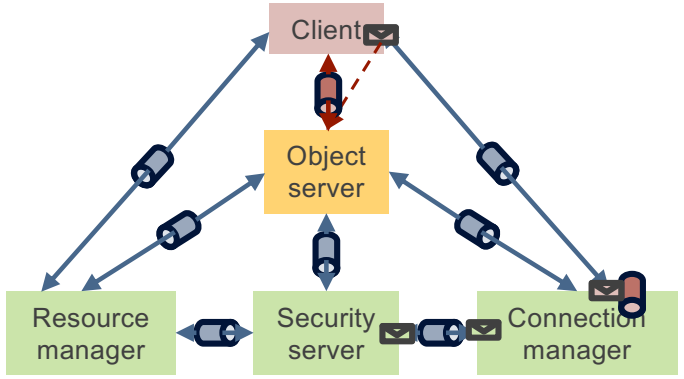
School of Computer Science & Engineering

COMP9242 Advanced Operating Systems

2025 T3 Week 10 Part 2

seL4 Research at TS@UNSW

@GernotHeiser



Copyright Notice

Today's Lecture

- More on LionsOS performance
- Secure General-purpose OS
- Pancake: System language for verified systems
- Time Protection: Prevention of μarch timing channels

More ob LionsOS

Fast – secure – adaptable!

Microkernel Overheads

Chen et al, OSDI'24

High syscall rate = 61k/s

seL4 round-trip address-space switch = 1k cy

Conservative IMHO

Assume average 2 R-T AS switches / syscall:

Switch O/H = 2 × 61k/k × 1kcy = 122M cy/s

Assume 3GHz clock:

O/H = 122M cy/s / 3Gcy/s = 122/3k = 4%

Assume 4-core CPU:

O/H = 4%/4 = 1% of CPU!

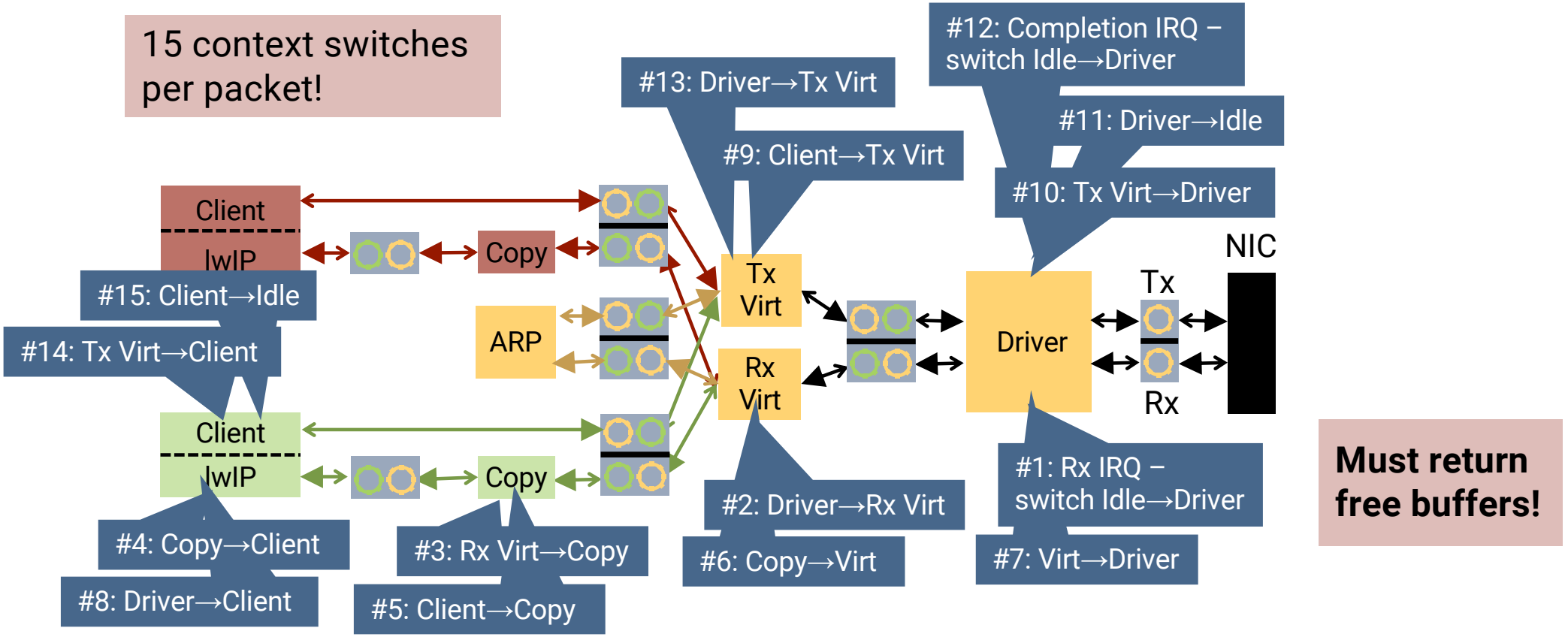
Assume Linux max CPU load = 25%

relative O/H = 4 × 1% = 4%

Why would anyone care?

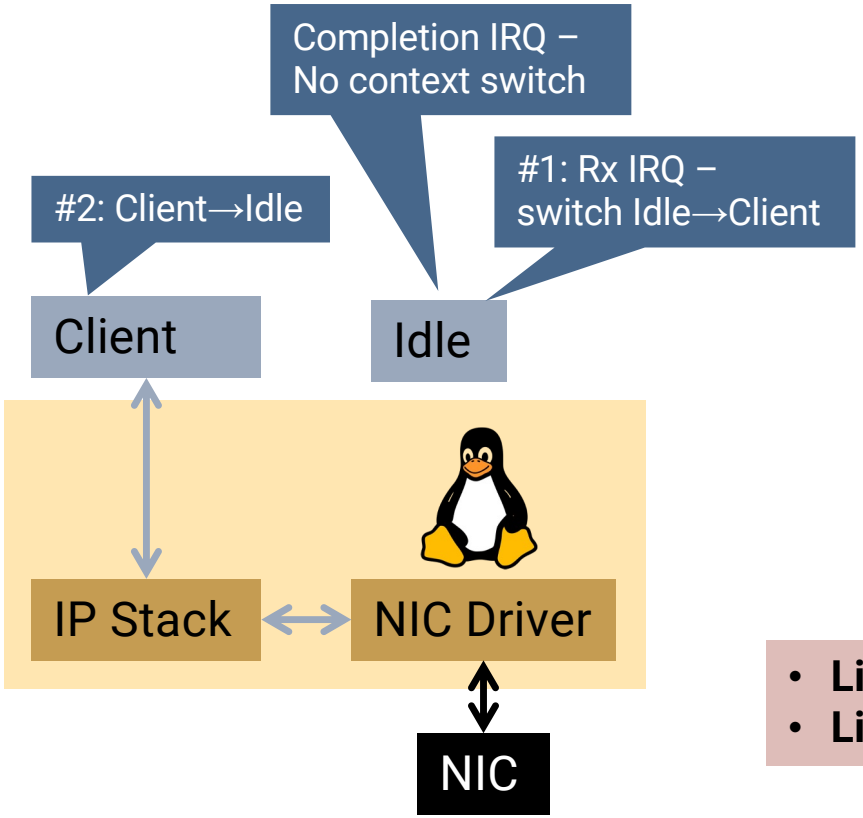
Packet Round-Trip Context Switches

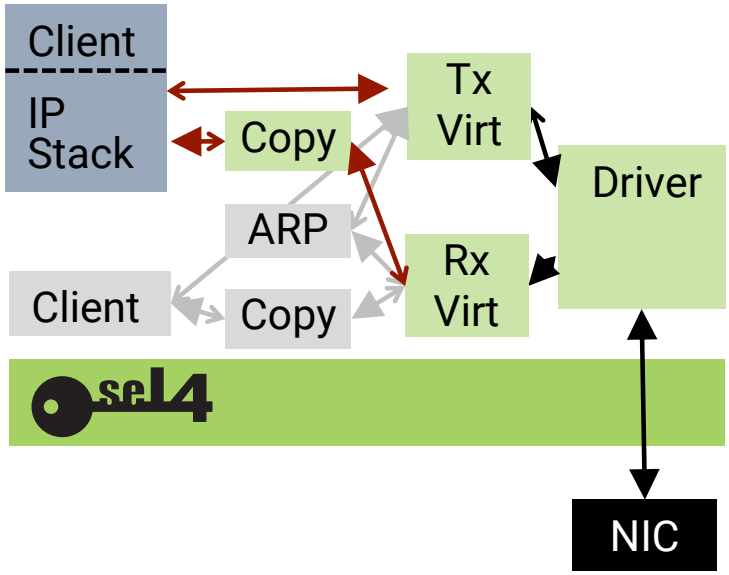
15 context switches per packet!



Must return free buffers!

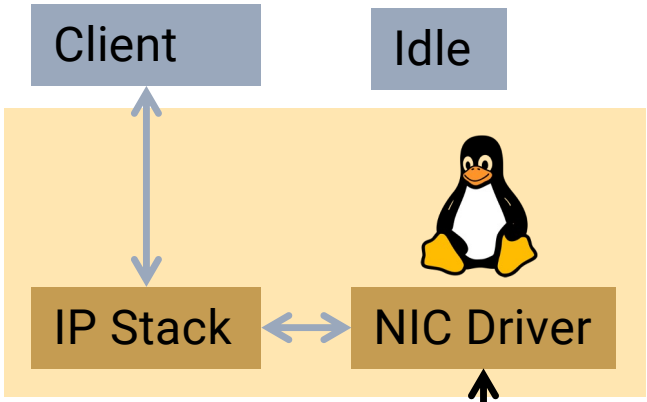
Comparing to Linux

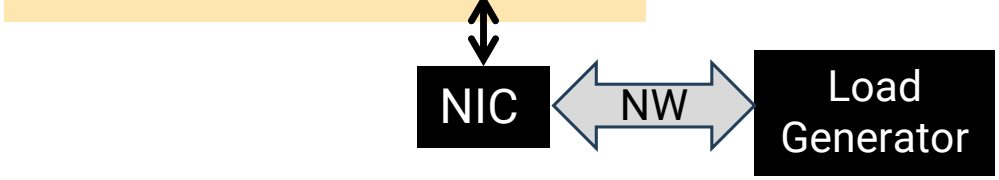


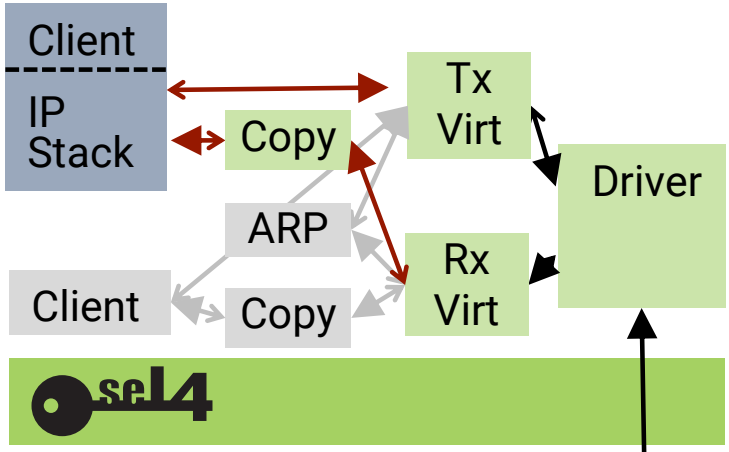


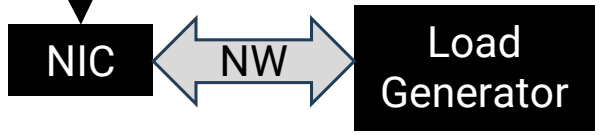
- **LionsOS:** 30 mode switches, 15 context switches
- **Linux:** 6 mode switches, 2 context switches

Comparing Performance: Setup









- External load generator
- Measures throughput, latency
- Client echoes packets

What Do We Expect?

Ethernet packet size = 1.5kB

Assume Linux mode switch = half context switch

LionsOS O/H = 12/pk × 0.5k cy = 6k cy/pkt

Max packet rate for 1Gb/s NIC:

rate = 1Gb/s / 1.5kB = 1Gb/s / 12kb = 833k/s

Worst-case O/H for 1Gb/s NIC:

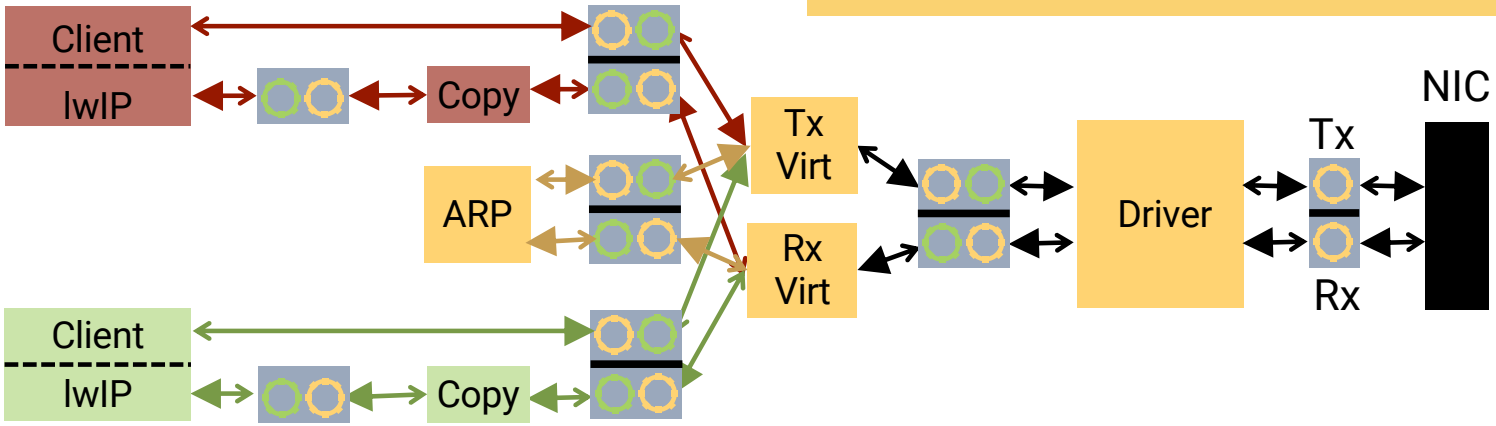
O/H = 6k cy/pkt * 833k pkt/s = 0.5G cy/s

Assume 3GHz clock:

rel O/H = 0.5G cy/s / 3G cy/s = 17% of core

However, There's Batching

- Each component will process everything in its queue before signalling another component
- No component will ever busy-poll!



- Dramatically reduces context switches under load!
- Measure 5–10 pkt/IRQ!

What Do We Expect?

Ethernet packet size = 1.5kB

Assume Linux mode switch = half context switch

LionsOS O/H = 12/pk × 0.5k cy = 6k cy/pkt

Max packet rate for 1Gb/s NIC:

rate = 1Gb/s / 1.5kB = 1Gb/s / 12kb = 833k/s

Worst-case O/H for 1Gb/s NIC:

O/H = 6k cy/pkt * 833k pkt/s = 0.5

Assume 3GHz clock:

rel O/H = 0.5G cy/s / 3G cy/s = 17% of core

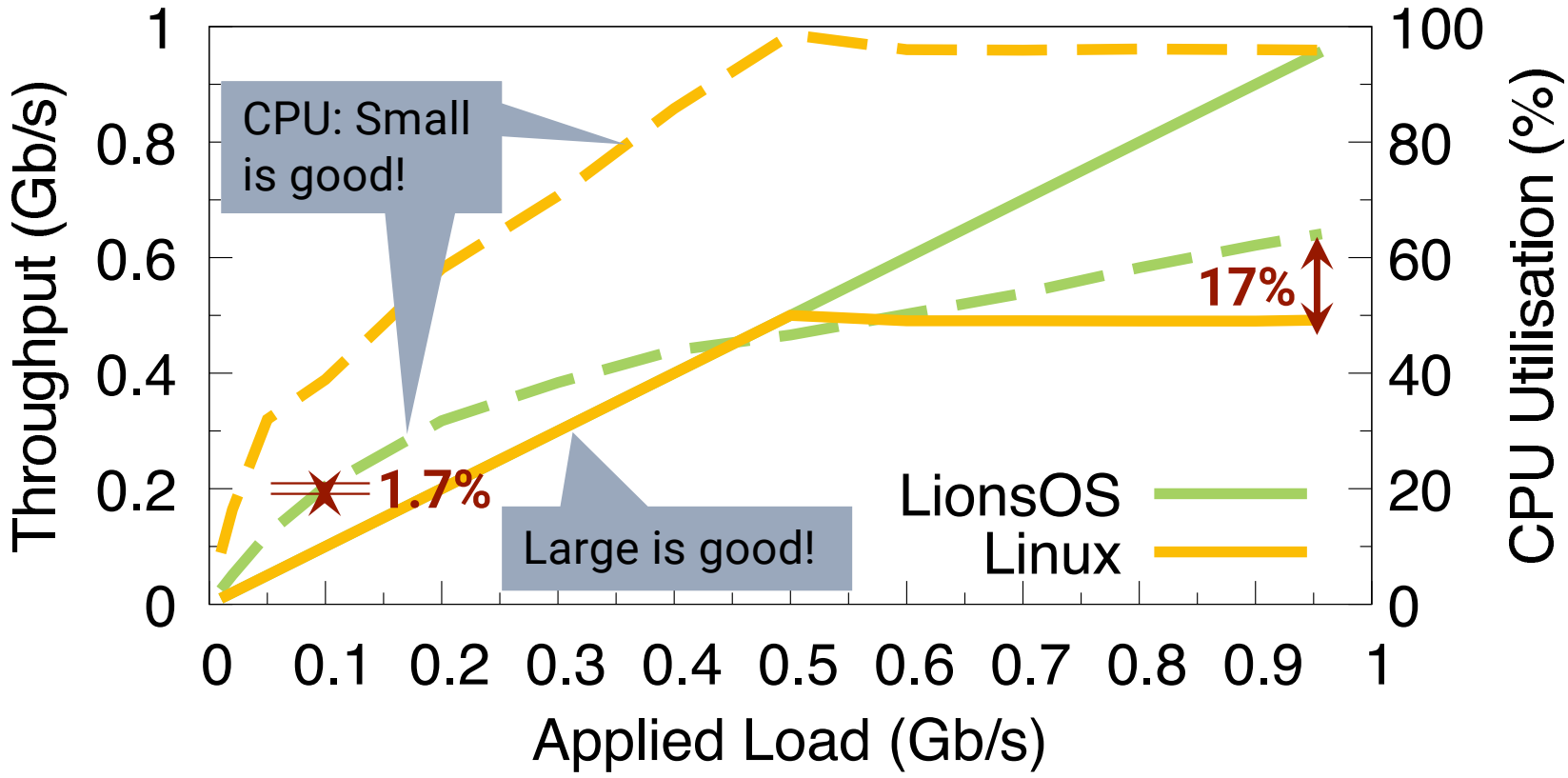
At 100Mb/s, packet spacing = 1/(83k/s) = 12µs

rel O/H = 17%/10 = 1.7% of core

Highly pessimistic due to natural batching!

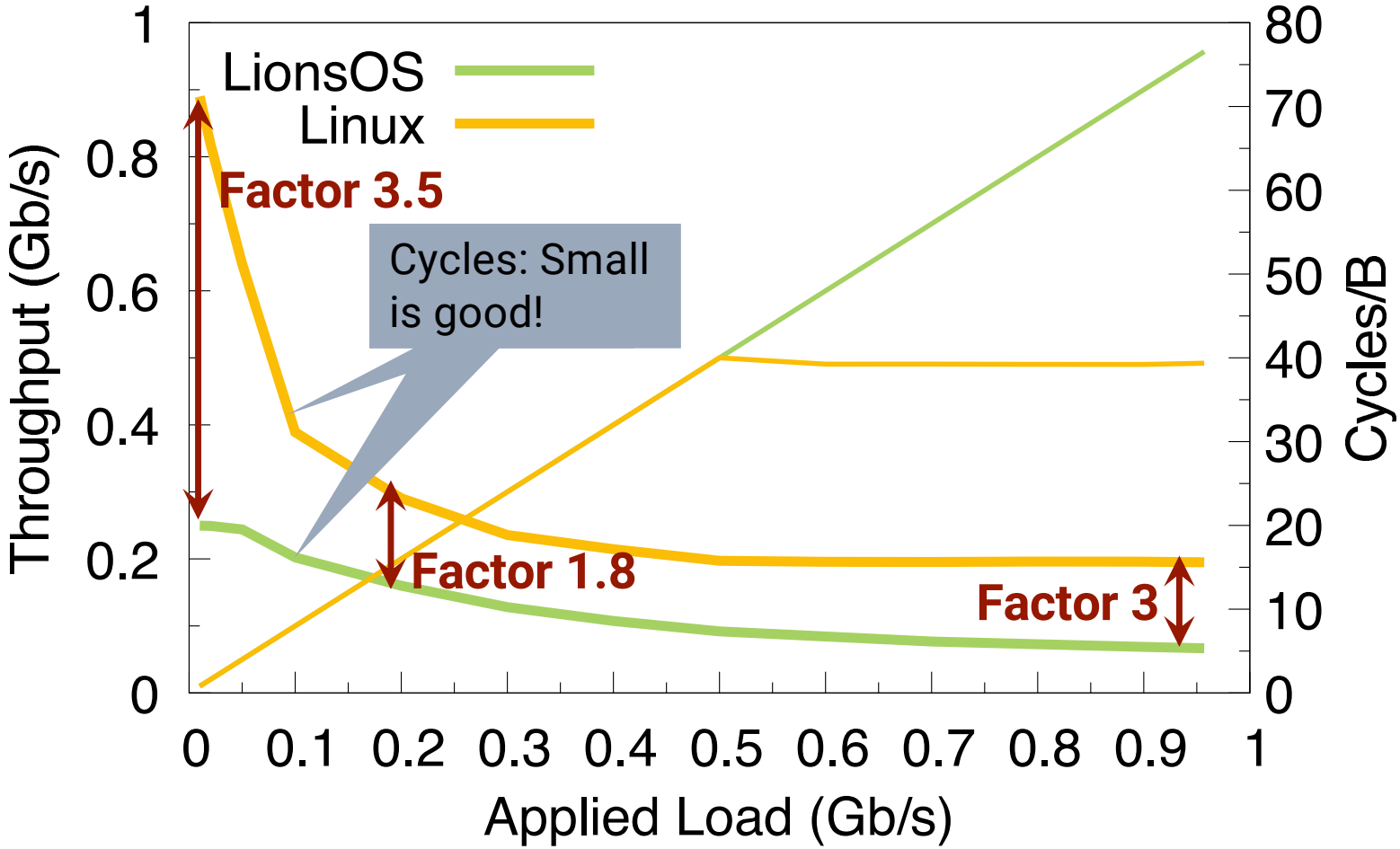
Avoid batching by spacing packets!

Performance: i.MX8M, 1Gb/s Eth, UDP

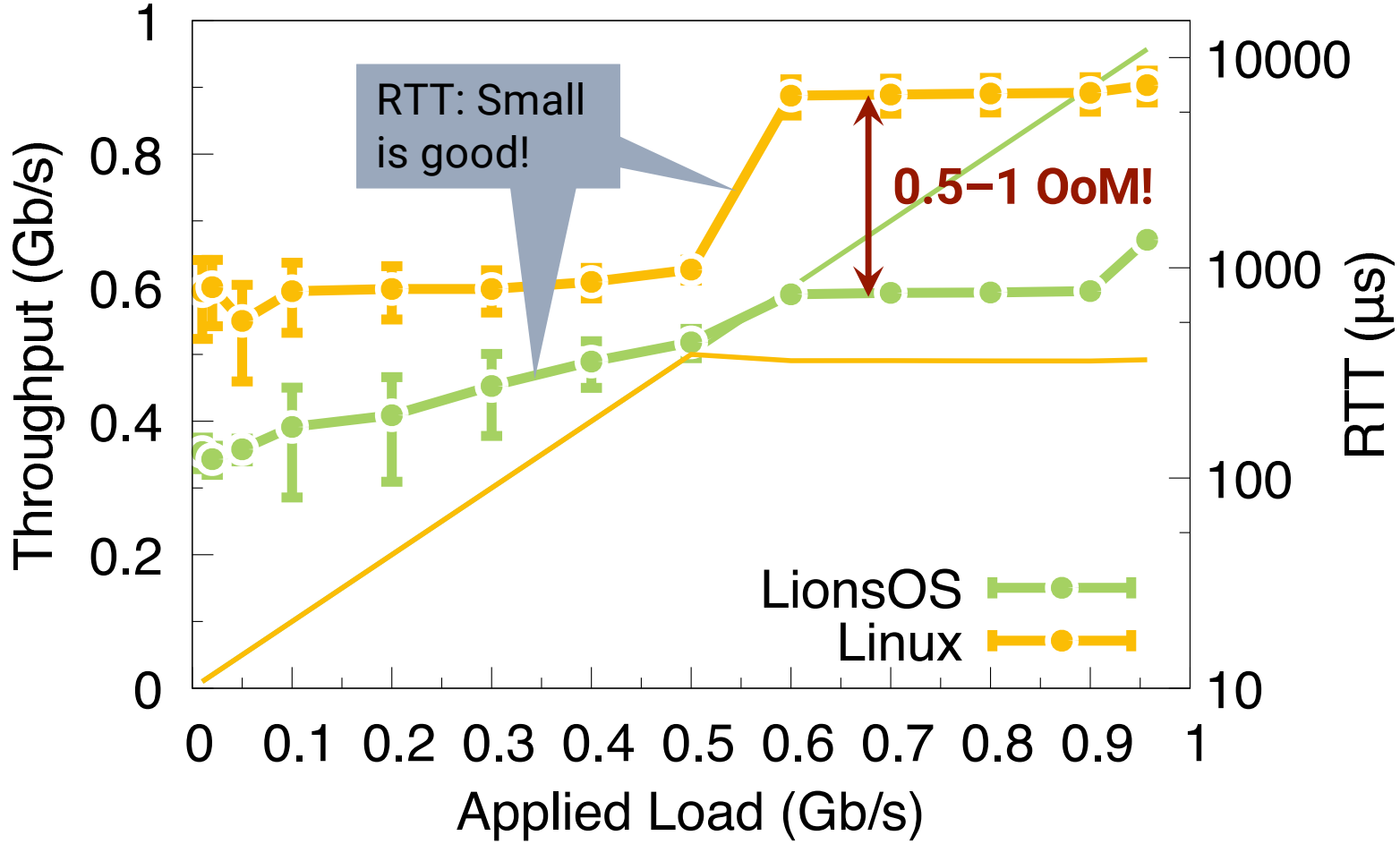


Single-core configuration

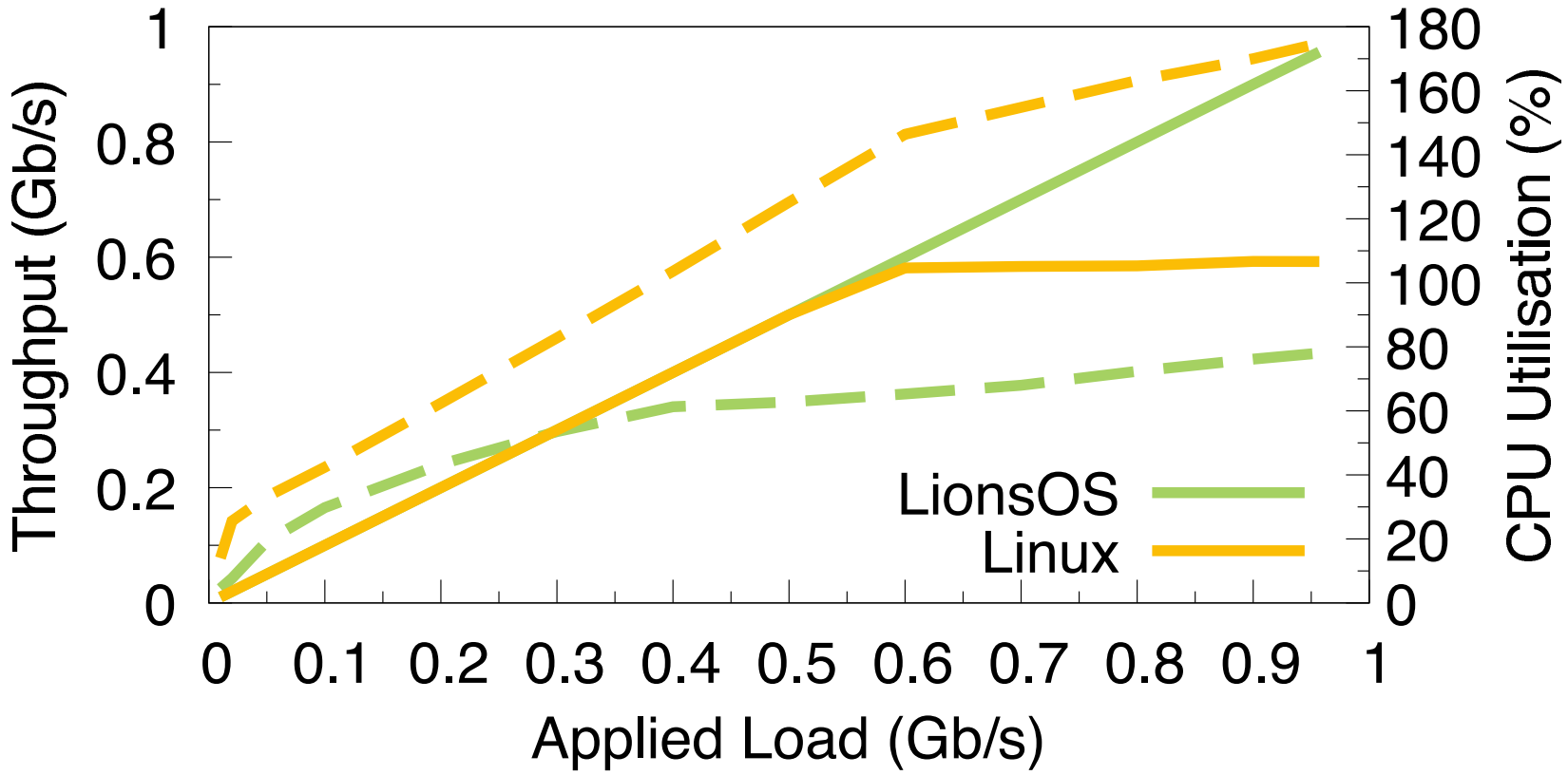
Performance: Processing Cost per Byte



Performance: Round-Trip Times

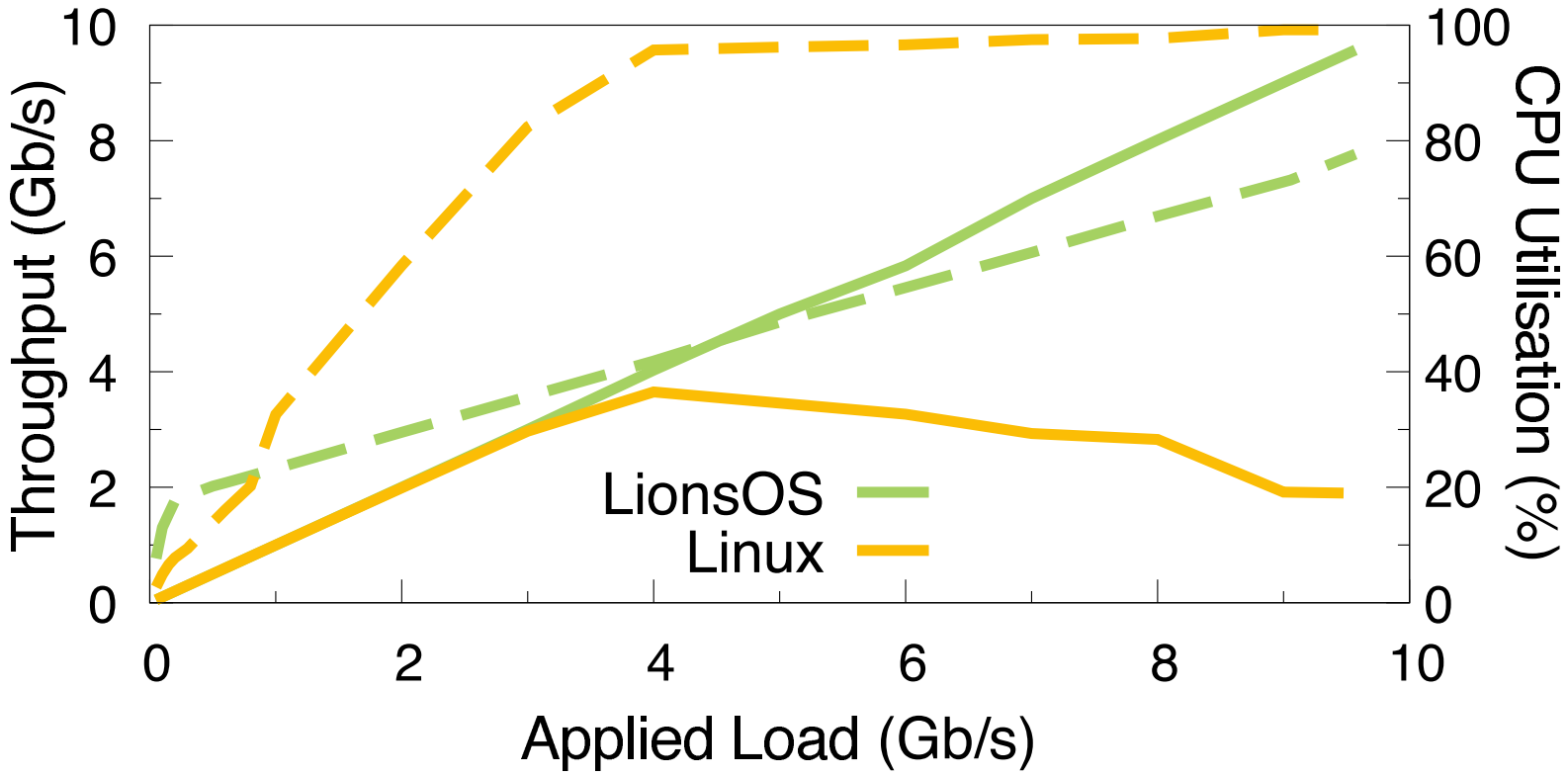


Performance: i.MX8M, 1Gb/s Eth, UDP



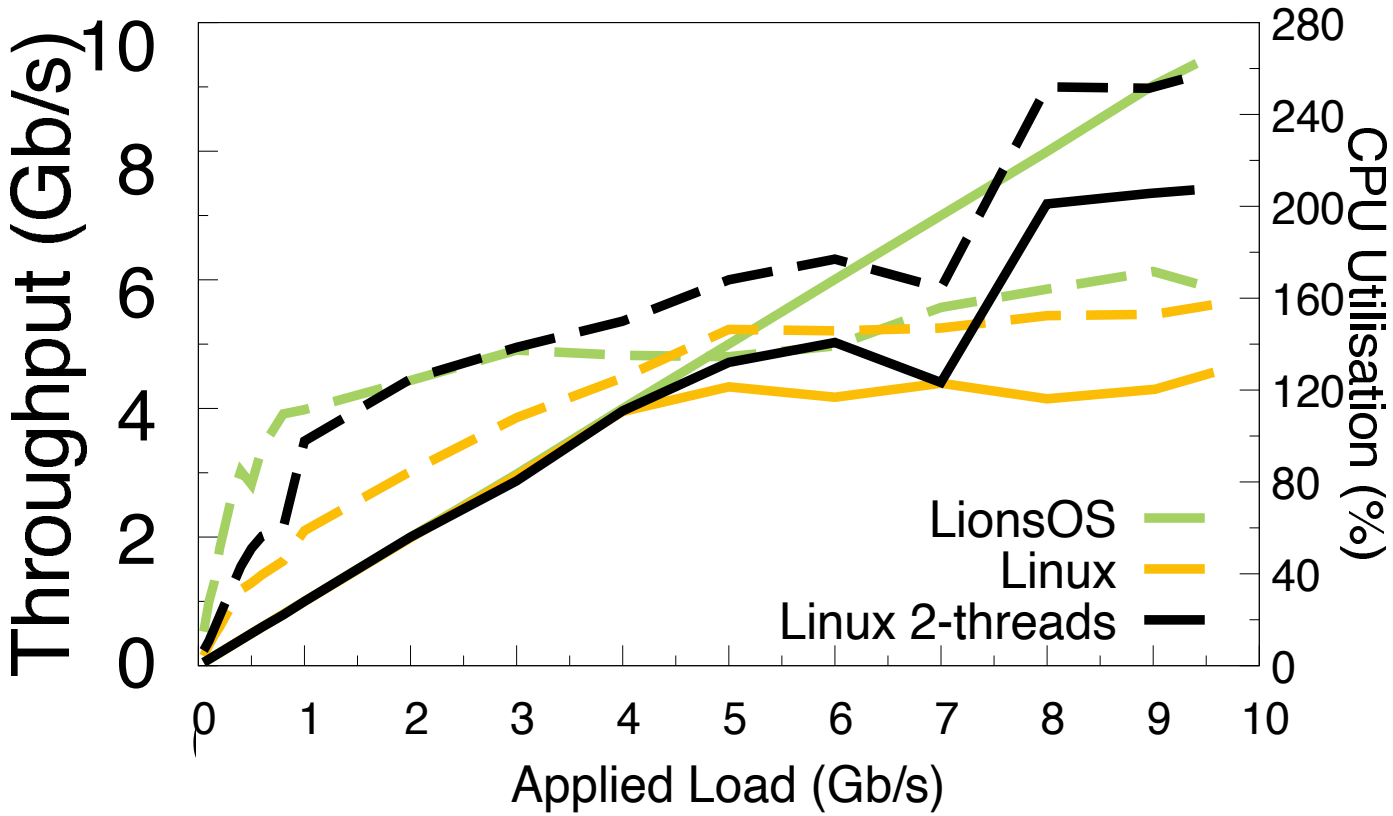
Multicore configuration

Performance: x86, 10Gb/s Eth, UDP



Single-core configuration

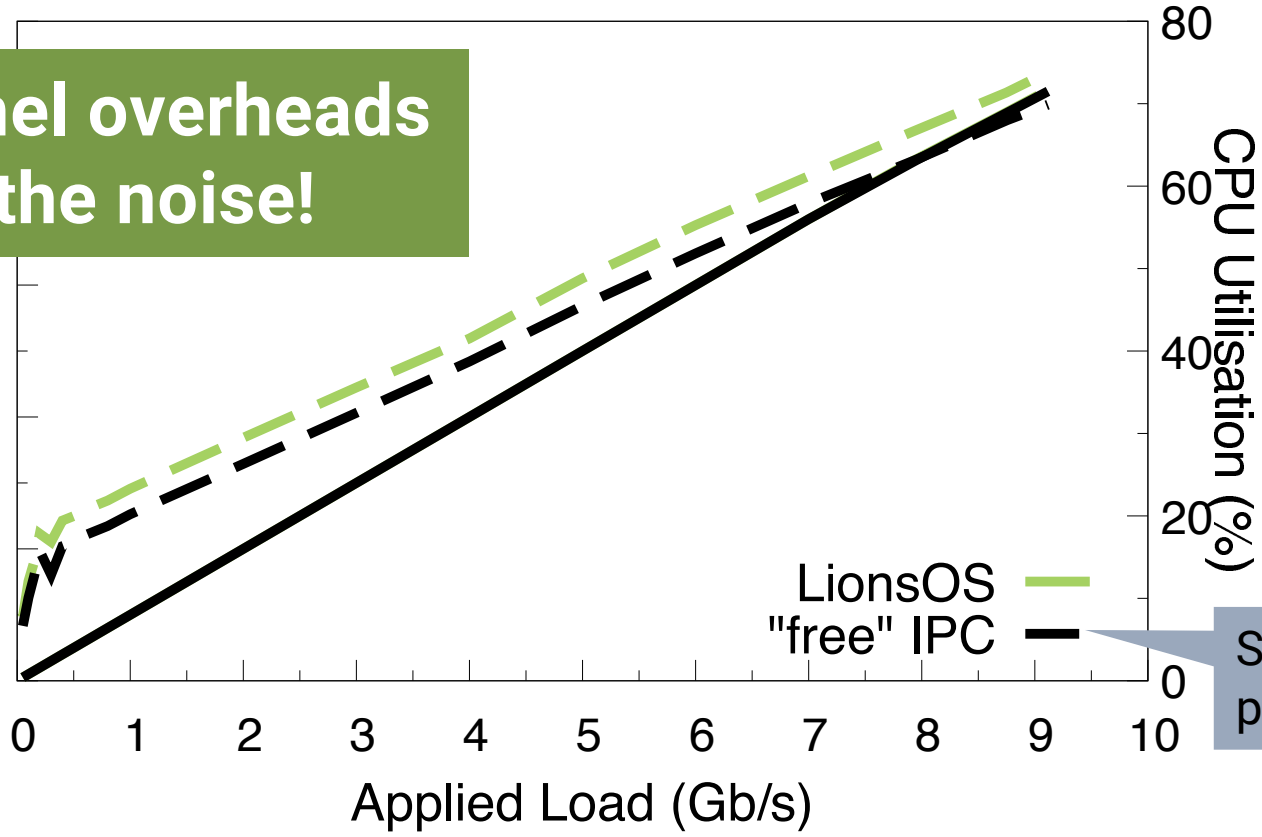
Performance: x86, 10Gb/s Eth, UDP



Multicore configuration

Syscall cost simulations (x86)

Microkernel overheads are in the noise!



Subtract 500cy per kernel entry

Single-core configuration

Why Is LionsOS Faster Than Linux?

Linux:
- NW driver: 3k lines
- NW system total: 1M lines

LionsOS executes less code!

LionsOS:
- NW driver: 400 lines
- Virtualiser: 160 lines
- Copier: 80 lines
- IP stack: much simpler, client library
- shared NW system total: < 1,000 lines

Why Is LionsOS So Simple?

Provide **exactly** the functionality needed, not more

Simple programming model:
- strictly sequential code (Microkit)
- event-based (Microkit)
- single-producer, single-consumer queues
- location transparency
- …

Static **architecture**, mostly static resource management

LionsOS Status

- Runs sel4.systems web site
- LionsOS-based firewall released as a community project
- Doom demo in TS lab 😀
- Runs on Arm, RISC-V, x86
- Native networking, storage, I2C, SPI, …
- Driver VMs for audio, 2G graphics, …
- Verification in progress

But I Want A **Real** OS!

Cost Of A Dynamic OS

- More complexity, larger code size

Might affect cache footprint?

- Double book-keeping, multiple server invocations

IPC overheads in the noise

- Higher startup times due to dynamic resource allocation

fork() will be the test!

- Resource revocation may require indirection

seL4 caps can be revoked without

- “Universal” policies are complex & costly

Do we need them?

Do We Need “Universal” Policies?

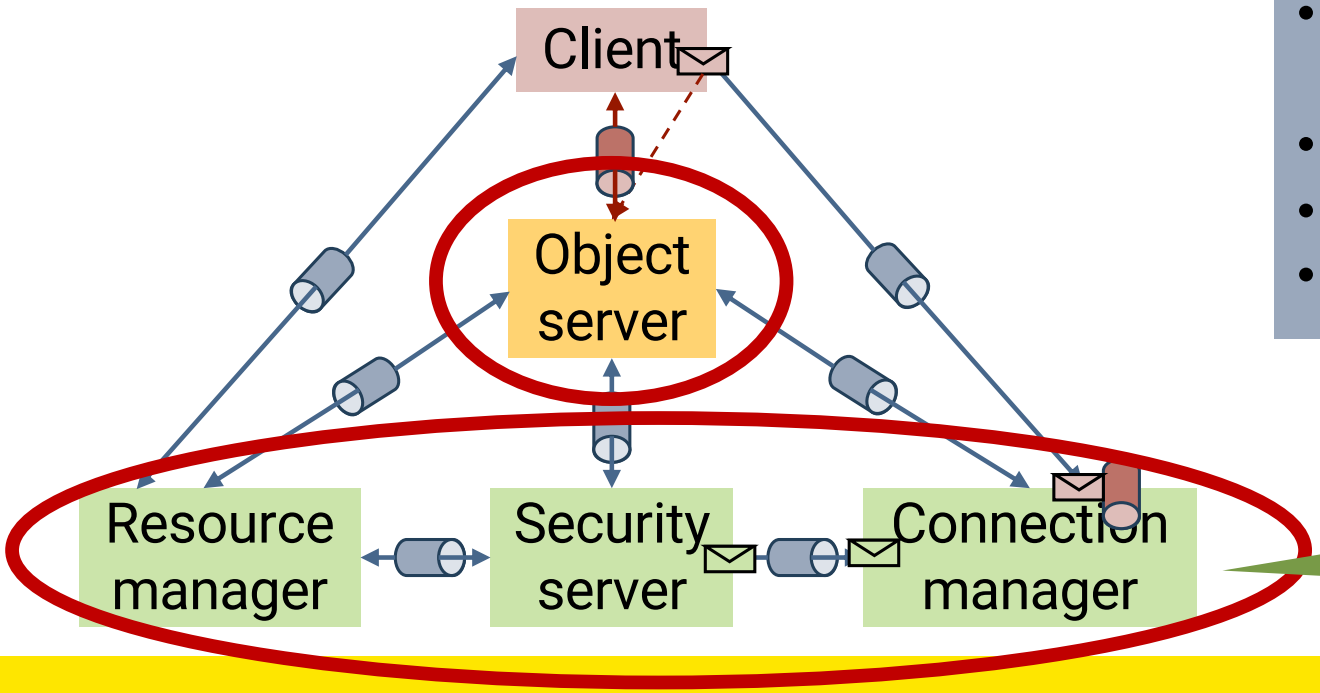
Claim:
- Systems rarely change policies on-the-fly
- Can change policy by replacing policy module

Keep configuration complexity off-line!

LionsOS experiment:
- Reload component with new policy implementation
- Cost: **17µs** on i.MX8M

Djawula: PoC Of General-Purpose OS

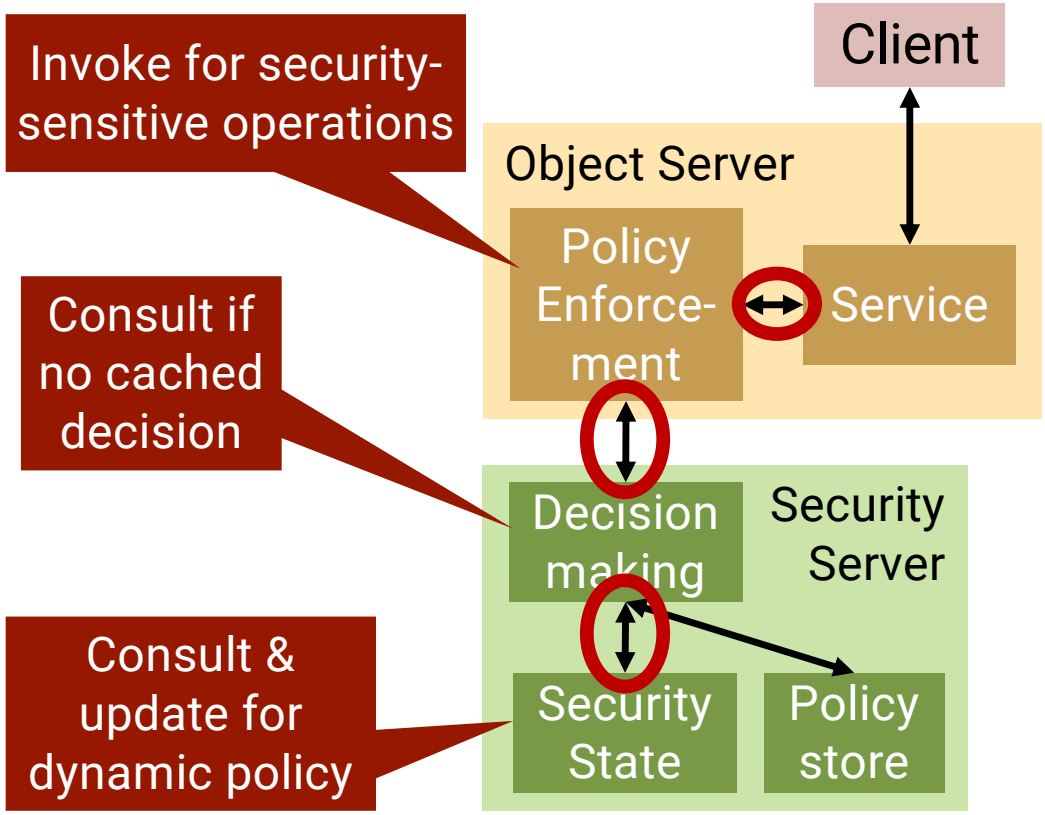
Aim: General-purpose OS that **provably** enforces a general security policy



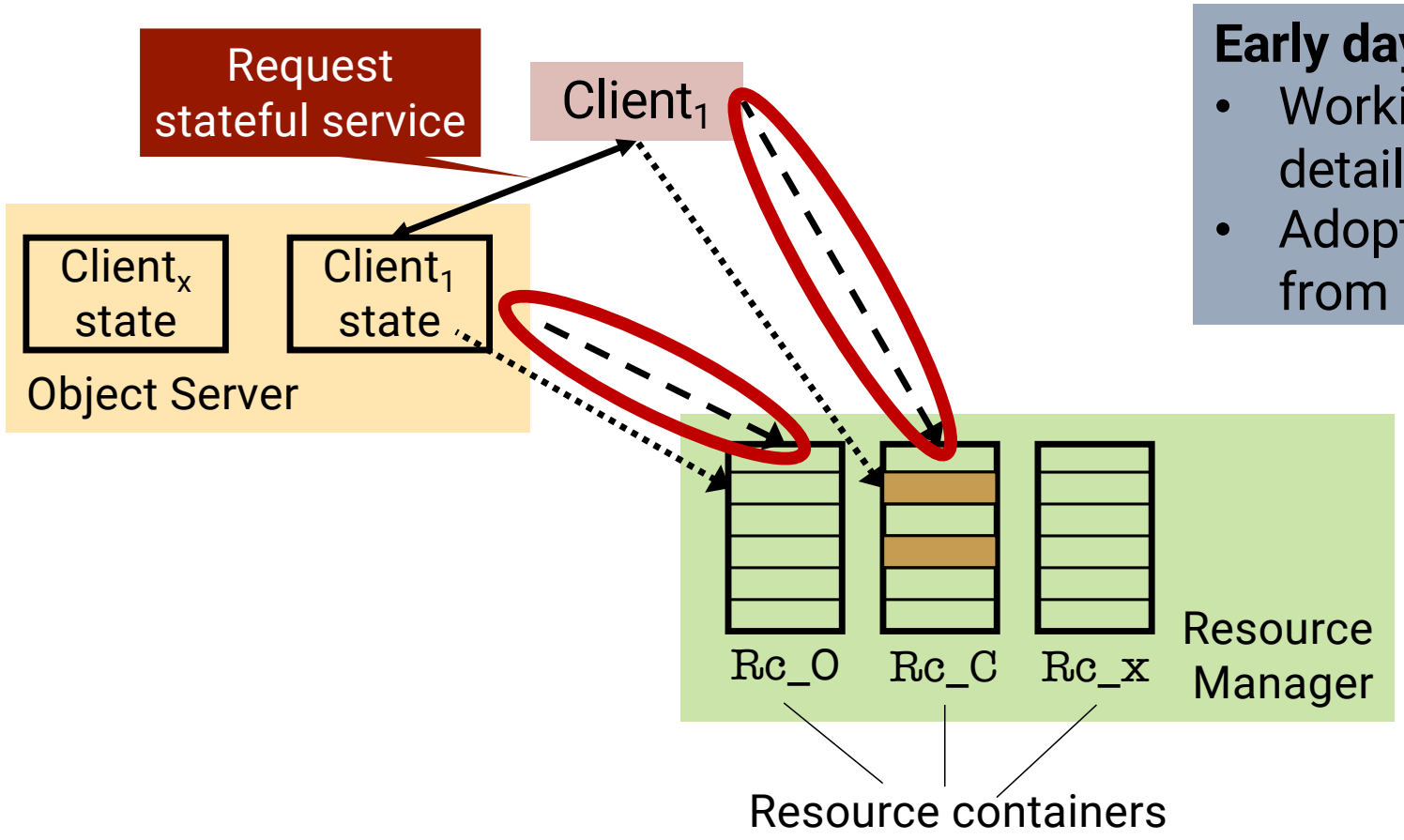
Requires:

- mandatory security-policy enforcement
- Security-policy diversity
- minimal TCB
- low-overhead enforcement

Core Ideas: Dynamic Enforcement



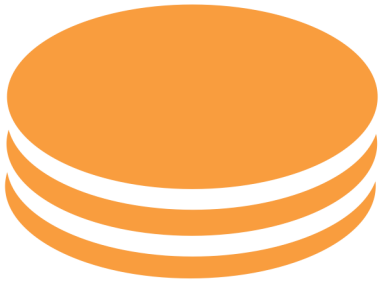
Core Ideas: Resource Donation



Early days:

- Working on framework, details of model
- Adopt components from LionsOS

Scaling Verification



PANCAKE
A Language for
Verified Systems Programming

Driver Dilemma

High seL4 verification costs partially due to C language

seL4 is one-off, justifies cost

Drivers are low-level, need C-like language

Better language would reduce cost

Drivers are commodity, must be cheap!

Lions OS

Idea:

1. Simplify drivers
2. Design verification-friendly systems language: Pancake
3. Automate (part of) verification

- Well-defined semantics
- Memory-safe

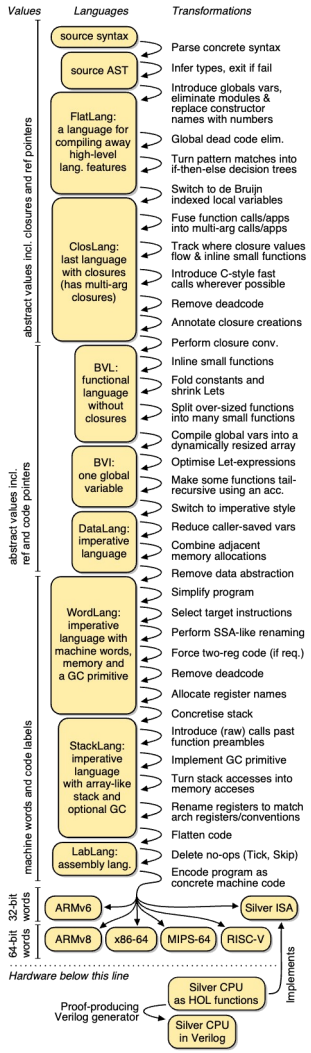
- Verified compiler
- de-compilation

CakeML: Verified Implementation of ML

- ✓ Mature functional language
- ✓ Large and active ecosystem of developers and users
- ✓ Code generation from abstract specs
- ❑ Managed ⇒ not suitable for systems code
- ✓ Used for verified application code

Re-use framework for new systems language: Pancake

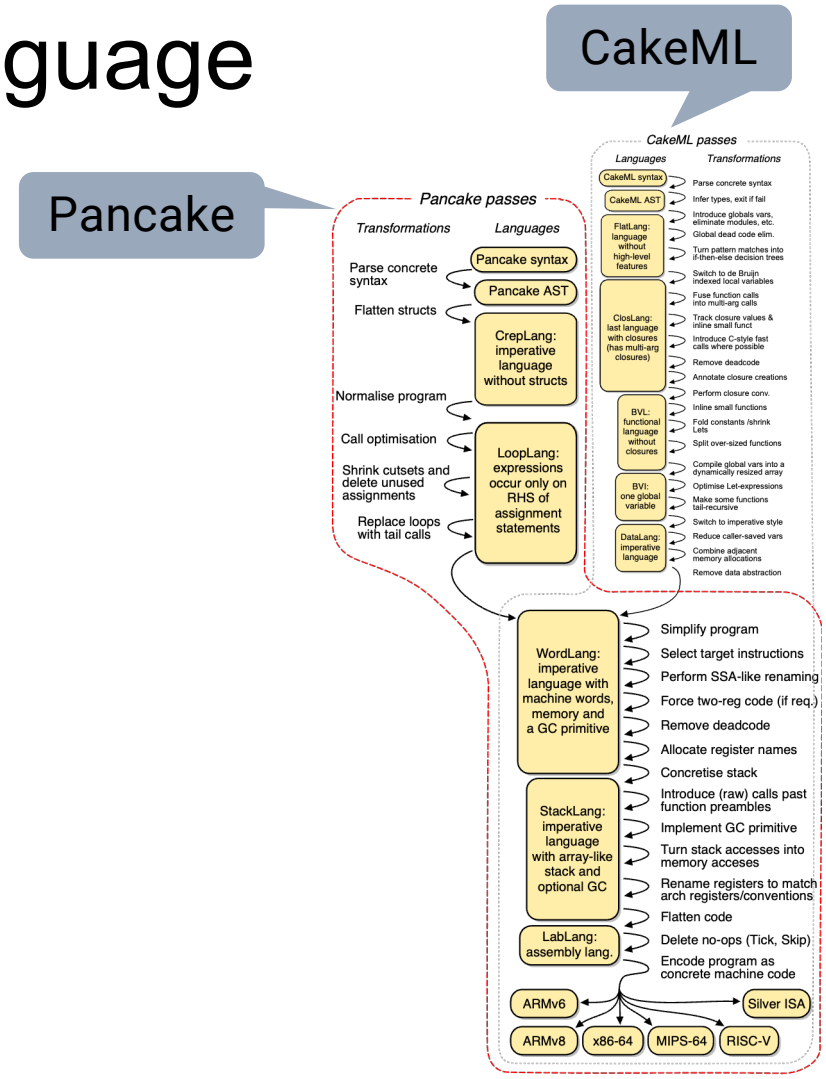
https://cakeml.org



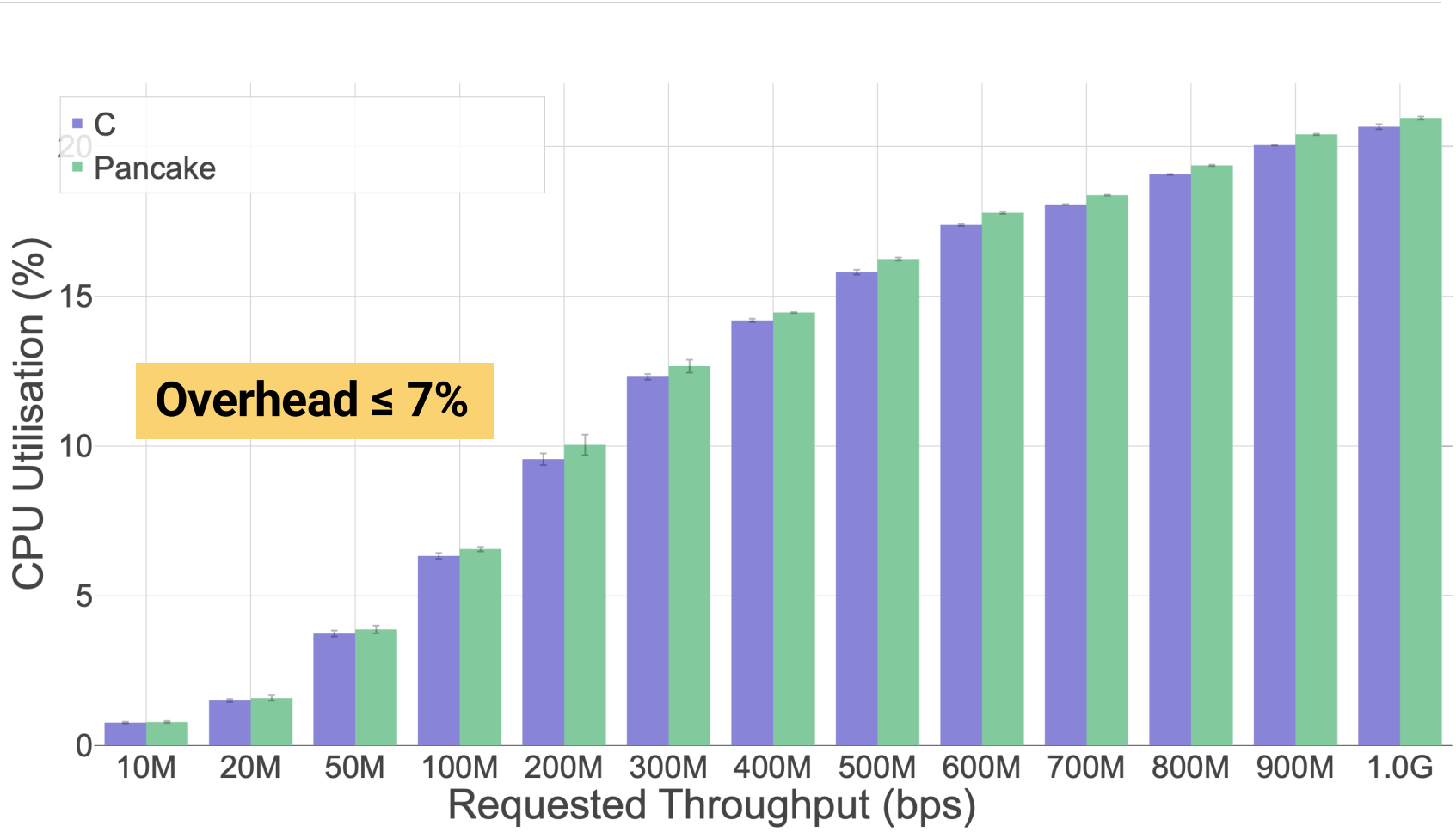
Pancake: New Systems Language

Approach:

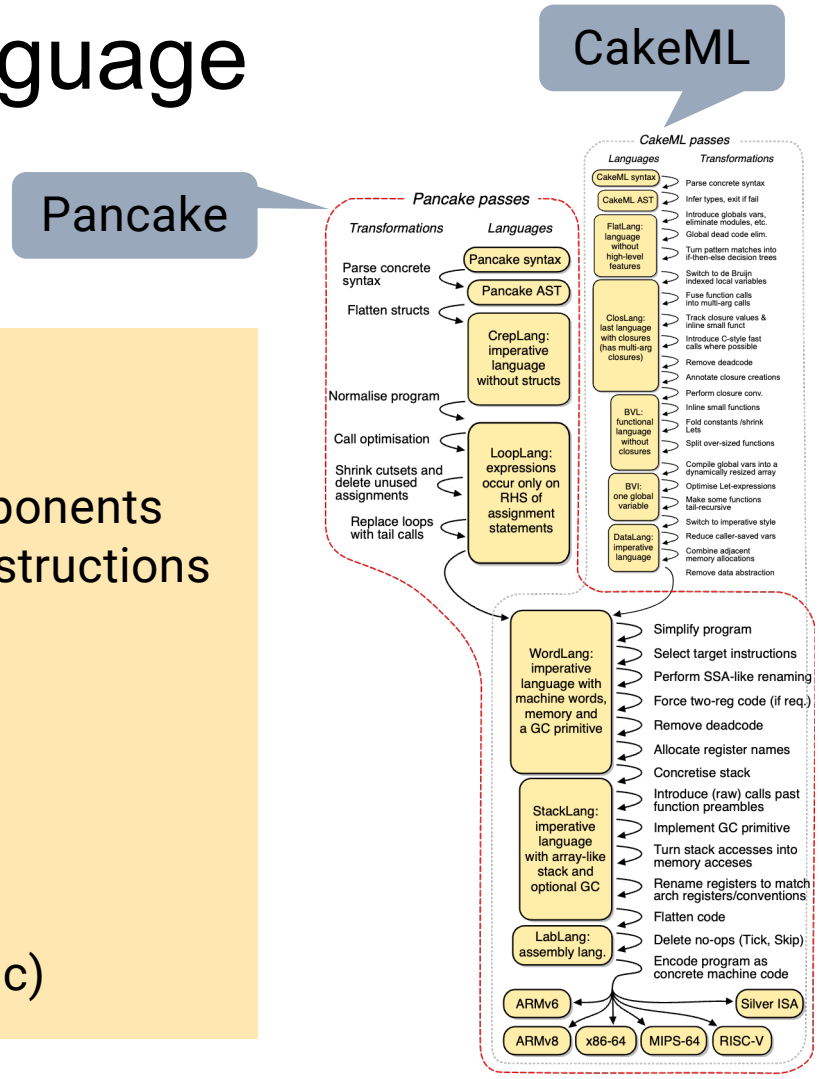
- Re-use lower part of CakeML compiler stack
- Get verified Pancake compiler quickly
- Retain mature framework/ecosystem



Performance: LionsOS Ethernet Driver



Pancake: New Systems Language



Status:

- "Usable" rump language, sufficient for LionsOS
  - Implemented drivers & other LionsOS components
  - Need C-escapes for cache management instructions
- **Verified compiler!**
- In progress:
  - Verification of components
  - More performance work
  - Decompilation from Pancake to HOL
  - Semantics for non-terminating programs
  - Efficient verification framework (Hoare logic)

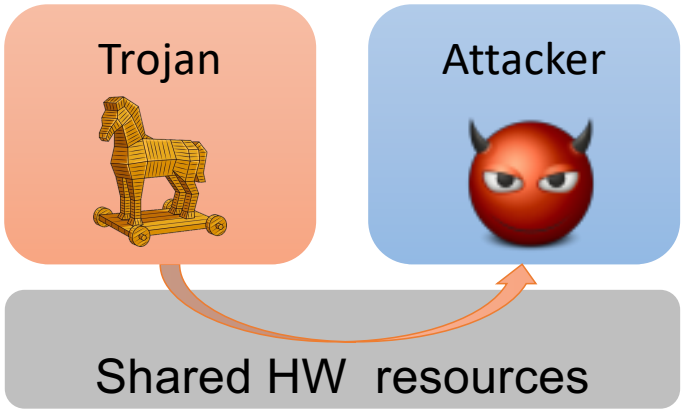
Time Protection

Principled Prevention of Microarchitectural Timing Channels

Refresh: Covert Timing Channels

- Created by contention for shared resource whose effect on timing can be monitored
  - Cache, network bandwidth, CPU load...

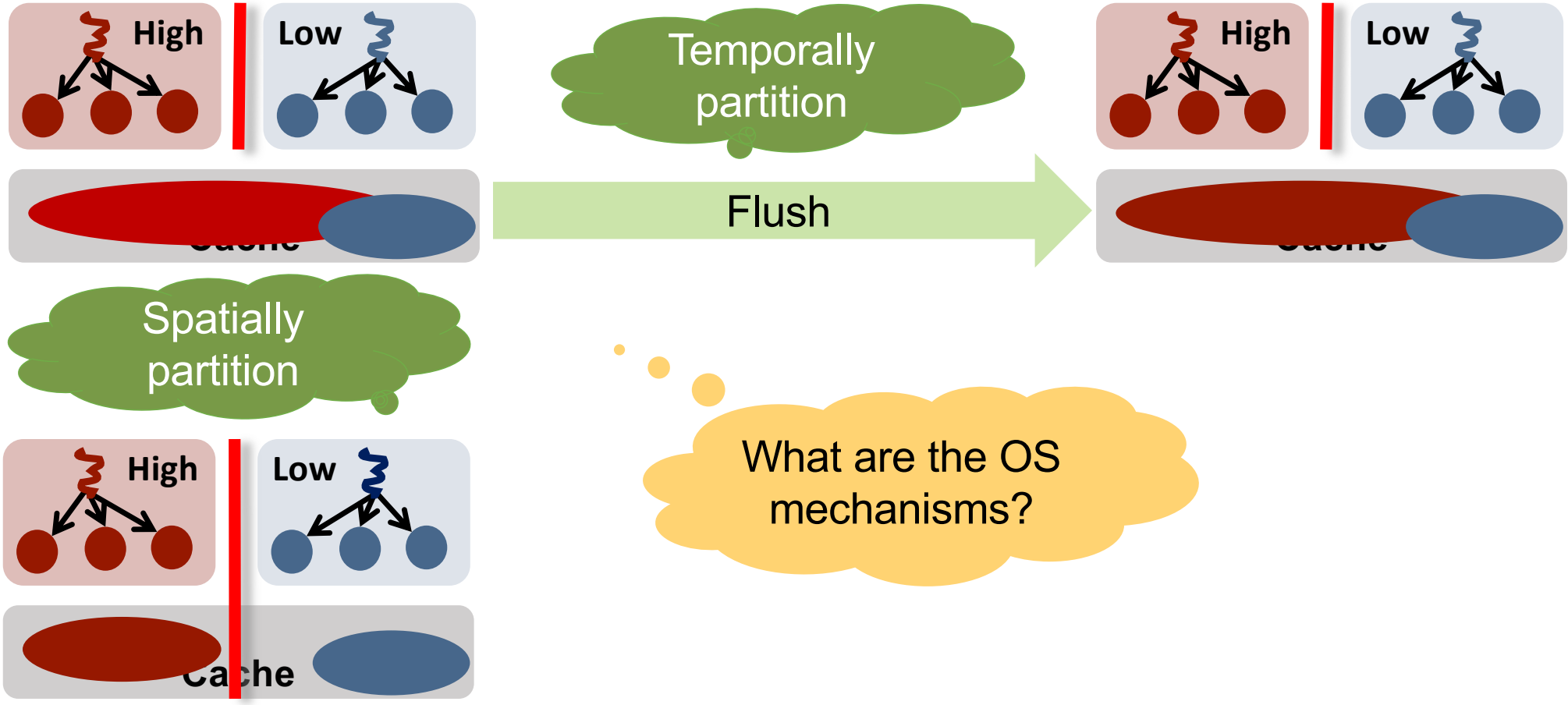
Usually based on some hidden state (e.g. caches)
⇒ timing/storage channel distinction is not deep!



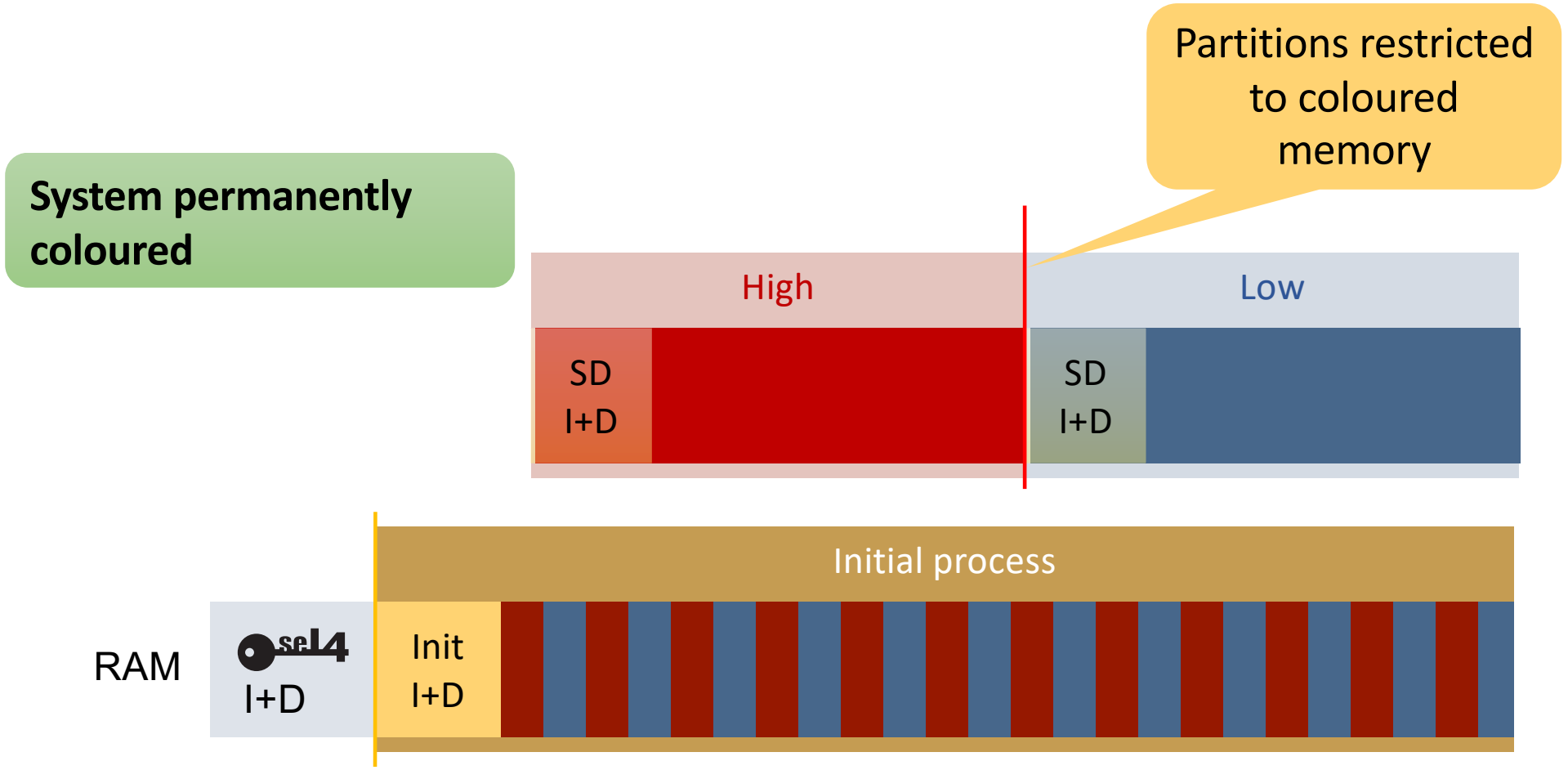
OS protection mechanisms:

- *Memory protection* for spatial isolation
- *Time protection* for temporal isolation

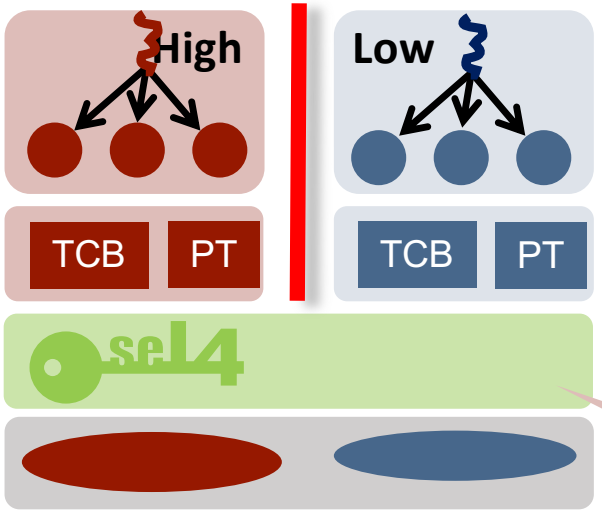
Time Protection: No Sharing of HW State



Spatial Partitioning: Cache Colouring

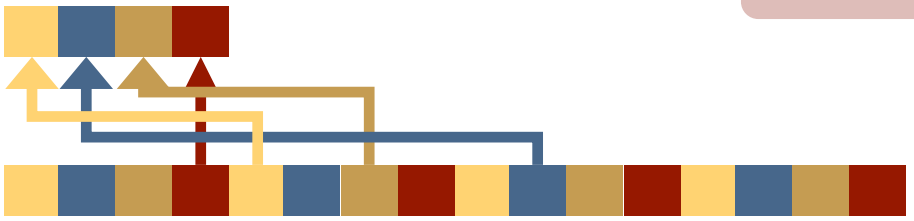


Spatial Partitioning: Cache Colouring



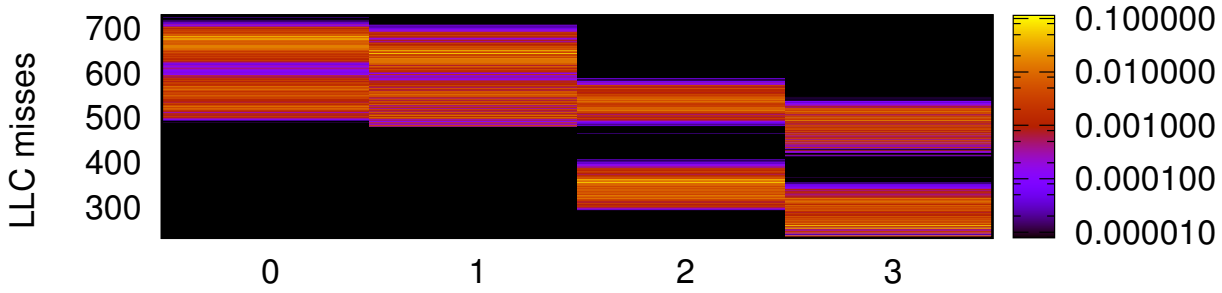
- Partitions get frame pools of disjoint colours
- seL4: userland supplies kernel memory
  ⇒ colouring userland colours kernel memory

Shared kernel image



Channel Through Kernel Code

Raw channel

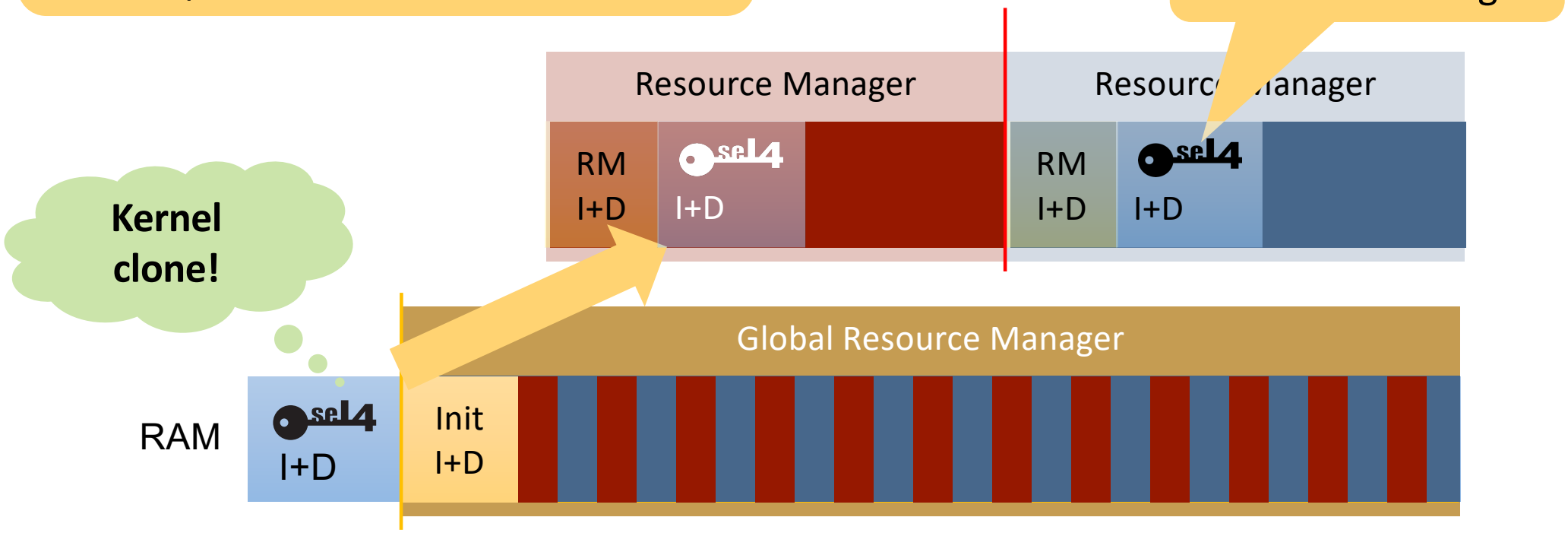


Channel matrix: Conditional probability of observing output signal (time) given input signal (system-call number)

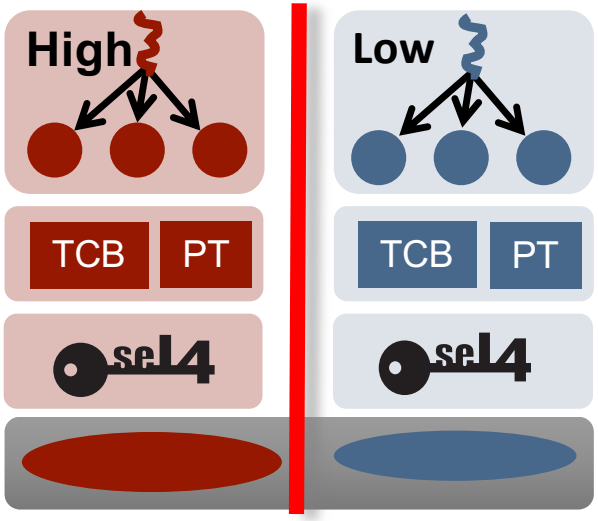
Colouring the Kernel

Remaining shared kernel data:
- Scheduler queue array & bitmap
- Few pointers to current thread state

Each partition has own kernel image

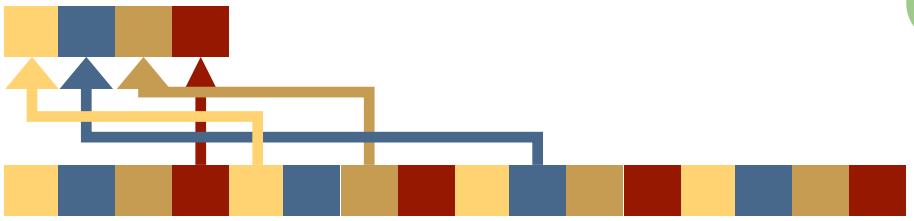


Spatial Partitioning: Cache Colouring



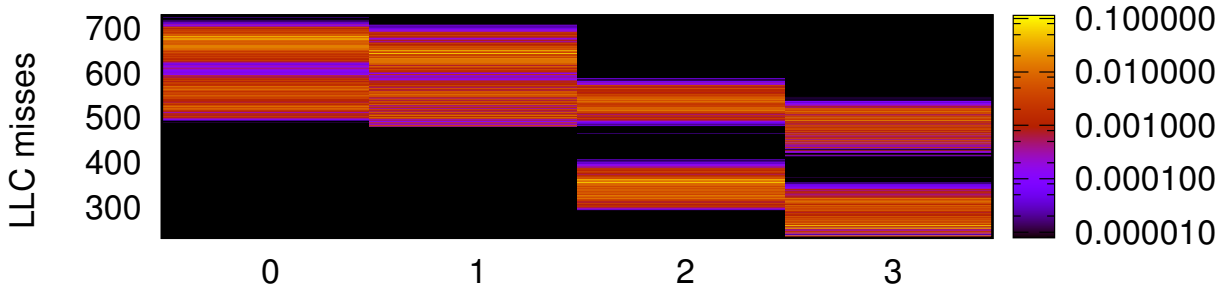
- Partitions get frame pools of disjoint colours
- seL4: userland supplies kernel memory
  ⇒ colouring userland colours kernel memory
- Per-partition kernel image to colour kernel

Must ensure deterministic access to remaining shared kernel state!

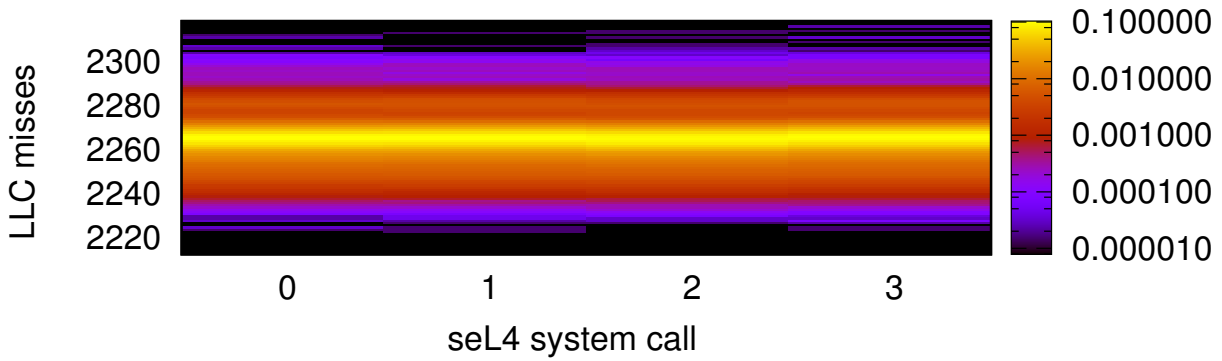


Channel Through Kernel Code

Raw channel



Channel with cloned kernel



Temporal Partitioning: Flush on Switch

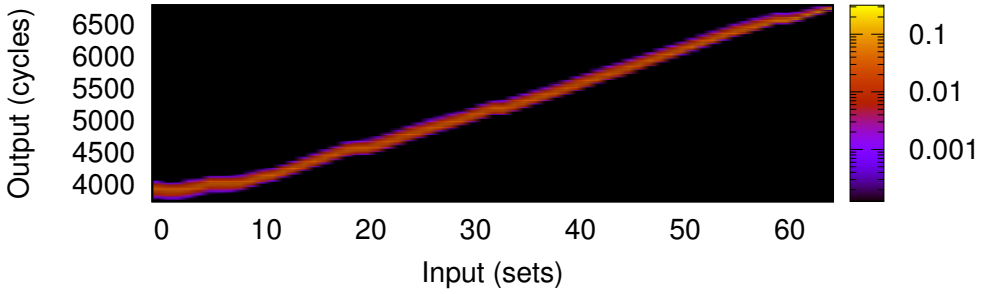
Must remove any history dependence!

2. Switch user context
3. Flush on-core state

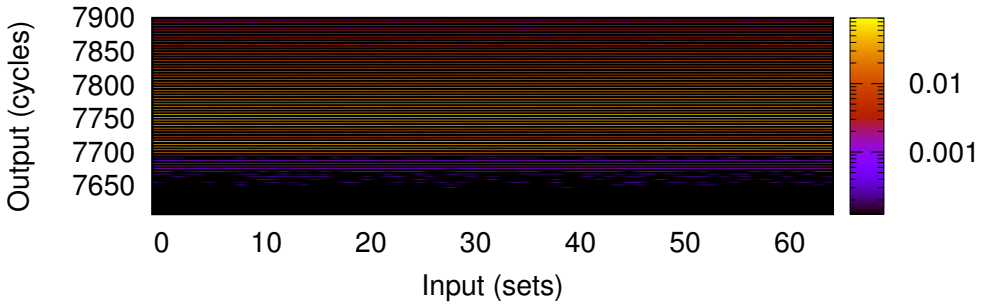
6. Reprogram timer
7. return

D-Cache Channel

Raw channel

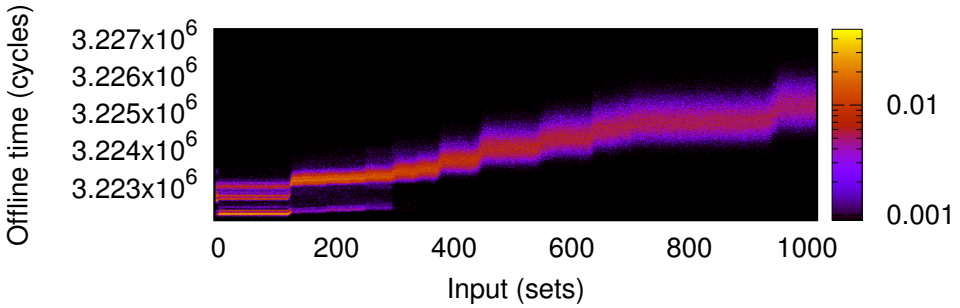


Channel with flushing



Flush-Time Channel

Raw channel



Temporal Partitioning: Flush on Switch

Must remove any history dependence!

1. T_0 = current_time()
2. Switch user context
3. Flush on-core state
4. Touch all shared data needed for return
5. while (T_0+WCET < current_time()) ;
6. Reprogram timer
7. return

Latency depends on prior execution!

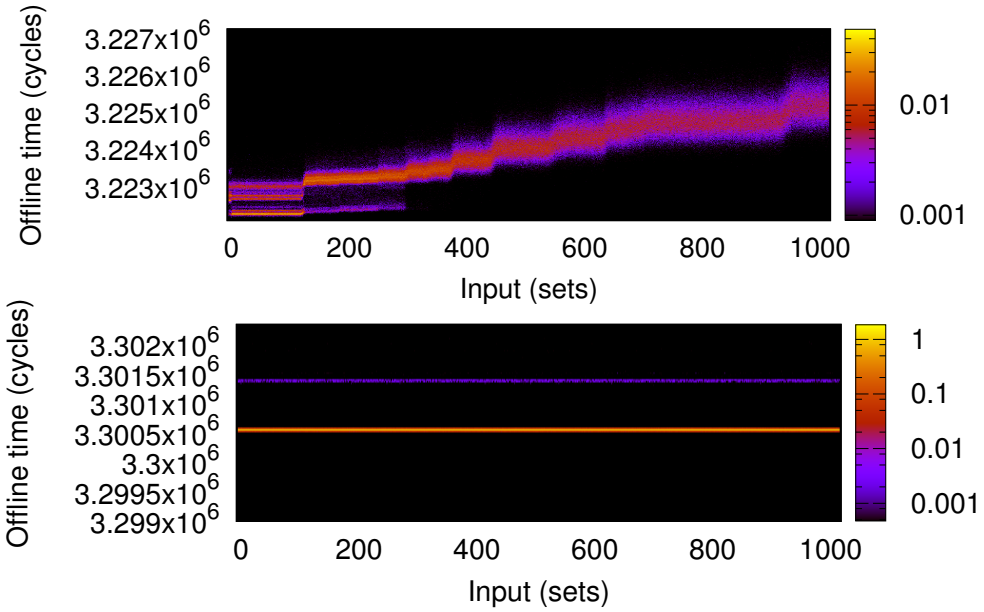
Time padding to remove dependency

Ensure deterministic execution

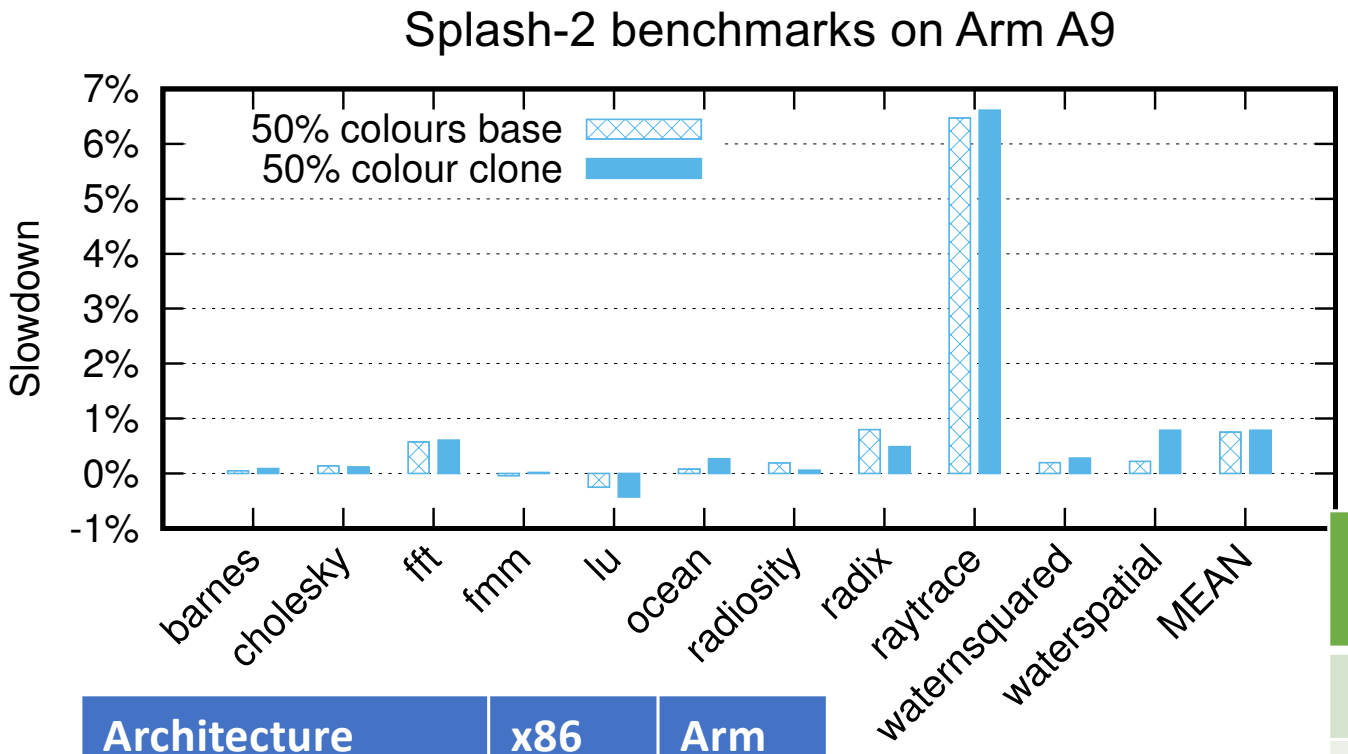
Flush-Time Channel

Raw channel

Channel with deterministic flushing



Performance Impact of Colouring



Architecture	x86	Arm
Mean slowdown	3.4%	1.1%

- Overhead mostly low
- Not evaluated is cost of not using super pages

[Ge et al., EuroSys'19]

Arch	seL4 clone	Linux fork+exec
x86	79 µs	257 µs
Arm	608 µs	4,300 µs

Temporal Partitioning: Flush State

Must remove any history dependence!

1. T_0 = current_time()
2. Switch user context
3. Flush on-core state
4. Touch all shared data needed for return
5. while (T_0+WCET < current_time()) ;
6. Reprogram timer
7. return

Problem: Processors do *not* provide mechanisms for resetting all microarchitectural state!

A New HW/SW Contract

For all shared microarchitectural resources:

aISA: augmented ISA

1. Resource must be spatially partitionable or flushable
2. Concurrently shared resources must be spatially partitioned
3. Resource accessed solely by virtual address must be flushed and not concurrently accessed

Cannot share HW threads across security domains!

4. Mechanisms must be sufficiently specified for OS to partition or reset
5. Mechanisms must be constant time, or of specified, bounded latency
6. Desirable: OS should know if resettable state is derived from data, instructions, data addresses or instruction addresses

[Ge et al., APSys'18]

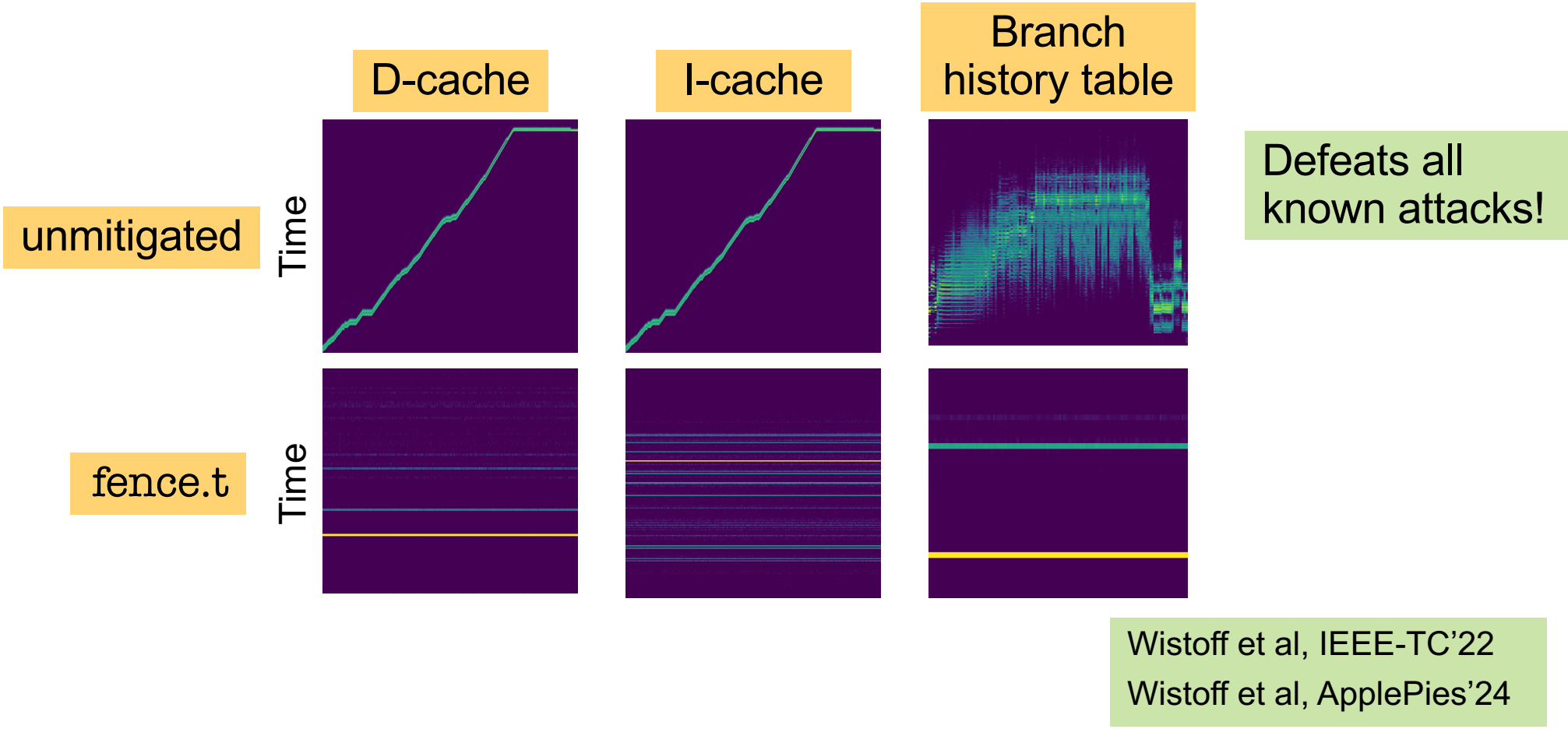
RISC-V To The Rescue: fence.t

fence.t instruction:

- Flush d-cache
- Reset all flip-lops that are **not** part of architected state

- Prototyped on in-order (CVA6) and OoO (C910) RISC-V processors
- Latency bounded by d-cache flush
- HW cost in the noise

fence.t Instruction on C910



Defeats all known attacks!

Wistoff et al, IEEE-TC'22
Wistoff et al, ApplePies'24

Can Time Protection Be Verified?

1. Correct treatment of spatially partitioned state:
   - Need hardware model that identifies all such state (augmented ISA)
   - To prove:
     No two domains can access the same physical state

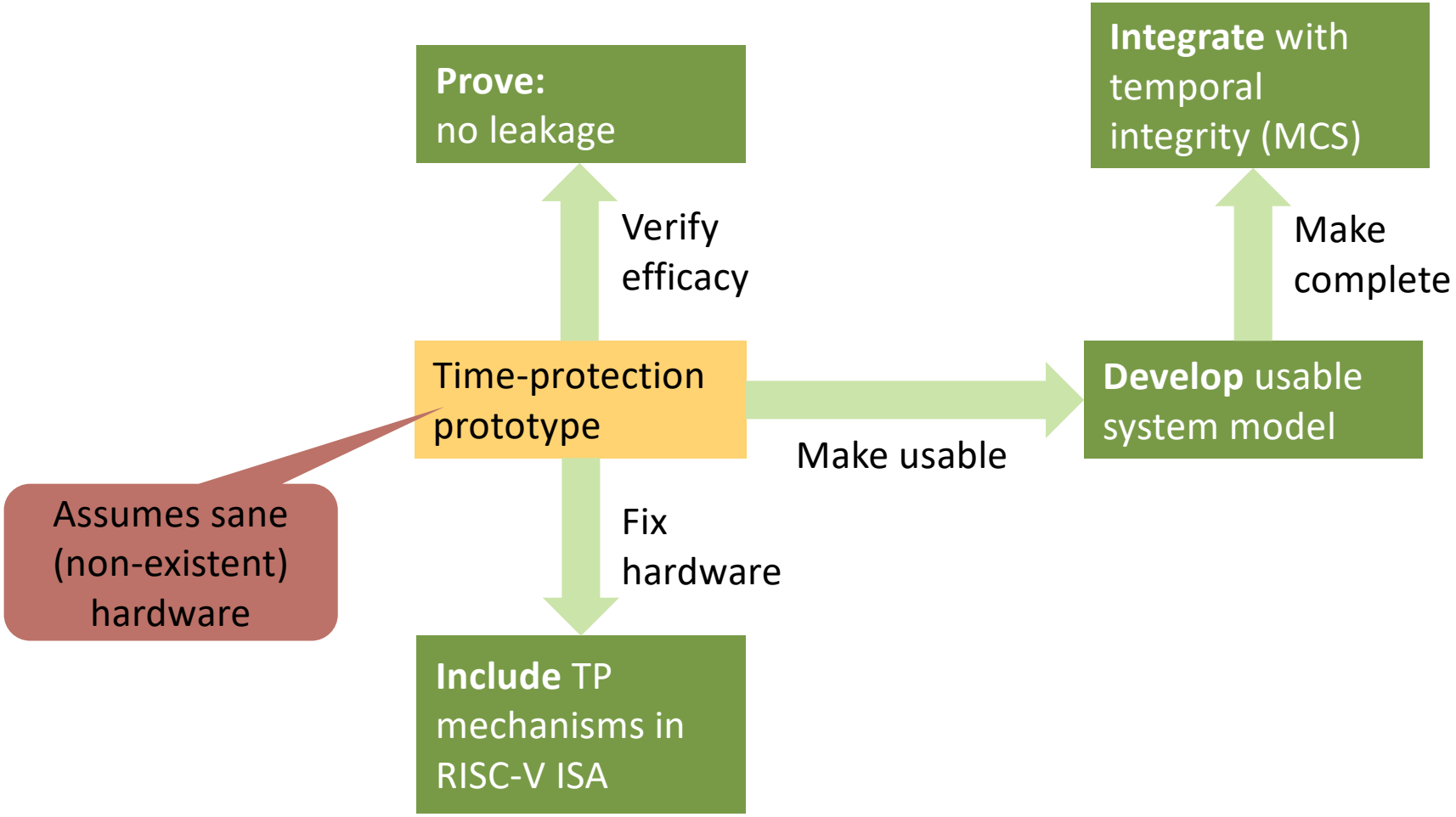
Functional property!

Transforms timing channels into storage channels!

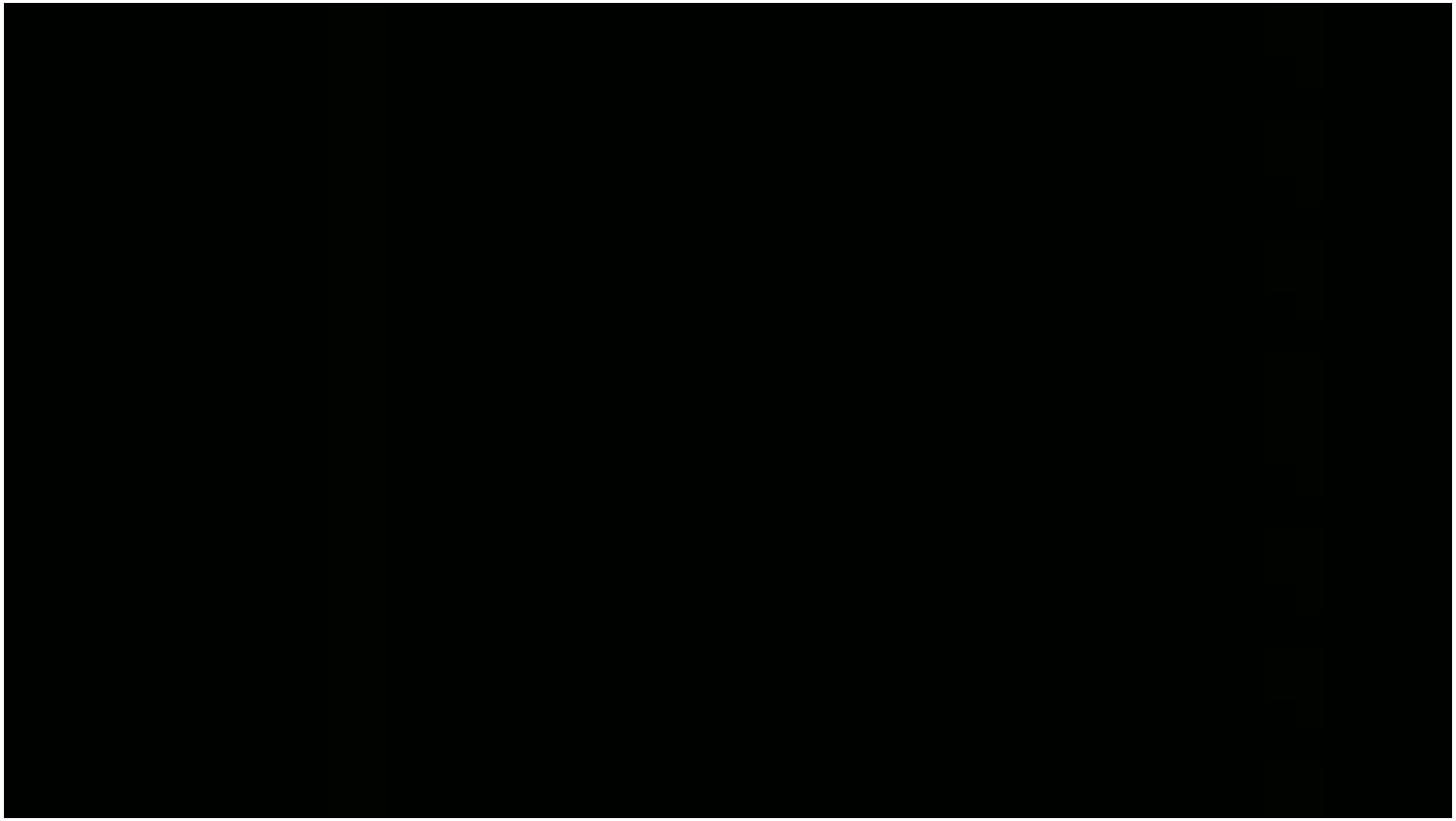
2. Correct flushing of time-shared state
   - Not trivial: eg proving all cleanup code/data are forced into cache after flush
     - Needs an actual cache model
   - Even trickier: need to prove padding is correct
     - … without explicitly reasoning about time!

Functional property!

Time Protection: On-Going Work



Real-World Use
Courtesy Boeing, DARPA



Thank you!

To the brave AOS students for their interest and dedication

To the world-class Trustworthy Systems team for making all possible

Please remember to do the myExperience survey

There'll also be a more detailed one we'll invite you to fill in