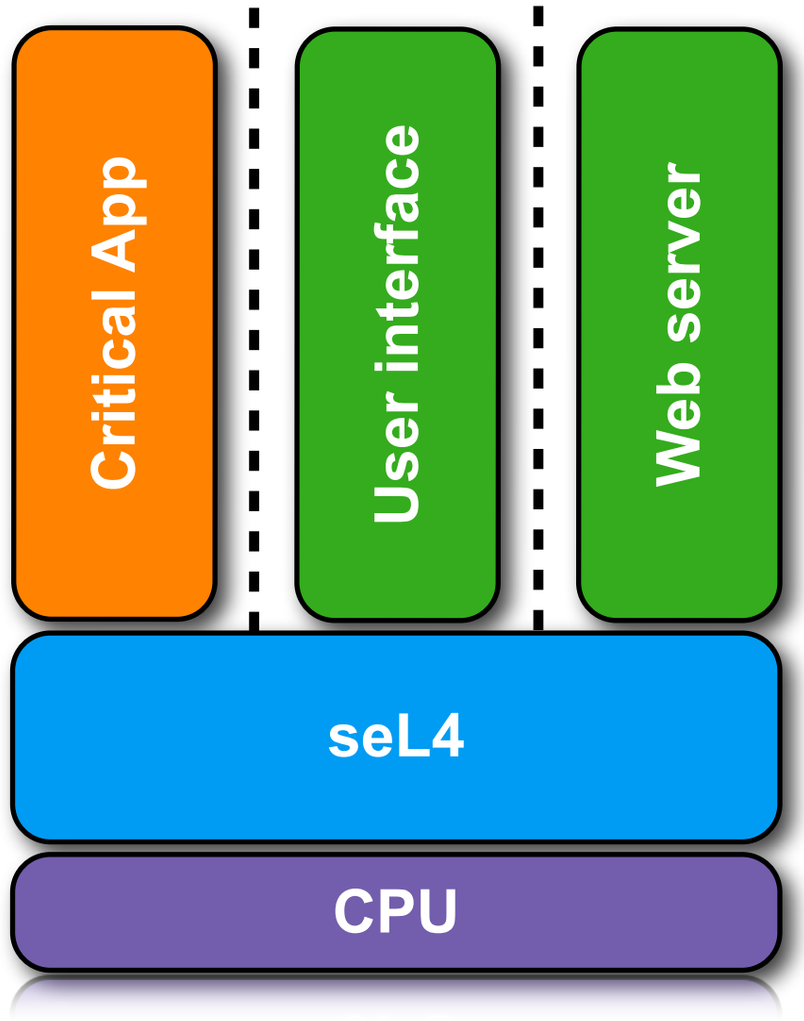
seL4 microkernel gives *trustworthiness* using

- MMU-based isolation
- Small trusted computing base

★ Formal specification of functional behaviour

★ Machine-checked formal proof of compliance to specification

[Klein et. al., SOSP'09]



Background

- Previously shown that interrupt latency can be computed on a formally verified kernel.

[Blackham et. al., RTSS 2011]

- Formal verification (today) requires a non-preemptible kernel.

- Interrupt latencies of several milliseconds!

Verification of RT kernel design

NICTA

Real-time demands often conflict with ease of verification

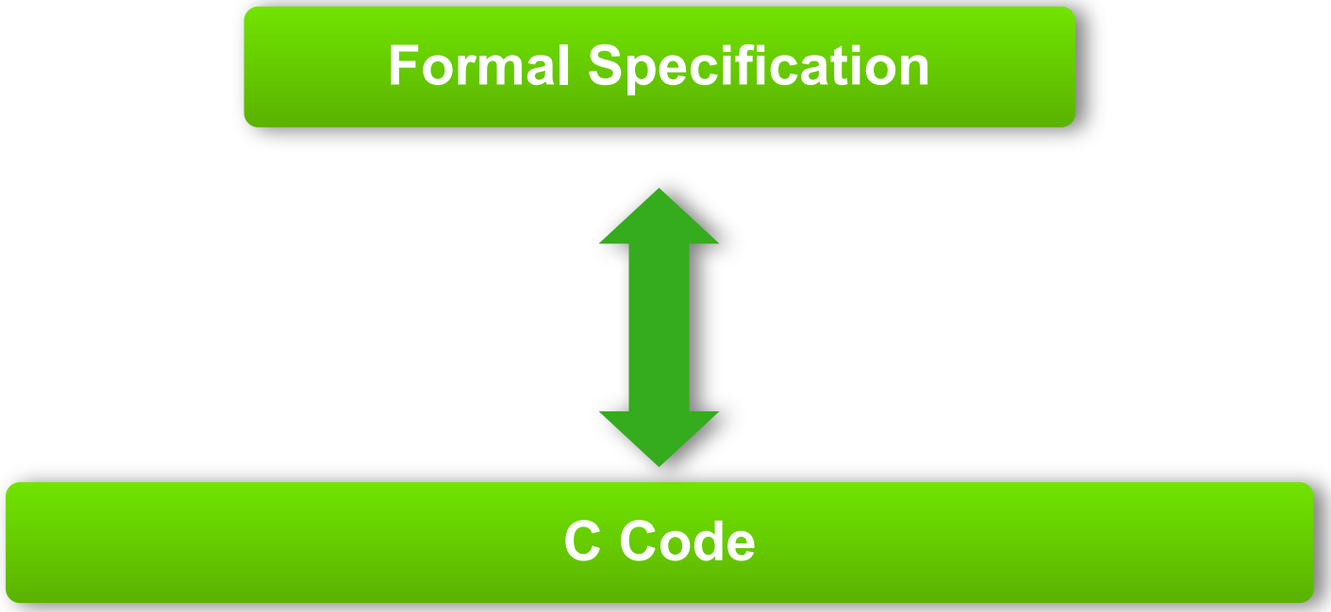
– e.g. preempted operations leave interesting intermediate states

How can we improve interrupt latency with minimum impact on:

- verification?
- overall performance?
- operational semantics?

seL4: a formally verified kernel

NICTA

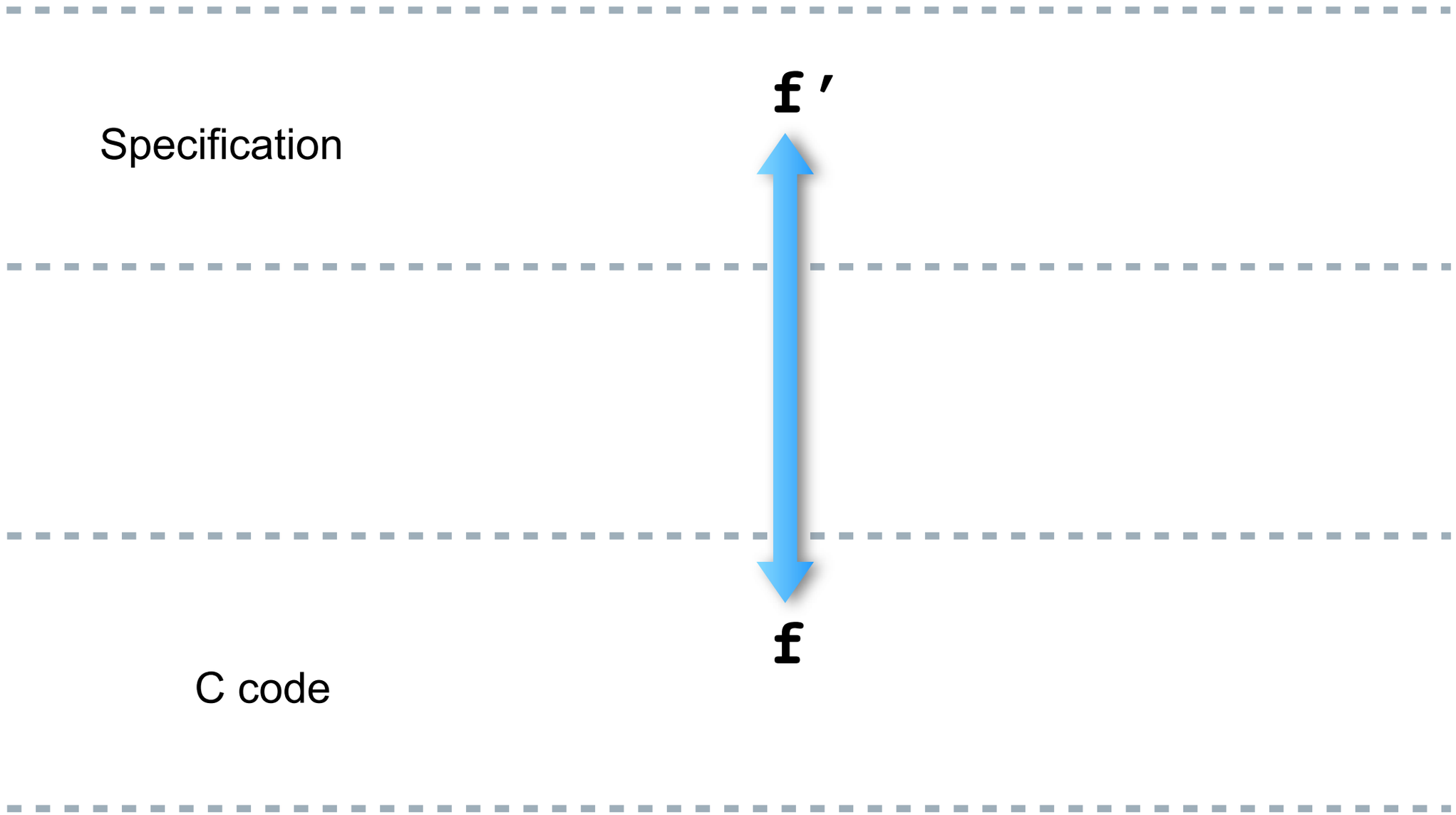


8,700 lines of C

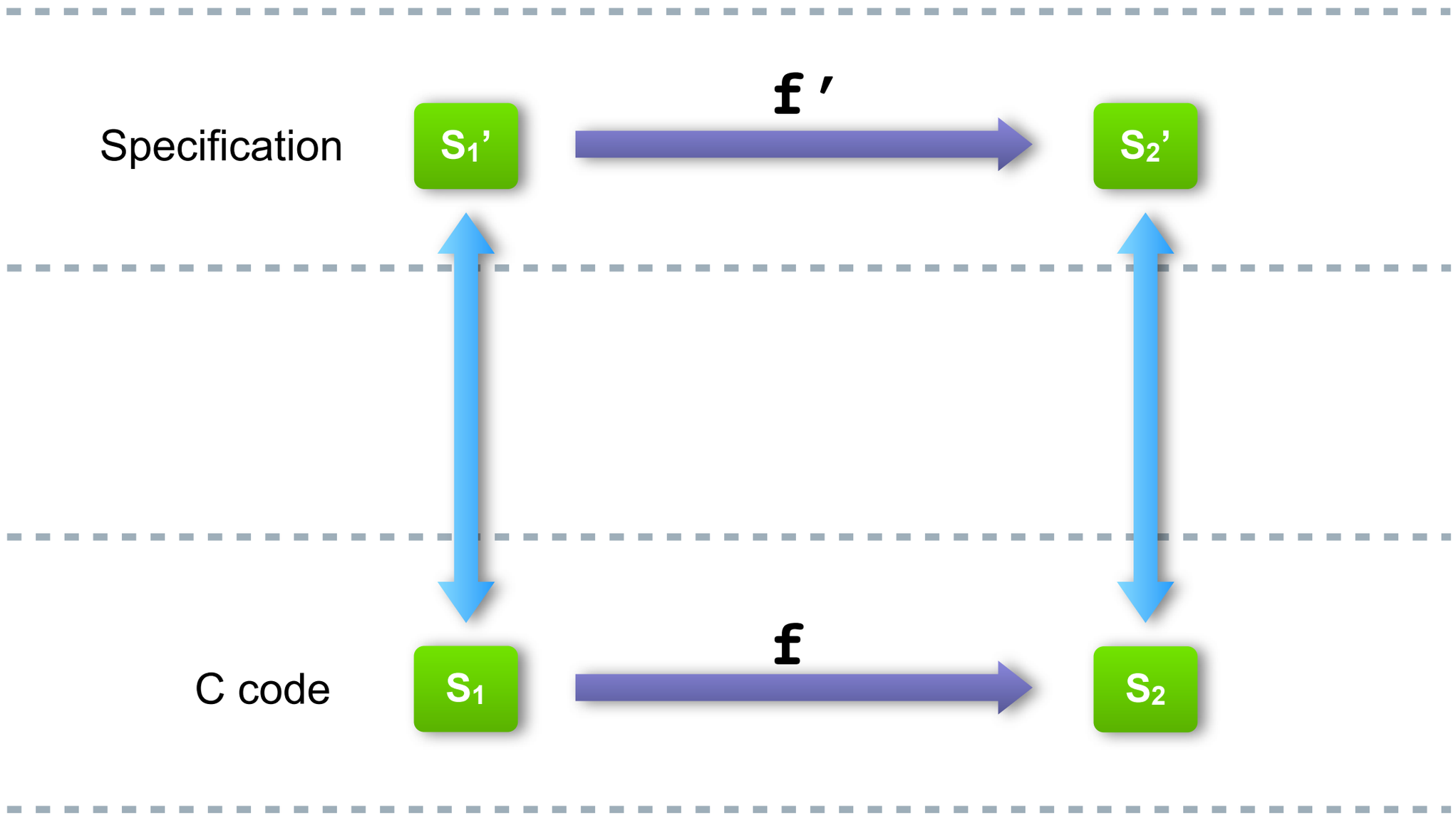
200,000 lines of proof

25 person-years

seL4 proof structure



seL4 proof structure



Kernel execution models

NICTA

Event-based, single kernel stack

- **Total kernel state is encapsulated within objects**
- All preemption is explicit
- No locks ⇒ better average-case performance

Process-based, per-thread kernel stack

- **Total kernel state includes both objects and stack contents**
- Preemption occurs anywhere that is not guarded by locks
- Locking degrades average-case performance

seL4 proof structure

NICTA

Proof shows that all kernel operations maintain global invariants

{invs} op {invs}

80% of the properties proven show that invariants are maintained

⇒ Don't break them!

(Unless you absolutely have to)

Common design patterns

“Incremental consistency”

- large composite objects are composed of individual components that can be added or deleted one at a time
- i.e. operations can be decomposed into multiple O(*small*) steps
- simple invariants at intermediate steps

Example: aborting IPC

➜ For each waiting thread,
Dequeue thread from endpoint and restart it

Example: aborting IPC

Disable the endpoint

➜ For each waiting thread,

Dequeue thread from endpoint and restart it

If interrupt pending, abort

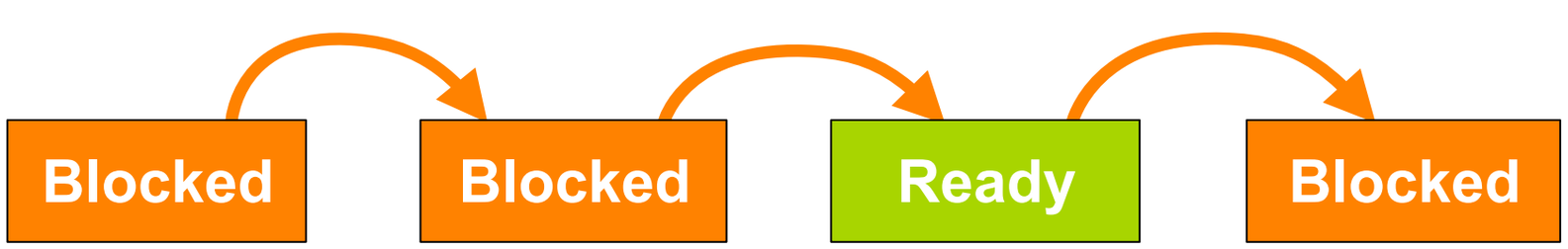
Example: lazy scheduling

NICTA

Frequent IPC leads to:

⇒ threads frequently blocking/unblocking

⇒ lots of run-queue manipulation



Lazy scheduling leaves blocked threads in run queue

- Assume threads will unblock before scheduler walks run queue
- Used first in L3 by Liedtke, and in almost all L4 kernels since

Example: lazy scheduling

```
tcb_t chooseThread(void) {
  foreach prio ∈ prios
    foreach thread ∈ runQueue[prio]
      if runnable(thread)
        return thread
      else
        schedDequeue(thread)
}
```

Replacement: “Benno scheduling”

NICTA

- Every thread on the run queue is runnable
- Every runnable thread (except the active thread) is on the run queue

Context switches due to IPC involve no run-queue manipulation

```
tcb_t chooseThread(void) {
  foreach prio ∈ prios
    thread = runQueue[prio].head
    if thread != NULL
      return thread
}
```

Replacement: "Benno scheduling"

NICTA

Invariant #1:

- Every thread on the run queue is runnable

Invariant #2:

- Every runnable thread (except the active thread) is on the run queue

```
tcb_t chooseThread(void) {
  foreach prio ∈ prios
    thread = runQueue[prio].head
    if thread != NULL
      return thread
}
```

Replacement: "Benno scheduling"

NICTA

Invariant #1:

- Every thread on the run queue is runnable

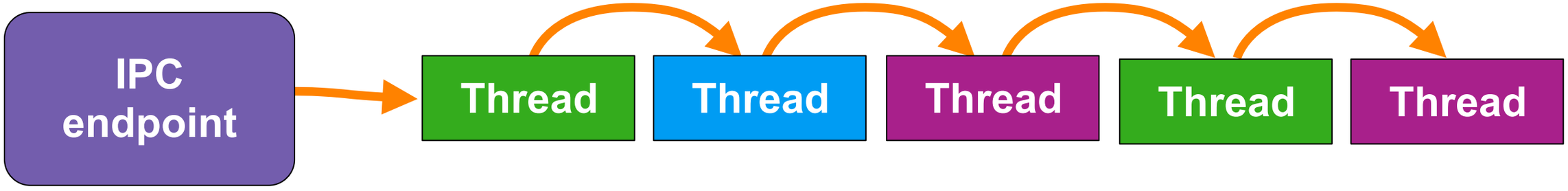
Invariant #2:

- Every runnable thread (except the active thread) is on the run queue

... which must be proven when:

➜ a thread is put on the run queue
➜ a thread's state is changed
➜ the active thread is changed

“Badged” IPC endpoint deletion



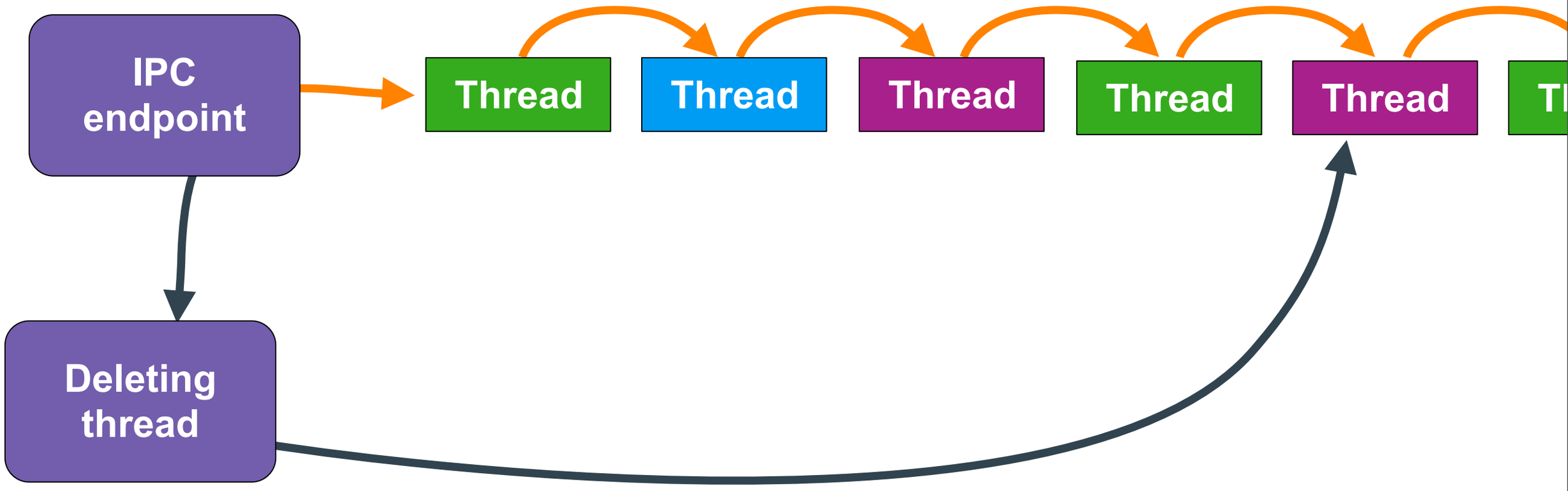
➜ For each waiting thread,

Does the thread use the badge being deleted?

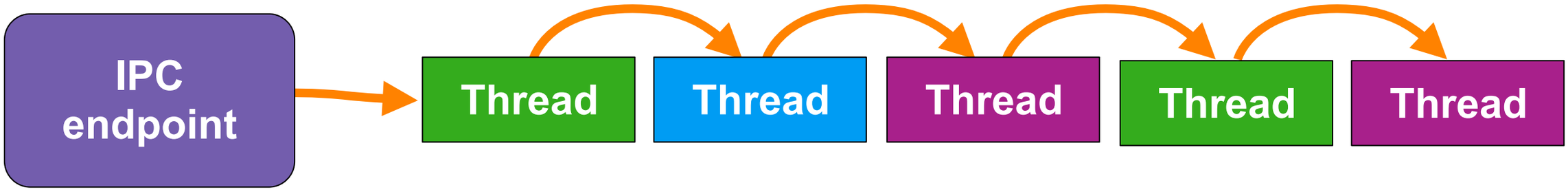
If so, dequeue thread from endpoint and restart it

“Badged” IPC endpoint deletion

NICTA



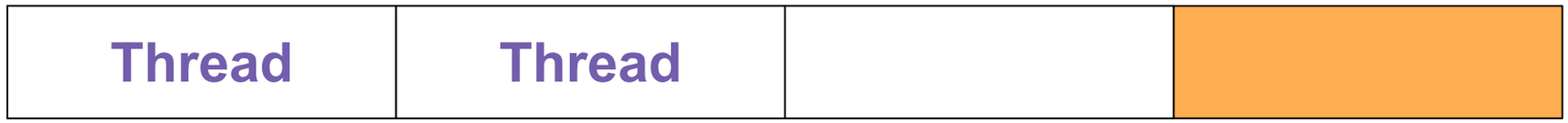
“Badged” IPC endpoint deletion



- Balanced binary tree?
  - Less memory efficient
  - Complex invariants
- Hash table?
  - Variable memory allocation is challenging
  - Still susceptible to pathological worst-case
- Linked-list approach?
  - Incremental modifications to code

Example: object creation

Creating a batch of 2^n objects:



1. Mark free memory region as allocated
2. Divide region into 2^n objects
3. For each object X:
   - Initialise region for X (clear memory)
   - Update bookkeeping data for X

Example: object creation

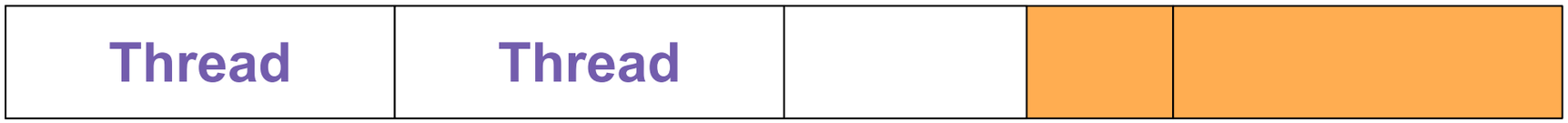
Creating a batch of 2^n objects:

Thread	Thread	Thread	Thread

1. Mark free memory region as allocated
2. Divide region into 2^n objects
3. For each object X:
   - Initialise region for X (clear memory)
   - Update bookkeeping data for X

Any preemption point has complex invariants!

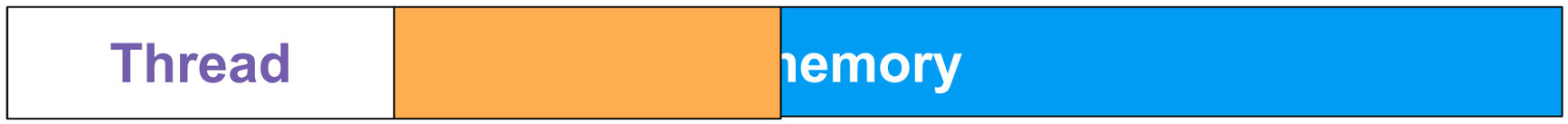
Example: object creation



1. Mark free memory region as allocated
2. Divide region into 2^n objects
3. For each object X:
   - Initialise region for X (clear memory)
   - Update bookkeeping data for X

Any preemption point has complex invariants!

Example: object creation



For each object X:

- Allocate region for X
- Initialise region for X
- Update bookkeeping data for X
- Check for interrupts

Example: object creation

Thread	Thread	Page table

For each object X:

- Allocate region for X
- Initialise region for X
- Update bookkeeping data for X
- Check for interrupts

Broken invariant:

unallocated regions of size 2^n are aligned to 2^n

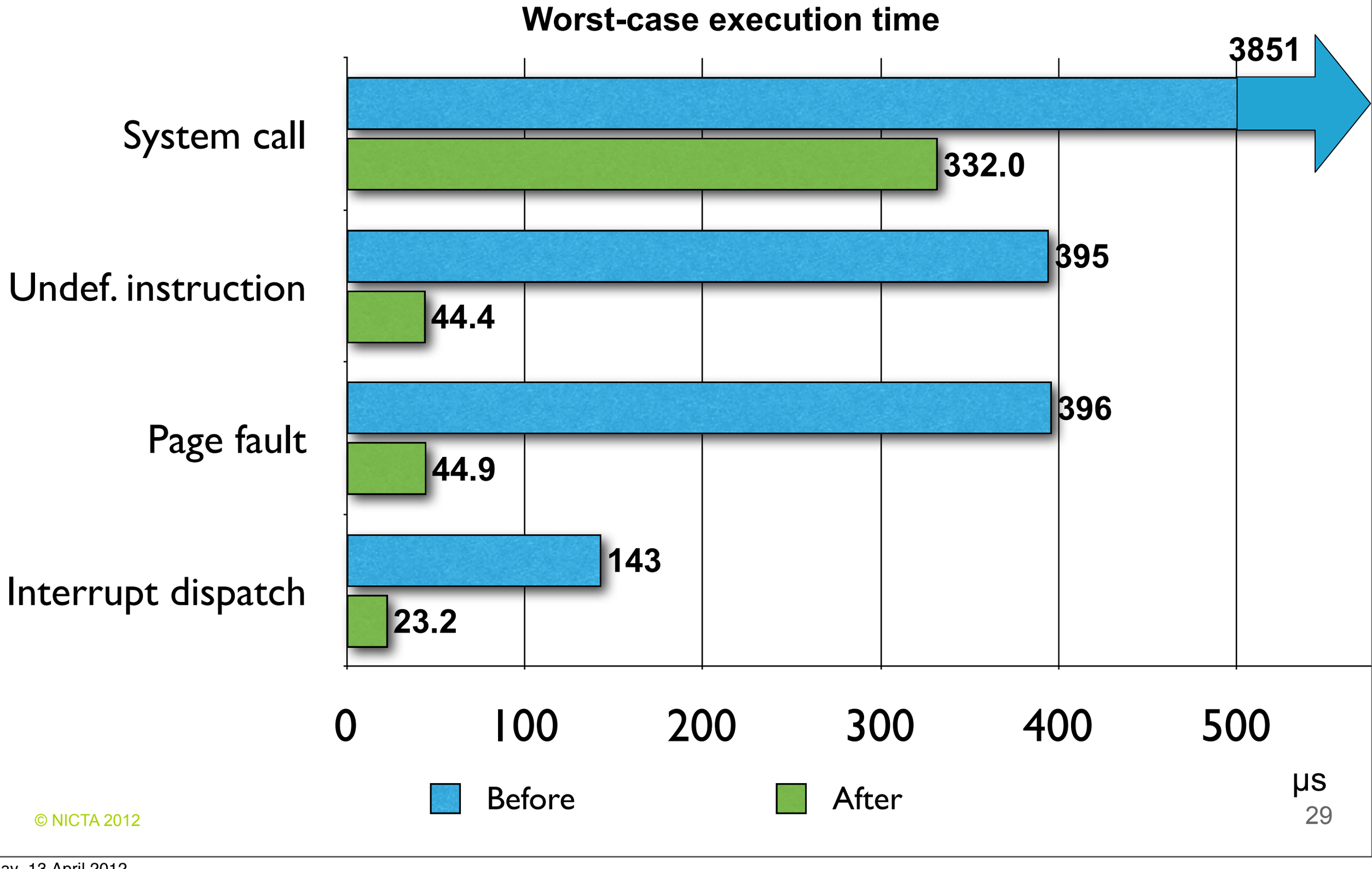
Re-verification: ~ 9 person-months

Common design patterns

“Incremental consistency”

- large composite objects are composed of individual components that can be added or deleted one at a time
- i.e. operations can be decomposed into multiple O(*small*) steps
- simple invariants at intermediate steps

End result...



Lessons learnt

- Don't break invariants
  - unless you need to
- Preemption points are often necessary, but not always sufficient
  - When redoing data structures or algorithms, aim to minimise re-verification overhead
- Design for *incremental consistency*
  - Simplifies the invariants

NICTA

ssrg.nicta.com.au

bernard.blackham@nicta.com.au