From L3 to seL4: What Have We Learnt in 20 Years of L4 Microkernels?

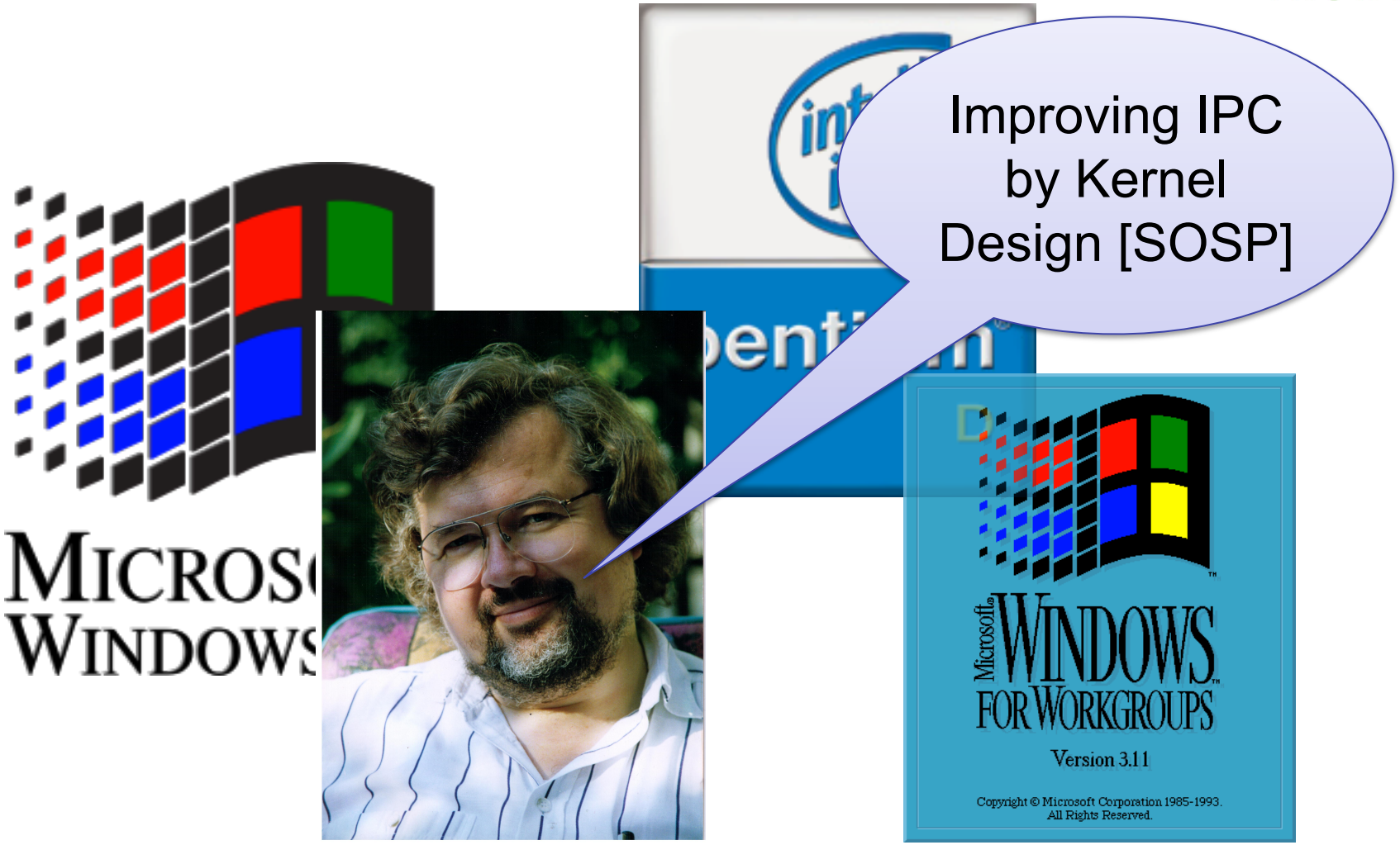
Kevin Elphinstone, Gernot Heiser
NICTA and University of New South Wales

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

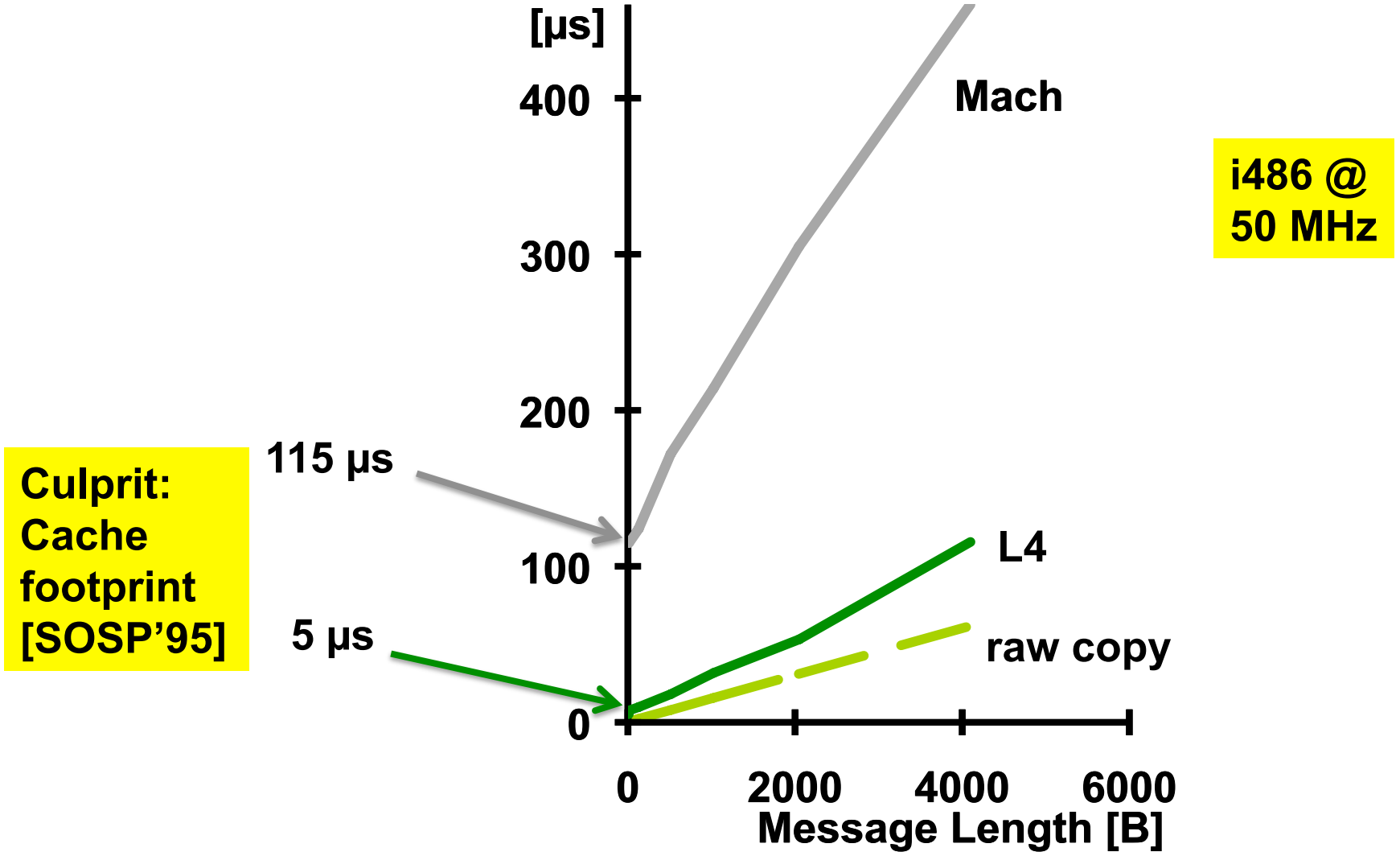
NICTA Funding and Supporting Members and Partners

1993

NICTA


Improving IPC by Kernel Design [SOSP]
Microsoft Windows for Workgroups
Version 3.11
Copyright © Microsoft Corporation 1985-1993.
All Rights Reserved.

1993 IPC Performance



Culprit: Cache footprint [SOSP'95]

i486 @ 50 MHz

IPC Performance over 20 Years

Name	Year	Processor	MHz	Cycles	µs
Original	1993	i486	50	250	5.00
Original	1997	Pentium	160	121	0.75
L4/MIPS	1997	R4700	100	86	0.86
L4/Alpha	1997	21064	433	45	0.10
Hazelnut	2002	Pentium 4	1,400	2,000	1.38
Pistachio	2005	Itanium	1,500	36	0.02
OKL4	2007	XScale 255	400	151	0.64
NOVA	2010	i7 Bloomfield (32-bit)	2,660	288	0.11
seL4	2013	i7 Haswell (32-bit)	3,400	301	0.09
seL4	2013	ARM11	532	188	0.35
seL4	2013	Cortex A9	1,000	316	0.32

©2013 Gernot Heiser, NICTA

Core Microkernel Principle: Minimality

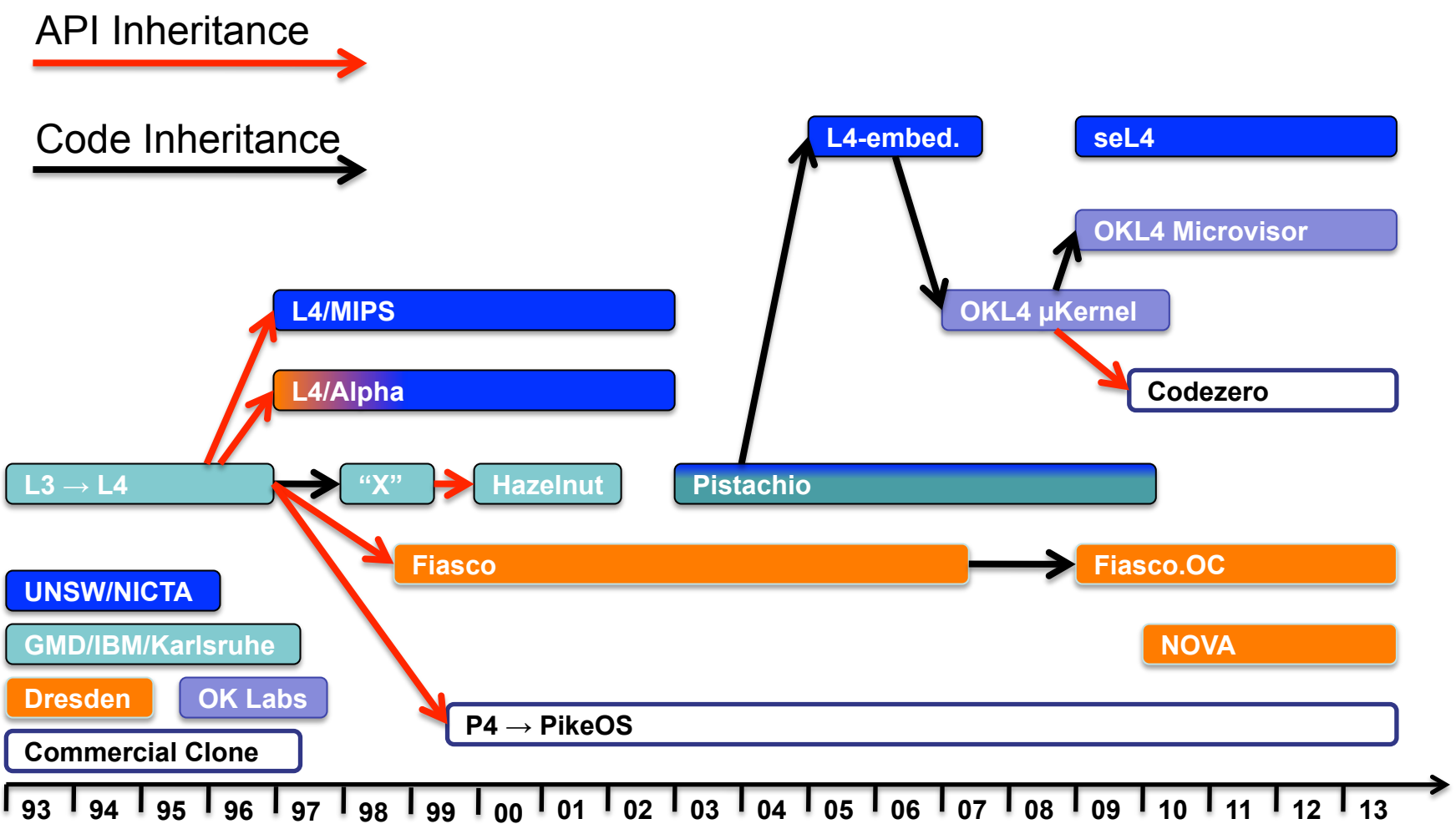
A concept is tolerated inside the microkernel only if moving it outside the kernel, i.e. permitting competing implementations, would prevent the implementation of the system's required functionality. [SOSP'95]

Minimality: Source Code Size

Name	Architecture	C/C++	asm	total kSLOC
Original	i486	0	6.4	6.4
L4/Alpha	Alpha	0	14.2	14.2
L4/MIPS	MIPS64	6.0	4.5	10.5
Hazelnut	x86	10.0	0.8	10.8
Pistachio	x86	22.4	1.4	23.0
L4-embedded	ARMv5	7.6	1.4	9.0
OKL4 3.0	ARMv6	15.0	0.0	15.0
Fiasco.OC	x86	36.2	1.1	37.6
seL4	ARMv6	9.7	0.5	10.2

©2013 Gernot Heiser, NICTA

L4 Family Tree



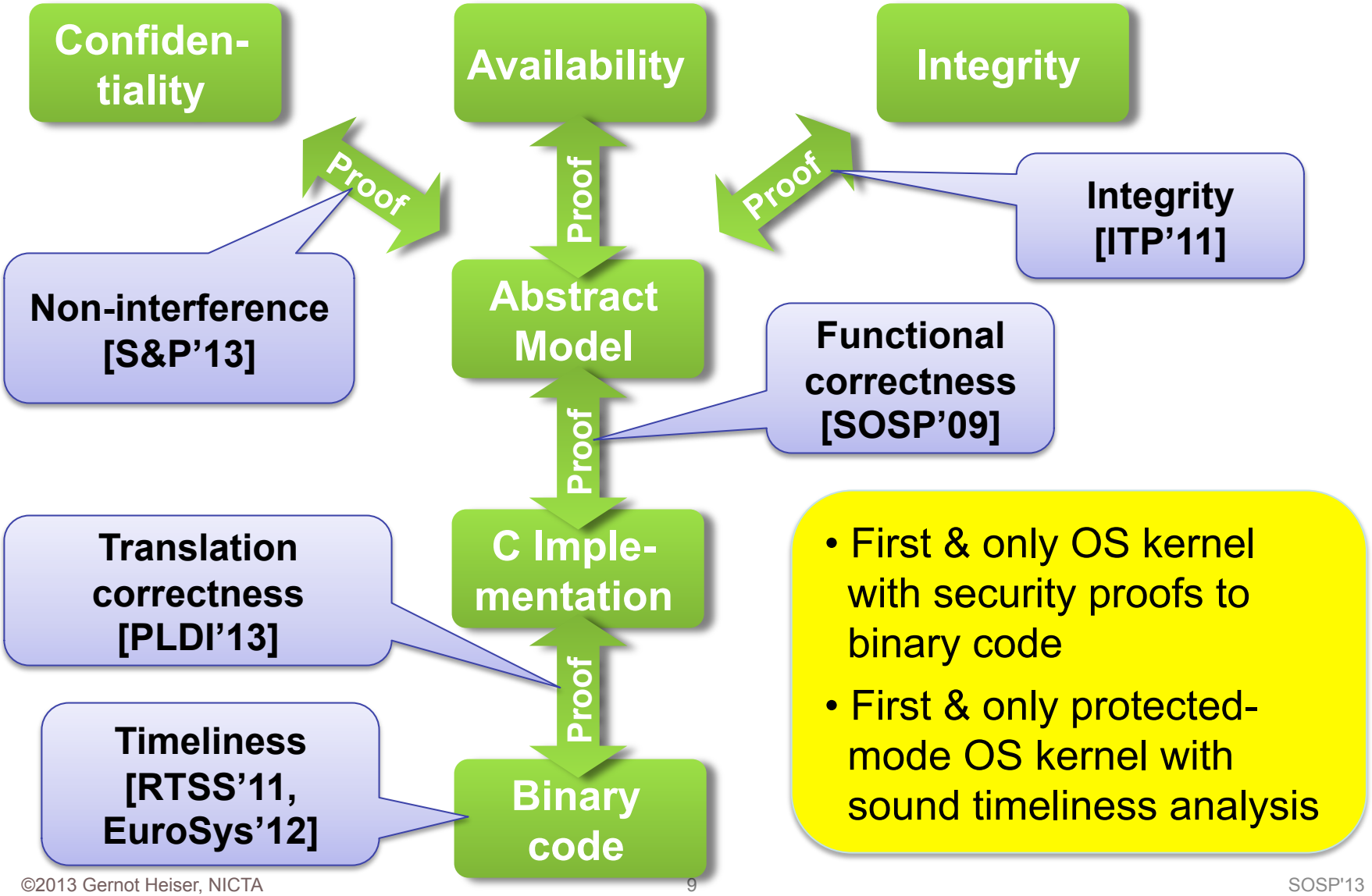
L4 Deployments – in the Billions

NICTA

SiMKo 3 “Merkelphone”

seL4: Unprecedented Dependability

NICTA



- First & only OS kernel with security proofs to binary code
- First & only protected-mode OS kernel with sound timeliness analysis

L4 Design and Implementation

Implement. Tricks [SOSP'93]

- Process kernel
- Virtual TCB array
- Lazy scheduling
- Direct process switch
- Non-preemptible
- Non-portable
- Non-standard calling convention
- Assembler

Design Decisions [SOSP'95]

- Synchronous IPC
- Rich message structure, arbitrary out-of-line messages
- Zero-copy register messages
- User-mode page-fault handlers
- Threads as IPC destinations
- IPC timeouts
- Hierarchical IPC control
- User-mode device drivers
- Process hierarchy
- Recursive address-space construction

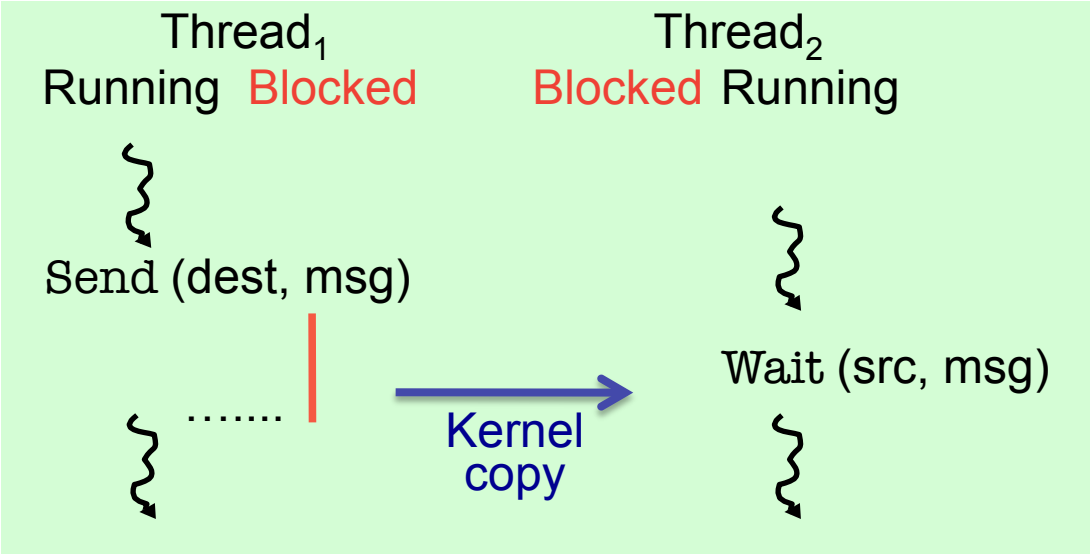
Objective: Minimise cache footprint and TLB misses

DESIGN

L4 Synchronous IPC

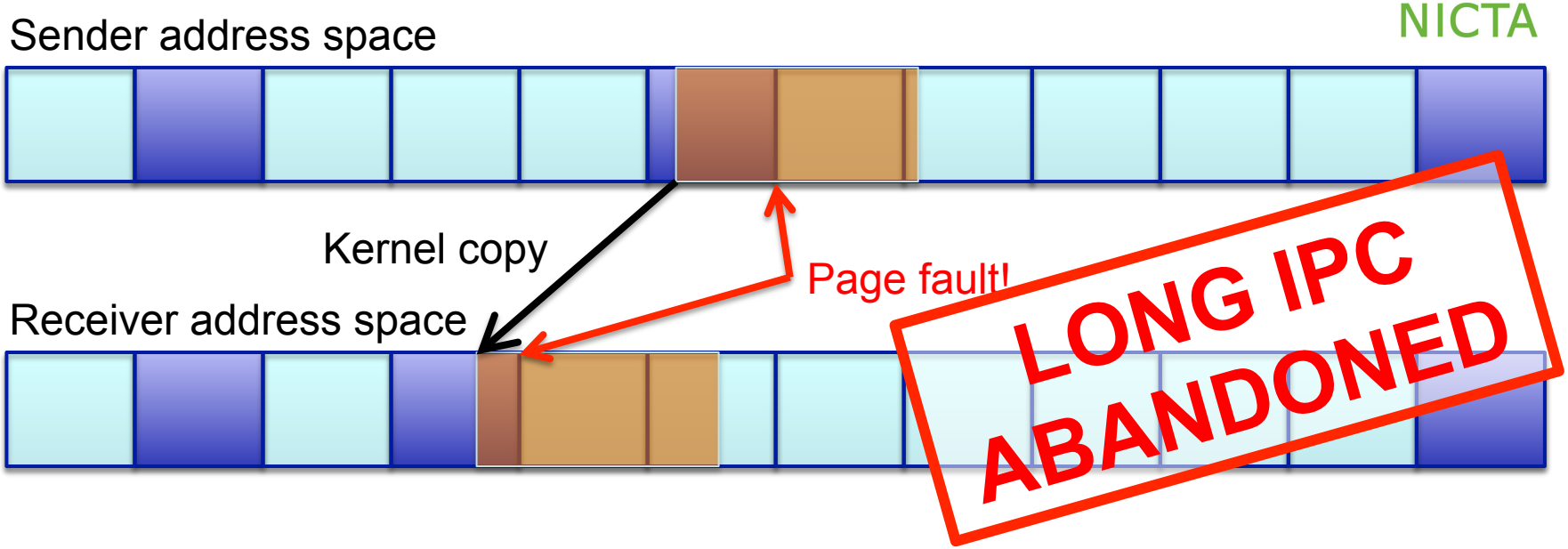
NICTA

Rendezvous model



Kernel executes in sender's context
- copies memory data directly to receiver (single-copy)
- leaves message registers unchanged during context switch (zero copy)

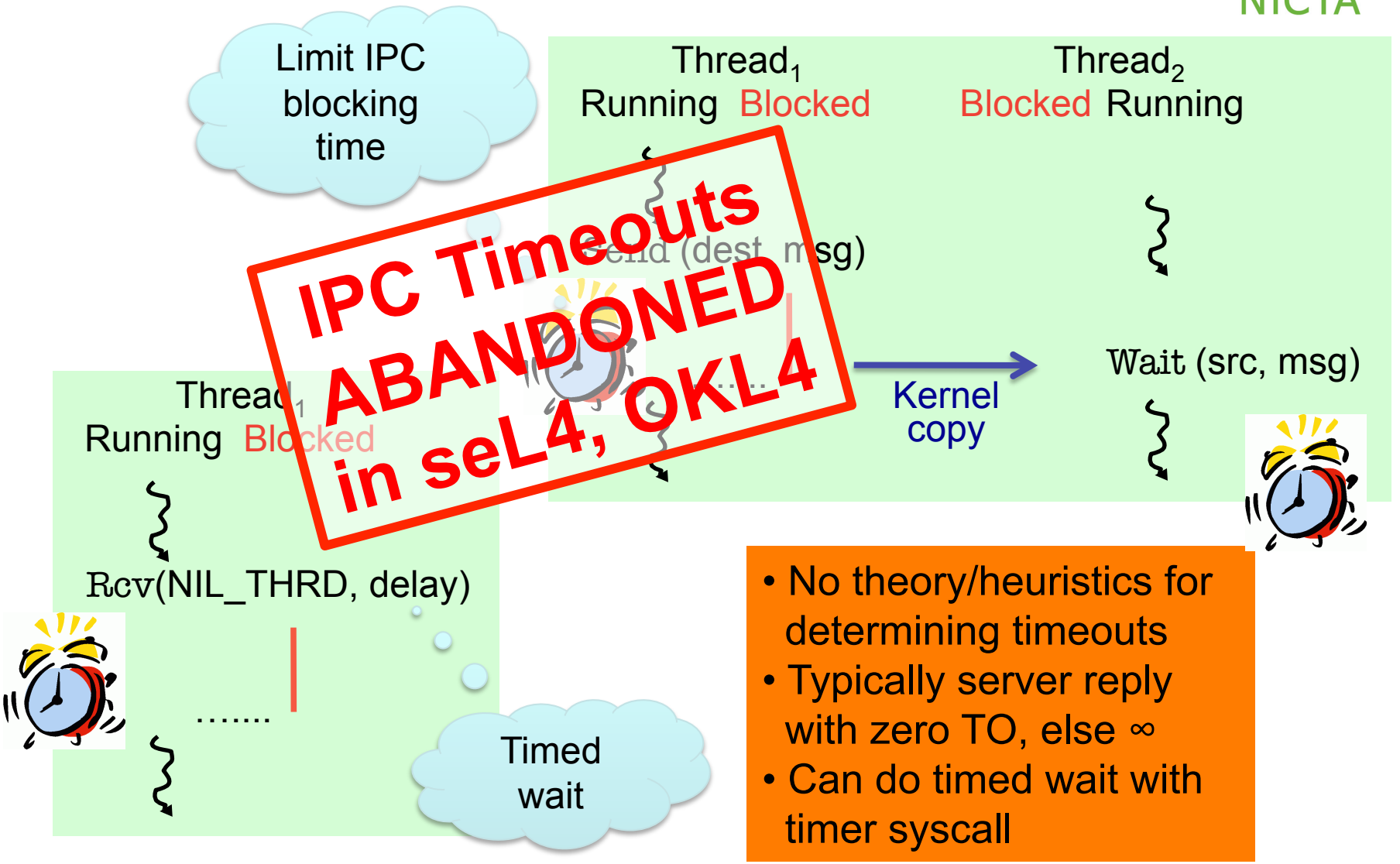
“Long” IPC



- IPC page faults are nested exceptions ⇒ In-kernel concurrency
  - L4 executes with interrupts disabled for performance, no concurrency
- Must invoke untrusted usermode page-fault handlers
  - potential for DOSing other thread
- Timeouts to avoid DOS attacks
  - complexity

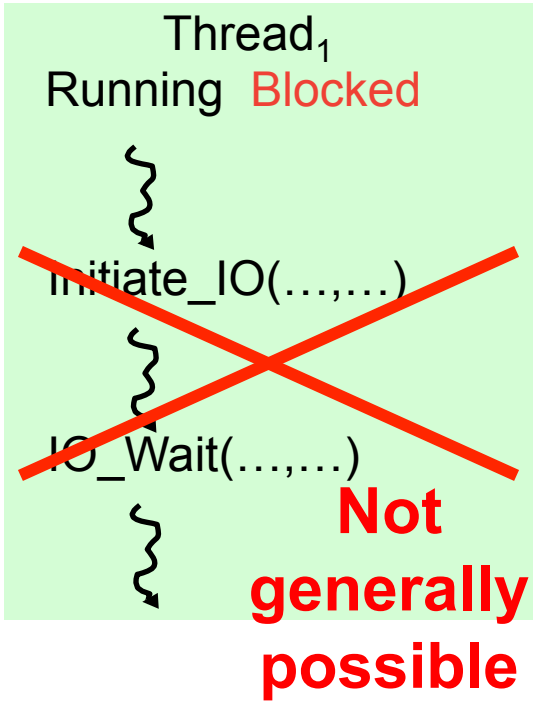
Why have long IPC?
- POSIX-style APIs
  write (fd, buf, nbytes)
- Usually prefer shared buffers

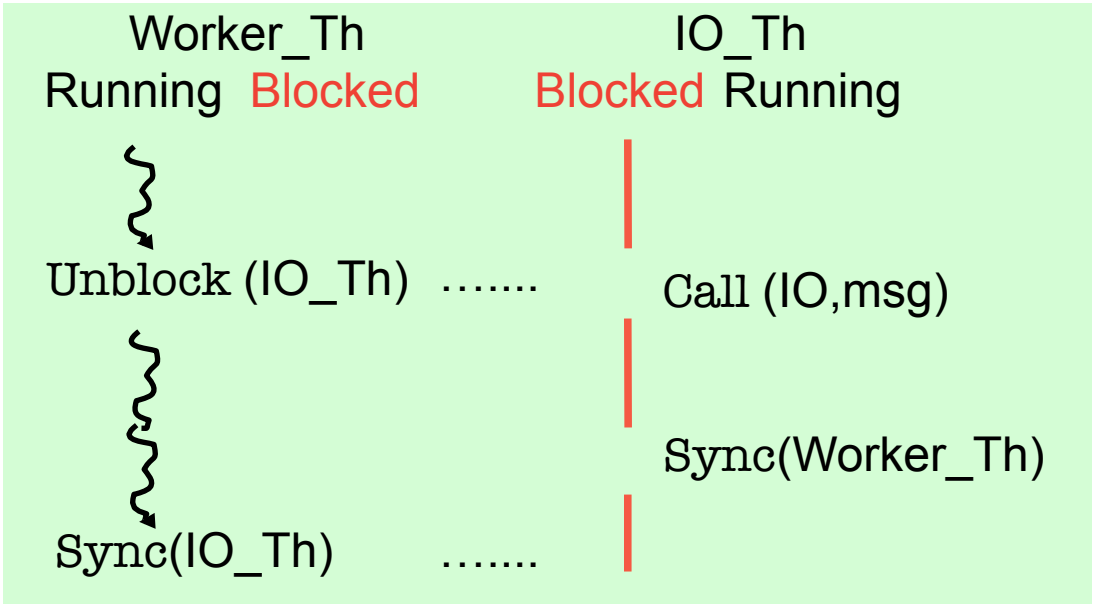
Timeouts



- No theory/heuristics for determining timeouts
- Typically server reply with zero TO, else ∞
- Can do timed wait with timer syscall

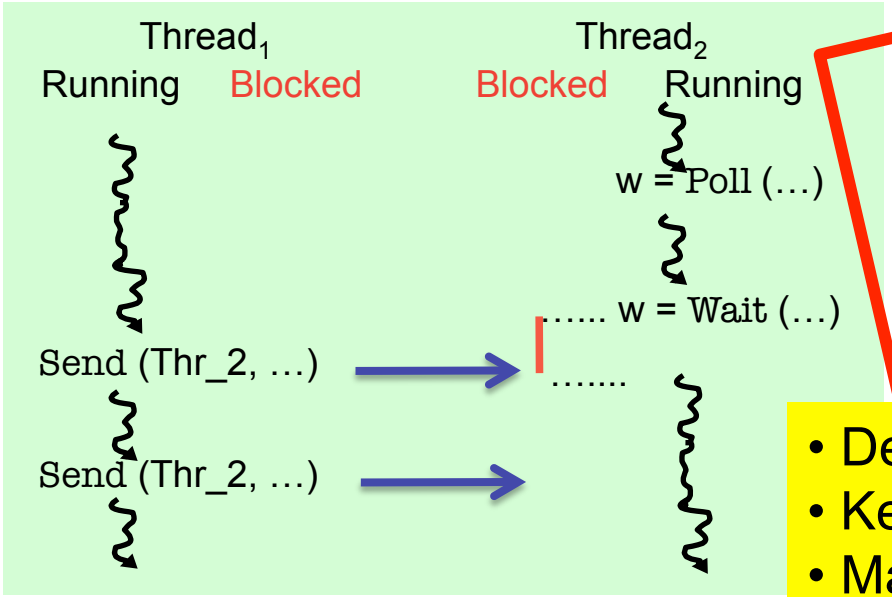
Synchronous IPC Issues





- Sync IPC forces multi-threaded code!
- Also poor choice for multi-core

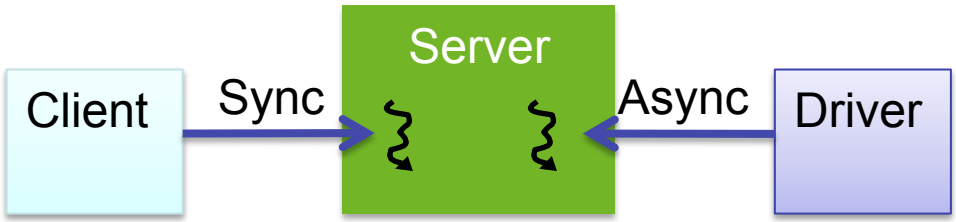
Asynchronous Notifications



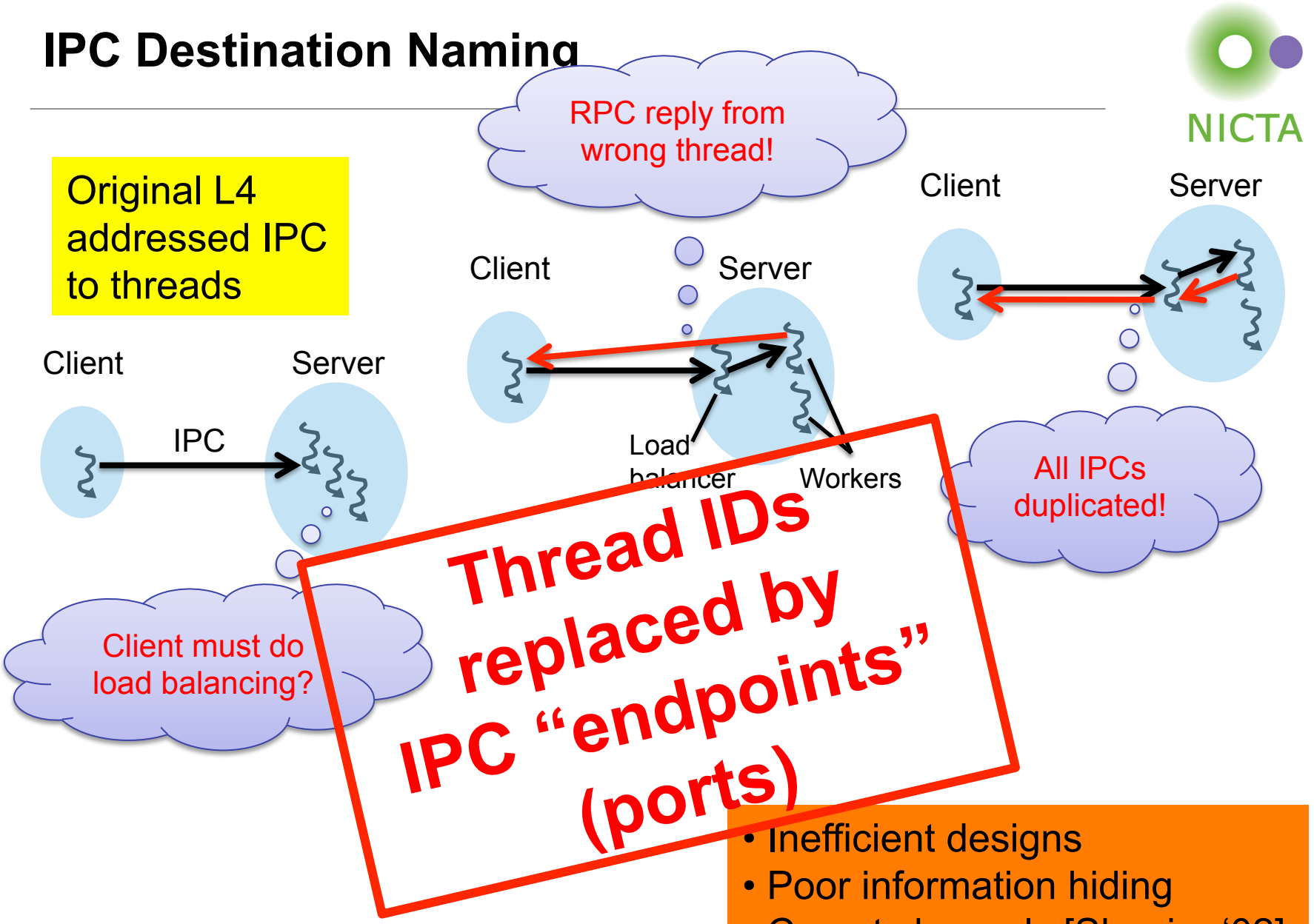
Sync IPC complemented with async

- Delivers few bits (destructively)
- Kernel only buffers single word
- Maps well to interrupts, exceptions

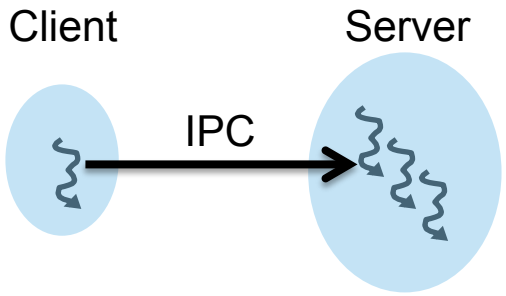
- Thread can wait for synchronous and asynchronous messages concurrently

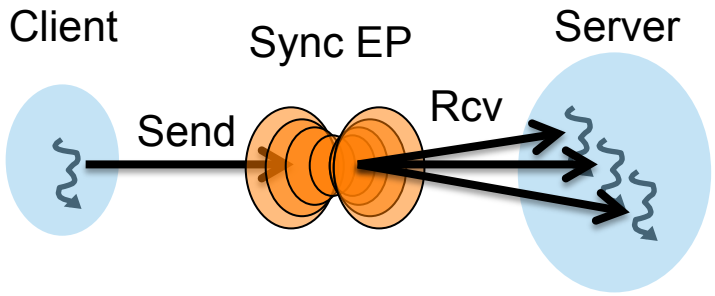


IPC Destination Naming

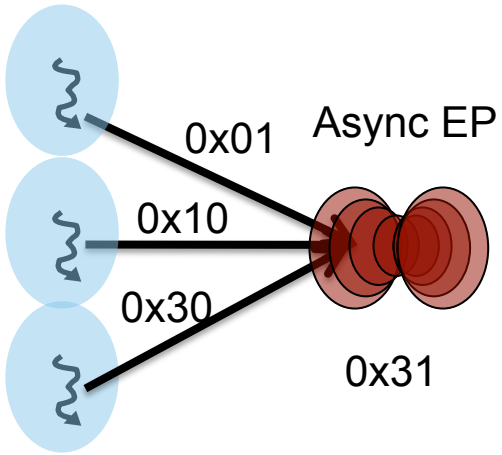


Endpoints





- Sync EP queues senders/receivers
- Does not buffer messages

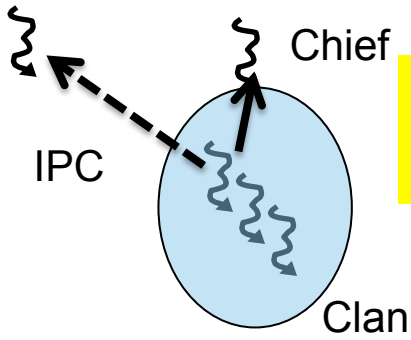


- Async EP accumulates bits

Other Design Issues

NICTA

IPC Control: "Clans & Chiefs"

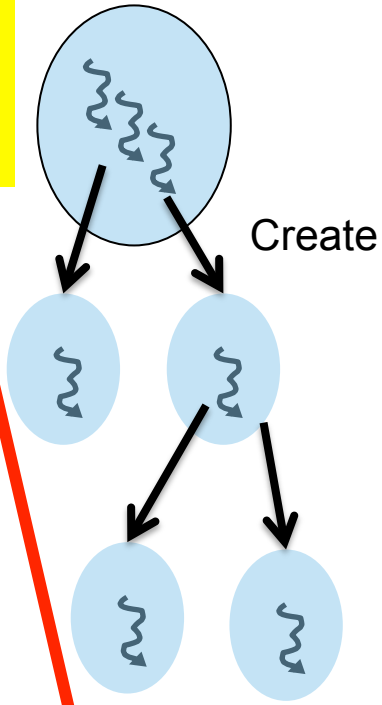


IPC outside clan re-directs to chief

Inflexible, clumsy, inefficient hierarchies!

Process Hierarchy

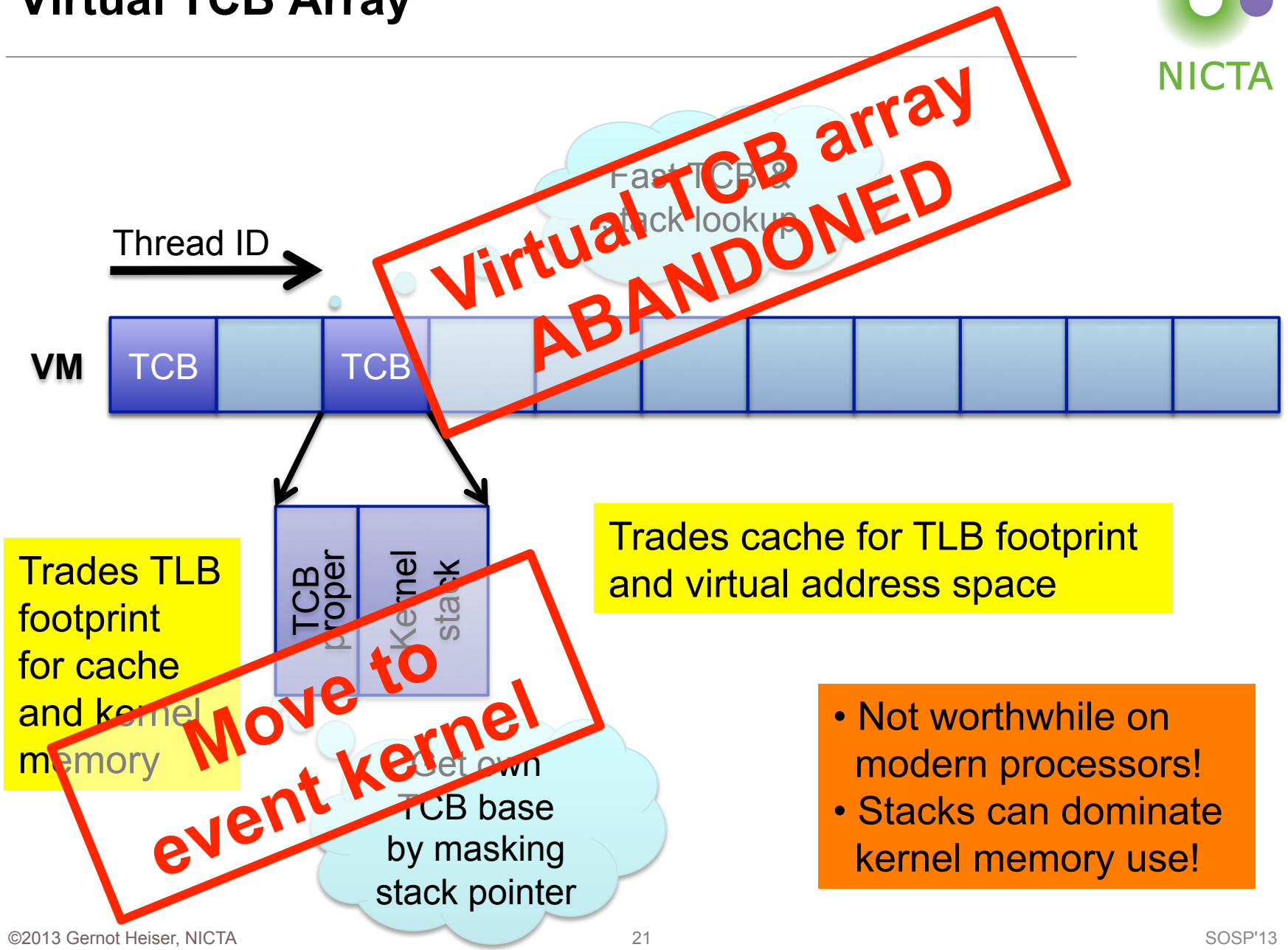
Hierarchical resource management



Hierarchies replaced by delegatable cap-based access control

IMPLEMENTATION

Virtual TCB Array



“Lazy” Scheduling

- In IPC-based systems, threads block and unblock frequenty
- Many ready queue manipulations

Idea: leave blocked threads in ready queue, scheduler cleans up

Lazy scheduling REPLACED

```
thread_t schedule() {
    foreach (prio in priorities) {
        foreach (thread in runQueue[prio]) {
            if (isRunnable(thread))
                return thread;
            else
                schedDequeue(thread);
            }
        }
    return idleThread;
}
```

Scheduler execution time is unbounded!

“Benno scheduling”:

- All threads on ready queue are runnable
- All runnable threads in ready queue *except the running one*

L4 Design and Implementation

Implement. Tricks [SOSP'93]

- ~~Process kernel~~
- ~~Virtual TCB array~~
- ~~Lazy scheduling~~
- Direct process switch
- Non-preemptible
- ~~Non-portable~~
- ~~Non-standard calling convention~~
- ~~Assembler~~

Design Decisions [SOSP'95]

- Synchronous IPC
- Rich message structure, ~~arbitrary out-of-line~~ messages
- Zero-copy register messages
- User-mode page-fault handlers
- ~~Threads as IPC~~ destinations
- ~~IPC timeouts~~
- ~~Hierarchical IPC~~ control
- User-mode device drivers
- ~~Process hierarchy~~
- Recursive address-space construction

What are the Principles?

- Minimality is excellent driver of design decisions
  - L4 kernels have become simpler over time
  - Policy-mechanism separation (user-mode page-fault handlers)
  - Device drivers really belong to user level
  - Minimality is key enabler for formal verification!

- IPC speed still matters
  - But not everywhere, premature optimisation is wastive
  - Compilers have got so much better
  - Verification does not compromise performance
  - Verification invariants can help improve speed! [Shi, OOPSLA'13]

- Also found that capabilities are the way to go
  - Shapiro (EROS) was right

- However, a clean abstraction of time still elusive

Conclusions

- Details changed, but principles remained
- Microkernels rock! (If done right!)

Thank you!

We're hiring:
- **Chair in Software Systems**
- **Postdocs / junior faculty**