

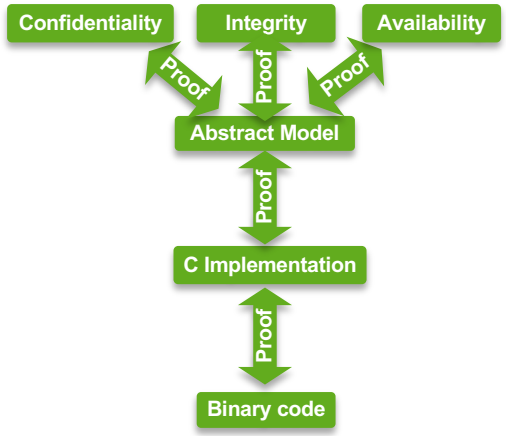
School of Computer Science & Engineering

COMP9242 Advanced Operating Systems

2026 T3 Week 10 Part 1

seL4: Verification & Performance

@GernotHeiser



Copyright Notice

These slides are distributed under the
Creative Commons Attribution 4.0 International (CC BY 4.0) License

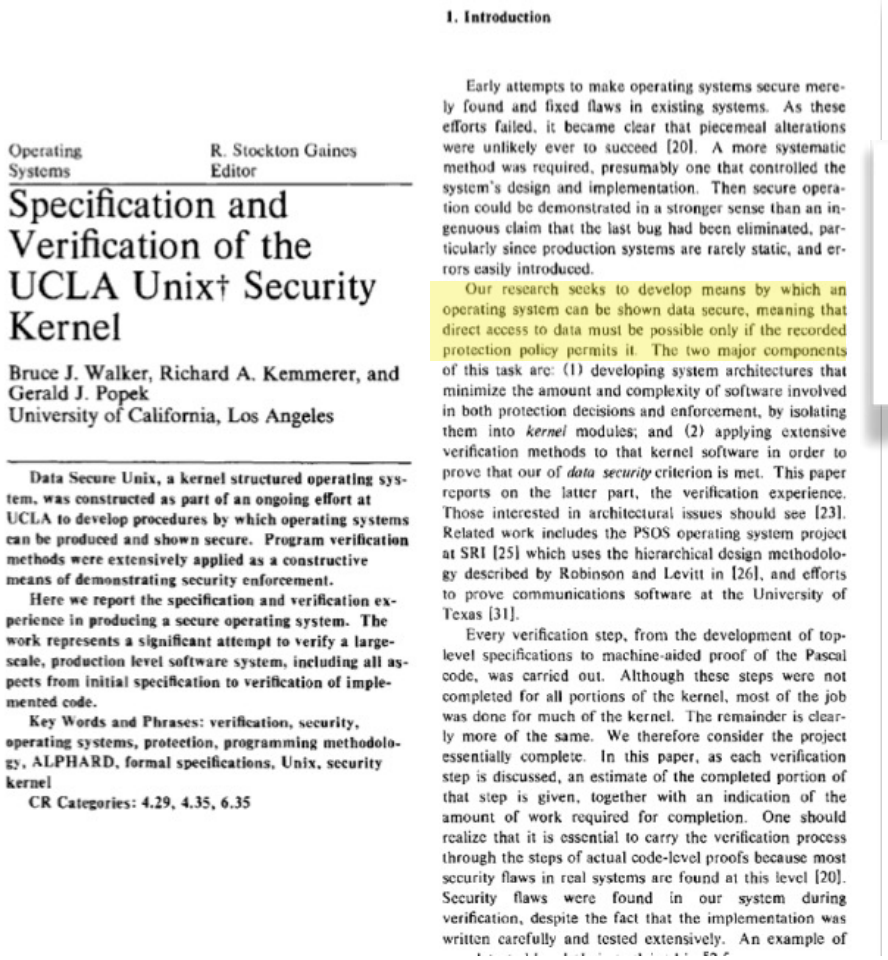
- You are free:
  - to share—to copy, distribute and transmit the work
  - to remix—to adapt the work
- under the following conditions:
  - **Attribution:** You must attribute the work (but not in any way that suggests that the author endorses you or your use of the work) as follows:

    "Courtesy of Gernot Heiser, UNSW Sydney"

The complete license text can be found at
http://creativecommons.org/licenses/by/4.0/legalcode

© Gernot Heiser 2019–25 – CC BY 4.0

A Thirty-Year Dream!



Operating Systems — R. Stockton Gaines, Editor

Specification and Verification of the UCLA Unix† Security Kernel

Bruce J. Walker, Richard A. Kemmerer, and Gerald J. Popek
University of California, Los Angeles

Data Secure Unix, a kernel structured operating system, was constructed as part of an ongoing effort at UCLA to develop procedures by which operating systems can be produced and shown secure. Program verification methods were extensively applied as a constructive means of demonstrating security enforcement.

Here we report the specification and verification experience in producing a secure operating system. The work represents a significant attempt to verify a large-scale, production level software system, including all aspects from initial specification to verification of implemented code.

Key Words and Phrases: verification, security, operating systems, protection, programming methodology, ALPHARD, formal specifications, Unix, security kernel

CR Categories: 4.29, 4.35, 6.35

1. Introduction

Early attempts to make operating systems secure merely found and fixed flaws in existing systems. As these efforts failed, it became clear that piecemeal alterations were unlikely ever to succeed [20]. A more systematic method was required, presumably one that controlled the system's design and implementation. Then secure operation could be demonstrated in a stronger sense than an ingenuous claim that the last bug had been eliminated, particularly since production systems are rarely static, and errors easily introduced.

Our research seeks to develop means by which an operating system can be shown data secure, meaning that direct access to data must be possible only if the recorded protection policy permits it. The two major components of this task are: (1) developing system architectures that minimize the amount and complexity of software involved in both protection decisions and enforcement, by isolating them into *kernel* modules; and (2) applying extensive verification methods to that kernel software in order to prove that our of *data security* criterion is met. This paper reports on the latter part, the verification experience. Those interested in architectural issues should see [23]. Related work includes the PSOS operating system project at SRI [25] which uses the hierarchical design methodology described by Robinson and Levitt in [26], and efforts to prove communications software at the University of Texas [31].

Every verification step, from the development of top-level specifications to machine-aided proof of the Pascal code, was carried out. Although these steps were not completed for all portions of the kernel, most of the job was done for much of the kernel. The remainder is clearly more of the same. We therefore consider the project essentially complete. In this paper, as each verification step is discussed, an estimate of the completed portion of that step is given, together with an indication of the amount of work required for completion. One should realize that it is essential to carry the verification process through the steps of actual code-level proofs because most security flaws in real systems are found at this level [20]. Security flaws were found in our system during verification, despite the fact that the implementation was written carefully and tested extensively. An example of

Our research seeks to develop means by which an operating system can be shown data secure, meaning that direct access to data must be possible only if the recorded protection policy permits it. The two major components

Communications of the ACM — February 1980, Volume 23, Number 2

© Gernot Heiser 2019–25 – CC BY 4.0

August 2009

A NICTA bejelentette a világ első, formális módszerekkel igazolt,

Slashdot

Stories Recent Popular Searc

Slashdot is powered by your subm

Technology: World's Firs

Posted by Soulskill on Thursday Aug
from the wait-for-it dept.

An anonymous reader writes

> "Operating systems usually have
> and so forth are known by almos
> to prove that a particular OS ker
> formally verified, and as such it
> researchers used an executable
> the Isabelle theorem prover to ge
> matches the executable and the

Does it run Linux? "We're pleased to



New Scientist
Saturday 29/8/2009
Page: 21
Section: General News
Region: National
Type: Magazines Science / Technology
Size: 196.31 sq.cms.
Published: -----S-

The ultimate way to keep your computer safe from harm

FLAWS in the code, or "kernel", that sits at the heart of modern computers leave them prone to occasional malfunction and vulnerable to attack by worms and viruses. So the development of a secure general-purpose microkernel could pave the

just mathematics, and you can reason about them mathematically," says Klein.

His team formulated a model with more than 200,000 logical steps which allowed them to prove that the program would always behave as its

eredmenyekeppen pedig egy olyan megbiznatosagot kapnak a szortvertől, amely e

© Gernot Heiser 2019–25 – CC BY 4.0

MIT Technology Review

LISTS | INNOVATORS UNDER 35 | DISRUPTIVE COMPANIES

MIT Technology Review

10 BREAKTHROUG TECHNOLOGIES

Crash-Proof Code

Making critical software safer

comments

WILLIAM BULKELEY

May/June 2011





© Gernot Heiser 2019–25 – CC BY 4.0

Today's Lecture

- Assurance and verification
  - Common Criteria
  - Formal verification
- seL4
  - Design principles & verification
  - Verification limitations & present status
- Performance of seL4-based systems
- seL4 on multicore

© Gernot Heiser 2019–25 – CC BY 4.0

Assurance and Verification

© Gernot Heiser 2019–25 – CC BY 4.0

Refresher: Assurance and Formal Verification

- **Assurance**:
  - systematic evaluation and testing
  - essentially an intensive and onerous form of quality assurance
- **Formal verification**:
  - mathematical proof
- **Certification**: independent examination
  - confirming that the assurance or verification was done right

Assurance and formal verification aim to establish correctness of
- mechanism design
- mechanism implementation

© Gernot Heiser 2019–25 – CC BY 4.0

Assurance: Substantiating Trust

- Specification
  - Unambiguous description of desired behaviour

Informal (English) or formal (maths)

- System design
  - Justification that it meets specification

Compelling argument or formal proof

- Implementation
  - Justification that it implements the design

Code inspection, rigorous testing, proof

- Maintenance
  - Justifies that system use meets assumptions

© Gernot Heiser 2019–25 – CC BY 4.0

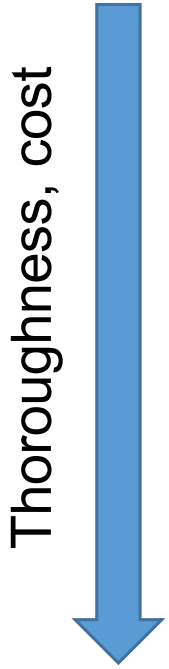
Common Criteria

Common Criteria for IT Security:

- ISO standard [ISO/IEC 15408, 99], for general use
- Evaluates QA used to ensure systems meet their requirements
- Developed out of the famous US DOD “Orange Book”: *Trusted Computer System Evaluation Criteria* [1985]

© Gernot Heiser 2019–25 – CC BY 4.0

CC: Evaluation Assurance Levels



Level	Requirements	Specification	Design	Implementation
EAL1	not evaluated	**Informal**	not eval	not evaluated
EAL2	not evaluated	**Informal**	**Informal**	not evaluated
EAL3	not evaluated	**Informal**	**Informal**	not evaluated
EAL4	not evaluated	**Informal**	**Informal**	not evaluated
EAL5	not evaluated	**Semi-Formal**	**Semi-Formal**	**Informal**
EAL6	**Formal**	**Semi-Formal**	**Semi-Formal**	**Informal**
EAL7	**Formal**	**Formal**	**Formal**	**Informal**

© Gernot Heiser 2019–25 – CC BY 4.0

COTS OS Certifications

- EAL3:
  - 2010 Mac OS X (10.6)
- EAL4:
  - 2003: Windows 2000
  - 2005: SuSE Enterprise Linux
  - 2006: Solaris 10 (EAL4+)
    - against CAPP (an EAL3 PP!)
  - 2007: Red Hat Linux (EAL4+)
  - 2025: L4Re (L4 microkernel, EAL4+)
- EAL6:
  - 2008: Green Hills INTEGRITY-178B (EAL6+)
    - relatively simple PPC-based hardware platform
- EAL7:
  - 2019: Prove & Run PROVENCORE
    - TEE OS for Arm TrustZone

Get regularly hacked!

Certification only valid for the specific target of evaluation (TOE)
Void if anything changes!

© Gernot Heiser 2019–25 – CC BY 4.0

Common Criteria Limitations

Effectively dead in 5-Eyes defence

- Very expensive
  - rule of thumb: EAL6+ costs $1K/LOC [Green Hills] design-implementation-evaluation-certification
- Too much focus on development process
  - rather than the product that was delivered
  - “Evaluating paperwork, not the product” [N Daughety, AFRL]
- Lower EALs of little practical use for OSes
  - c.f. COTS OS EAL4 certifications
- Commercial Evaluation Facilities licenses rarely revoked
  - Leads to potential “race to the bottom” [Anderson & Fuloria, 2009]

© Gernot Heiser 2019–25 – CC BY 4.0

Formal Verification

Prove properties about a mathematical model of a system

Automatic ("push-button") techniques

- **Model checking / abstract interpretation / SMT**
- **Systematic exploration of system state space**

❑ Cannot generally prove code correct
  - Proves specific properties
  - Functional correctness in simple cases

❑ Generally have to
  - over-approximate (false positives), or
  - under-approximate (false negatives, unsound)

❑ Suffers state-space explosion

✓ Can scale to large code bases

Interactive techniques:

- **Theorem proving**
- **Proofs about state spaces**

✓ Can deal with large (even infinite) state spaces

✓ Can prove functional correctness against a spec

❑ Very labour-intensive

Automatically proved functional correctness of simple systems using SMT solvers [Hyperkernel, SOSP'17; Atmosphere, SOSP'25; LionsOS Ethernet driver, PLOS'26]

© Gernot Heiser 2019–25 – CC BY 4.0



© Gernot Heiser 2019–25 – CC BY 4.0



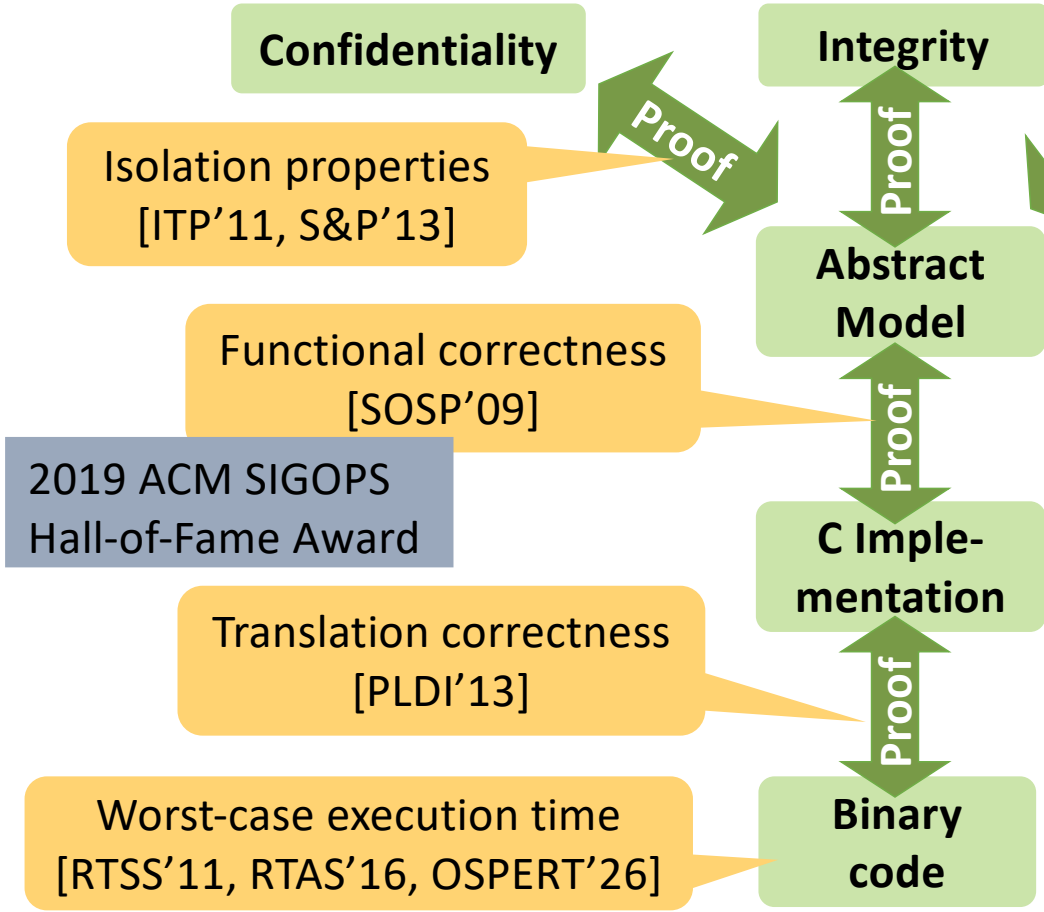
Remember: Design Principles

- Fully delegable access control via capabilities
- All resource management is subject to user-defined policies
  - Applies to kernel resources too!
- Performance on par with best-performing L4 kernels
  - Prerequisite for real-world deployment!
- Suitability for real-time use
  - Important for safety-critical systems
- Suitable for *formal verification*
  - Requires small size, avoid complex constructs

Largely in line with traditional L4 approach!

© Gernot Heiser 2019–25 – CC BY 4.0

Proving Security and Safety (Armv6/7)



Still most compre-hensive verification

Still only verified capability-based OS

Exclusions (at present, Armv7):

- Kernel initialisation not yet verified
- MMU & caches modelled abstractly
- Multicore in progress (multikernel)
- Covert *timing* channels not precluded

© Gernot Heiser 2019–25 – CC BY 4.0

Security Is No Excuse For Bad Performance!

Cost	seL4	Fiasco.OC	Zircon
IPC RT latency (cycles)	986	2717	8157
Mand. HW cost (cycles)	790	790	790
Abs. overhead (cycles)	196	1972	7367
Rel. overhead (%)	25	240	930

Hardware cost dominates

SW overheads dominate

Round-trip, cross-address-space IPC on x64 (Intel Skylake)

Operation	1-way	RT
SYSCALL	82	164
SWAPGS	2×26	104
Switch PT	186	372
SYSRET	75	150
Total	**395**	**790**

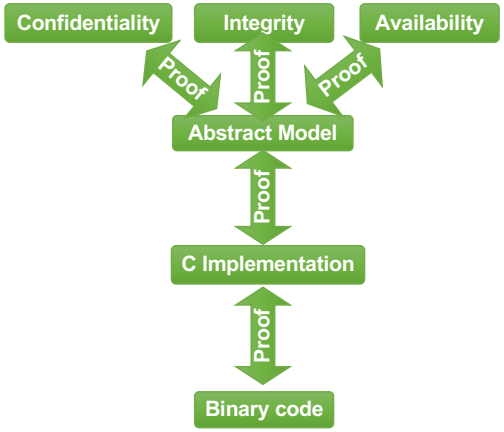
Source: Zeyu Mi, Dingji Li, Zihan Yang, Xinran Wang, Haibo Chen: "SkyBridge: Fast and Secure Inter-Process Communication for Microkernels", EuroSys, April 2019

© Gernot Heiser 2019–25 – CC BY 4.0

Verification Limitations

© Gernot Heiser 2019–25 – CC BY 4.0

Verification Assumptions

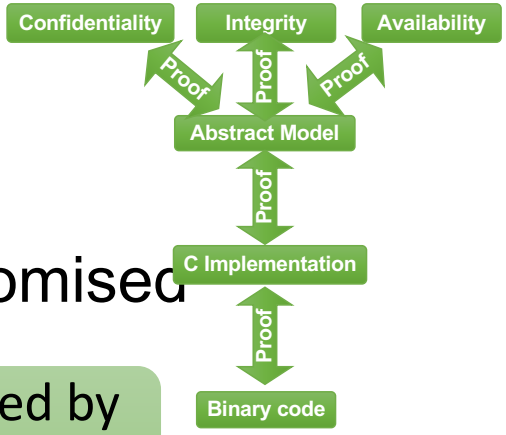


1. Hardware behaves as expected
   - Formalised hardware-software contract (ISA)
   - Hardware implementation free of bugs, Trojans, …
2. Spec matches expectations
   - Can only prove “security” if specify what “security” means
   - Spec may not be what we think it is
3. Proof checker is correct
   - Isabel/HOL checking core that validates proofs against logic

With binary verification do **not** need to trust C compiler!

© Gernot Heiser 2019–25 – CC BY 4.0

Present Verification Limitations



- Not verified boot code
  - **Assume** it leaves kernel in safe state
- Caches/MMU presently modeled at high level / axiomised
- Multicore kernel not verified
  - … in progress (multikernel)
- Not proved any temporal properties
  - Presently not proved scheduler observes priorities, properties needed for RT
  - WCET analysis subject to sound instruction latencies
  - No proofs about timing channels (yet)

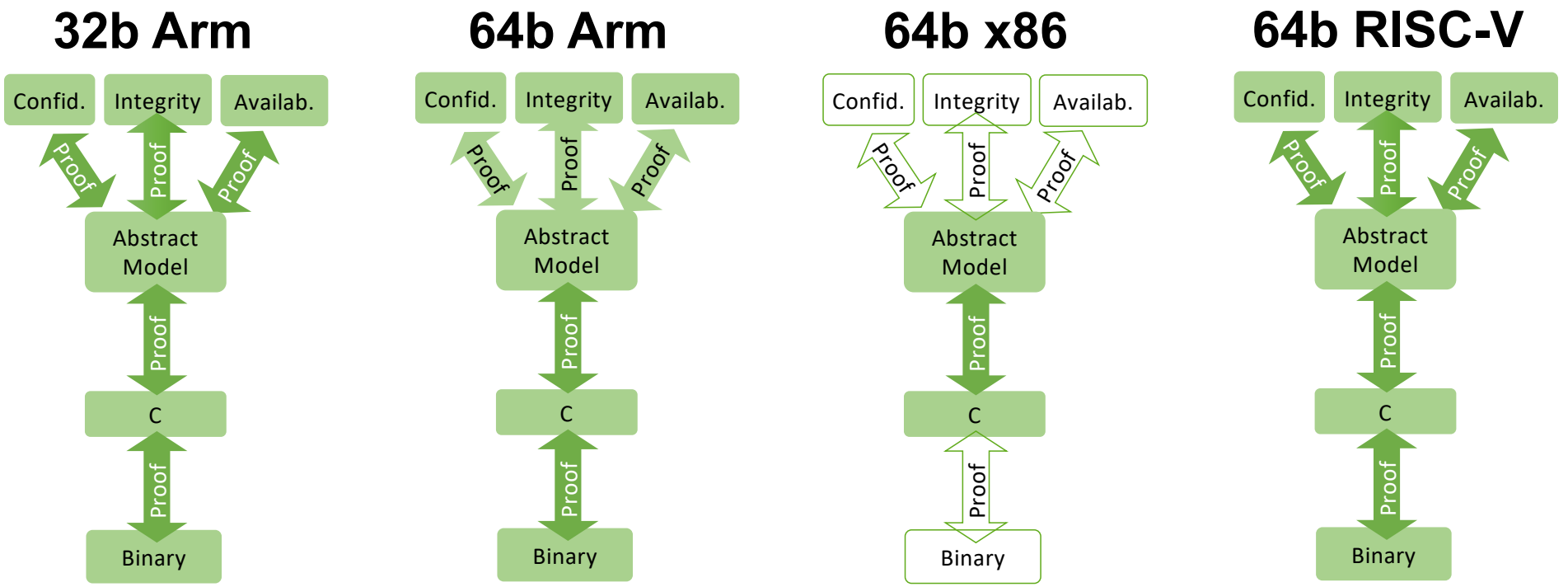
MMU model finished by PhD but not integrated

Available for open RISC-V cores!

Present research!

© Gernot Heiser 2019–25 – CC BY 4.0

Present Status



© Gernot Heiser 2019–25 – CC BY 4.0



Common Criteria?

Level	Requirements	Specification	Design	Implementation
EAL1	not evaluated	**Informal**	not eval	not evaluated
EAL2	not evaluated	**Informal**	**Informal**	not evaluated
EAL3	not evaluated	**Informal**	**Informal**	not evaluated
EAL4	not evaluated	**Informal**	**Informal**	not evaluated
EAL5	not evaluated	**Semi-Formal**	**Semi-Formal**	**Informal**
EAL6	**Formal**	**Semi-Formal**	**Semi-Formal**	**Informal**
EAL7	**Formal**	**Formal**	**Formal**	**Informal**
	**Formal**	**Formal**	**Formal**	**Formal**

© Gernot Heiser 2019–25 – CC BY 4.0

Performance of seL4-based Systems

Particularly LionsOS

© Gernot Heiser 2019–25 – CC BY 4.0

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%

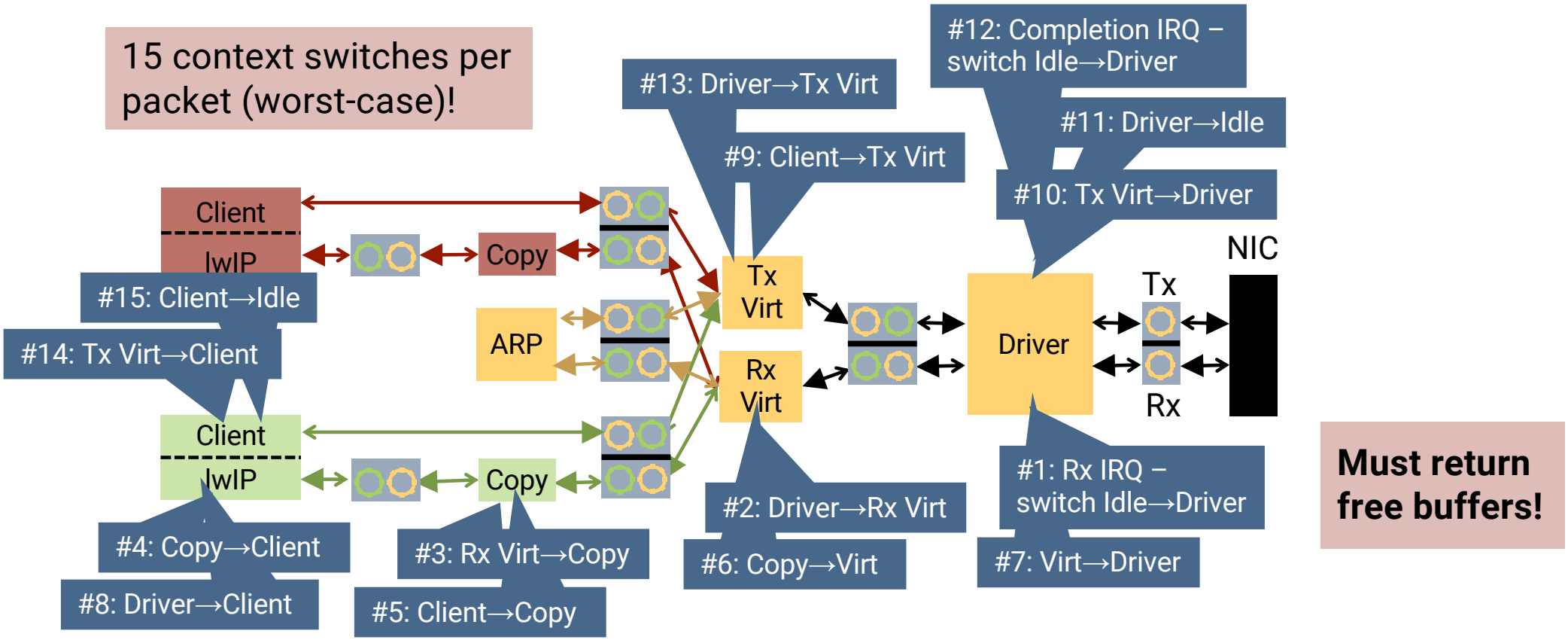
Why would anyone care?

© Gernot Heiser 2019–25 – CC BY 4.0



Packet Round-Trip Context Switches

15 context switches per packet (worst-case)!

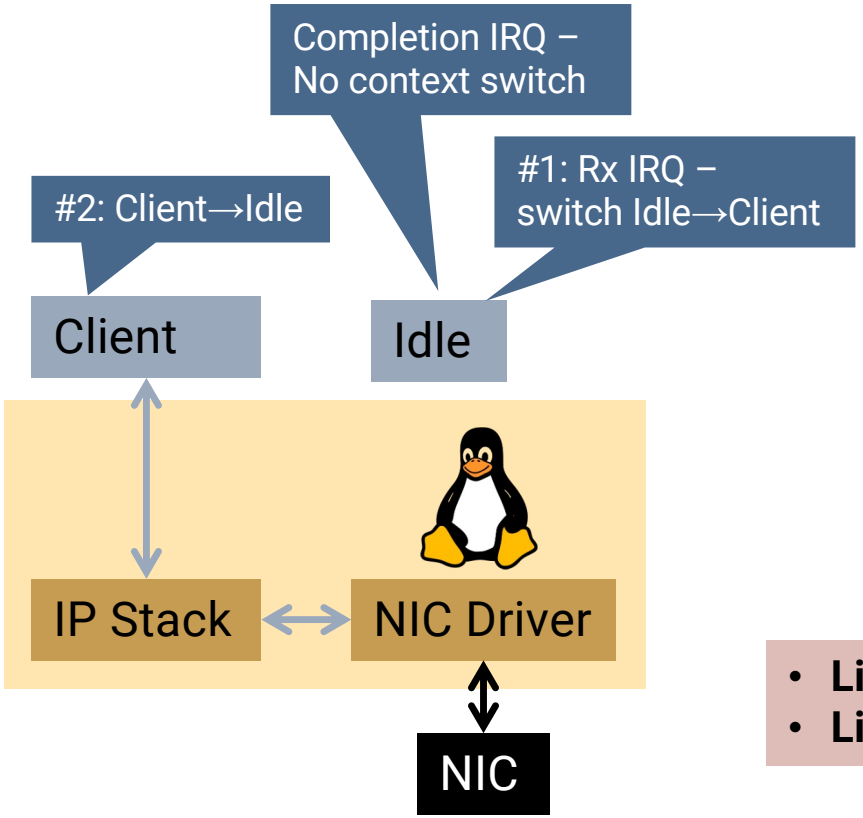


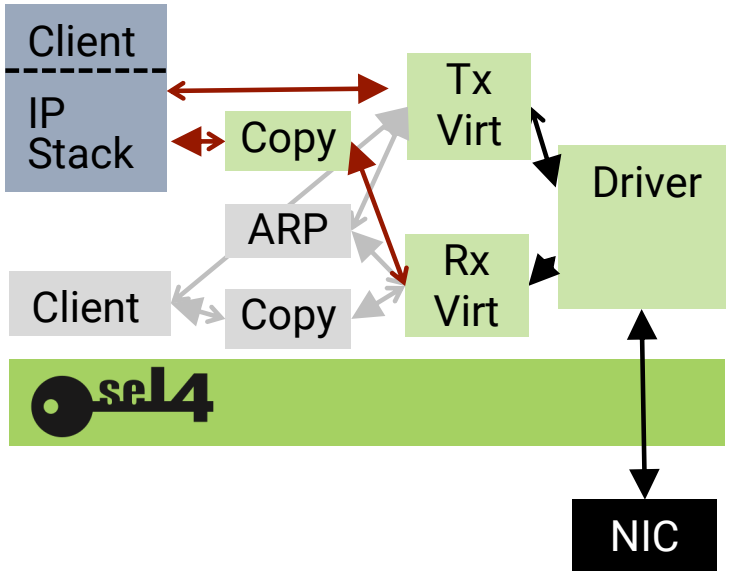
Must return free buffers!

© Gernot Heiser 2019–25 – CC BY 4.0



Comparing to Linux



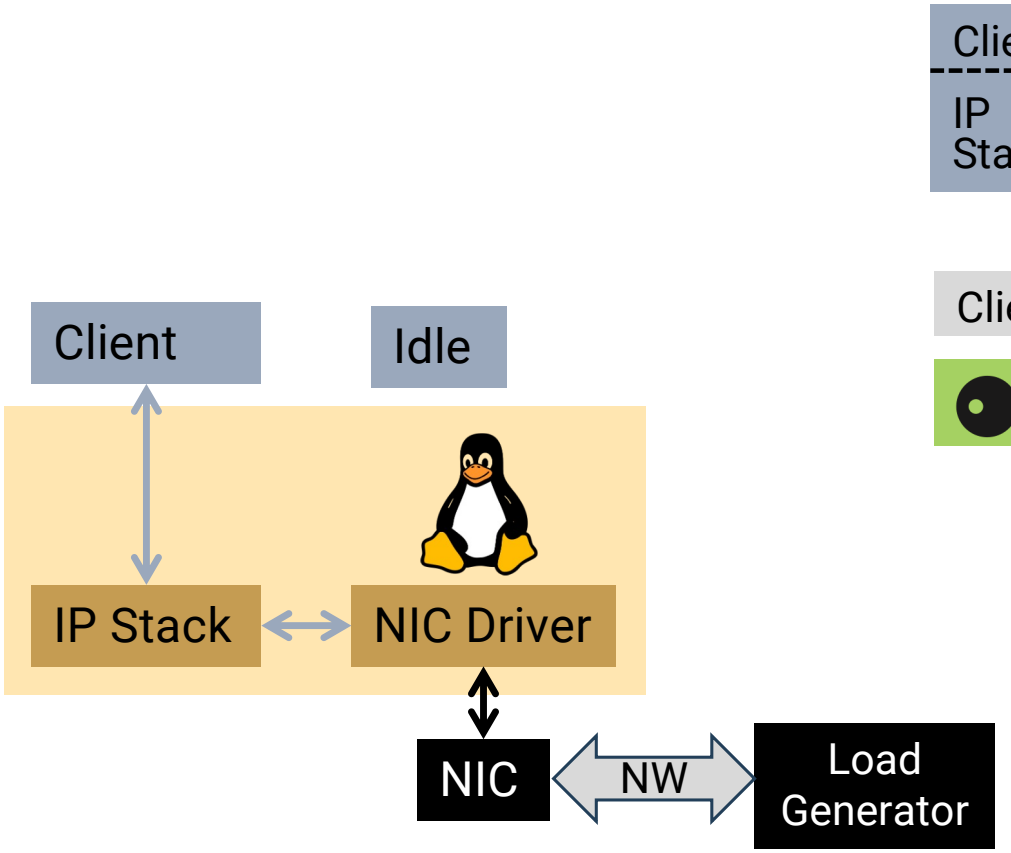


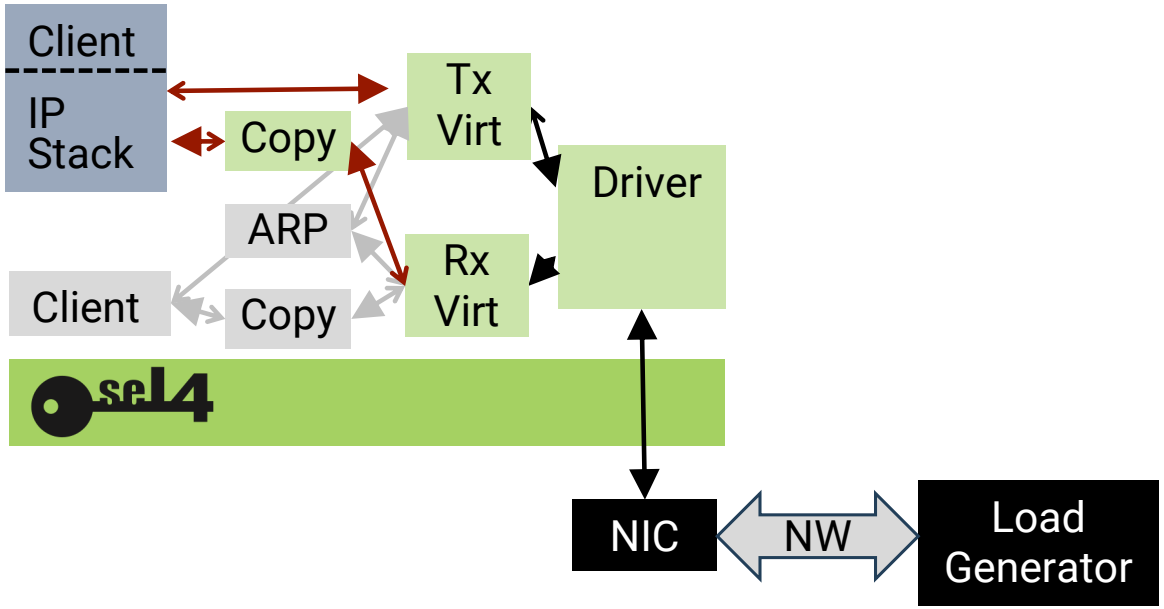
- **LionsOS:** 30 mode switches, 15 context switches
- **Linux:** 6 mode switches, 2 context switches

© Gernot Heiser 2019–25 – CC BY 4.0



Comparing Performance: Setup





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

© Gernot Heiser 2019–25 – CC BY 4.0

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:

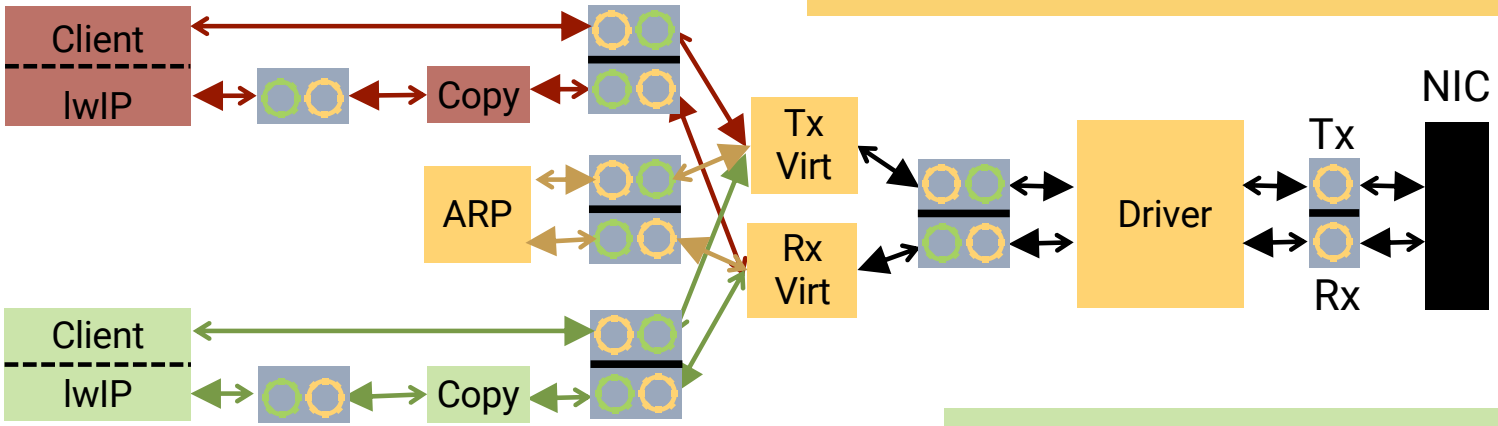
rel O/H = 0.5G cy/s / 3G cy/s = 17% of core

© Gernot Heiser 2019–25 – CC BY 4.0



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!
- Without forced batching (IRQ holdoff)!

© Gernot Heiser 2019–25 – CC BY 4.0

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

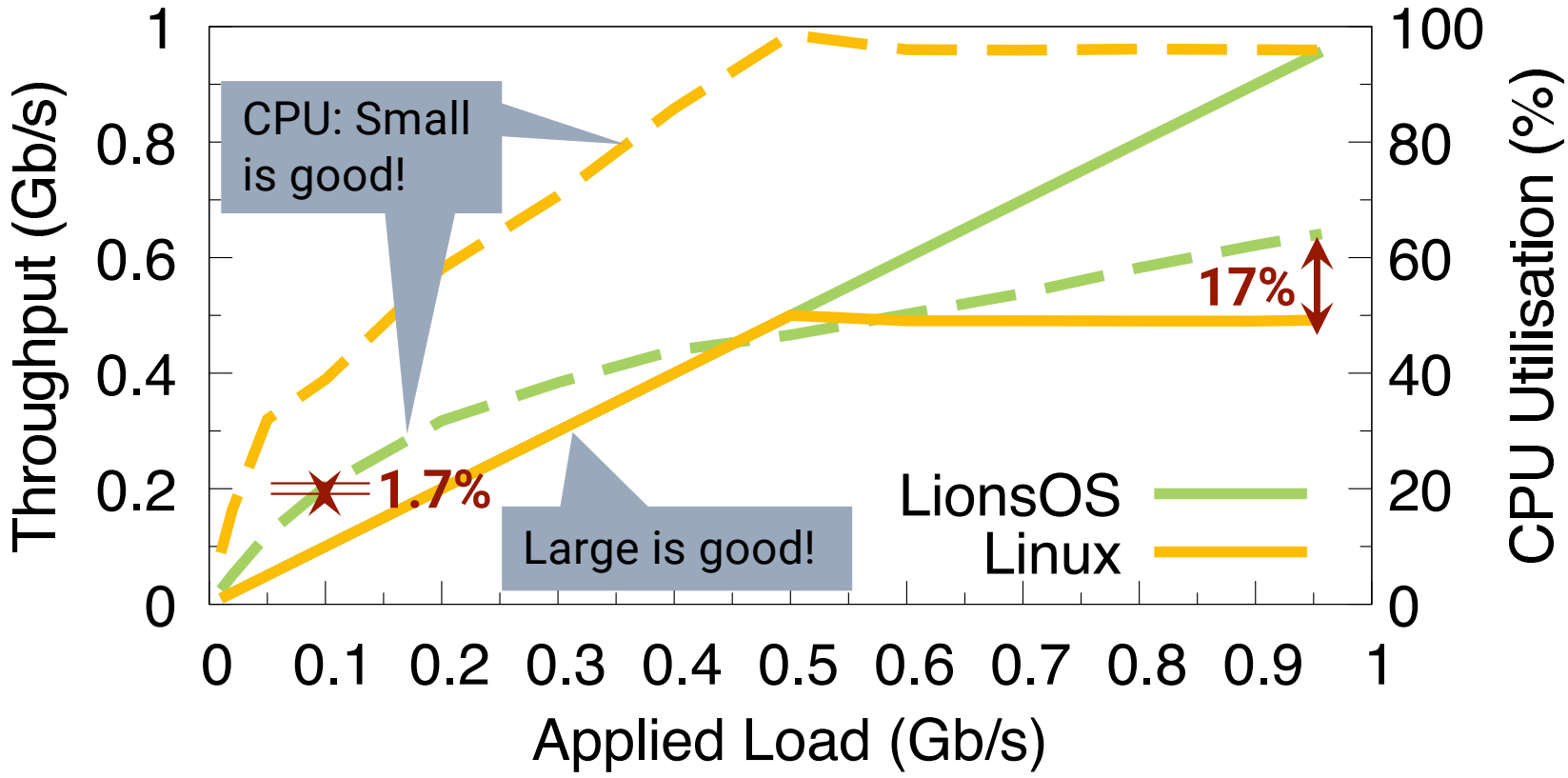
Highly pessimistic due to natural batching!

Avoid batching by spacing packets!

© Gernot Heiser 2019–25 – CC BY 4.0



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

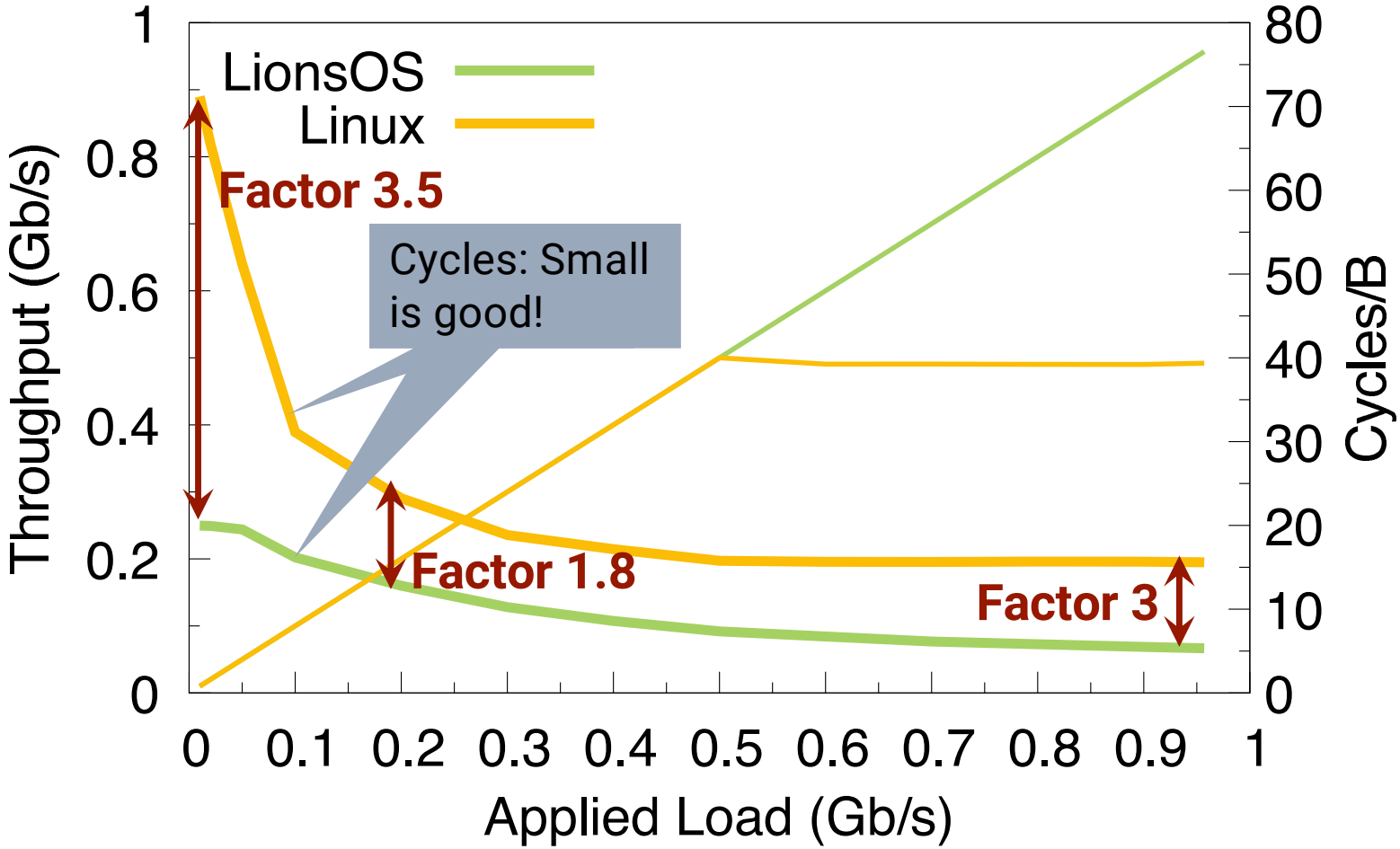


Single-core configuration

© Gernot Heiser 2019–25 – CC BY 4.0



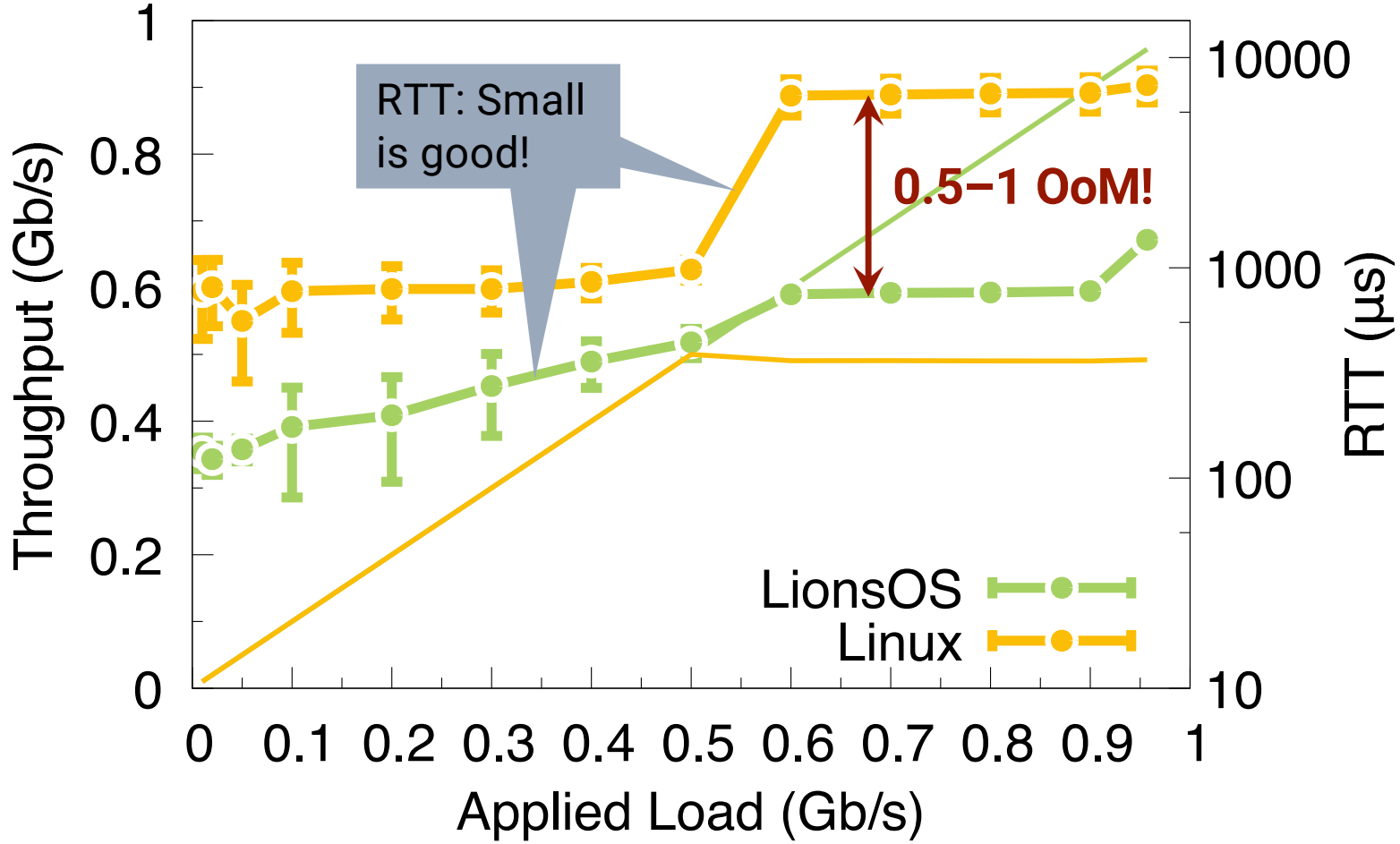
Performance: Processing Cost per Byte



© Gernot Heiser 2019–25 – CC BY 4.0



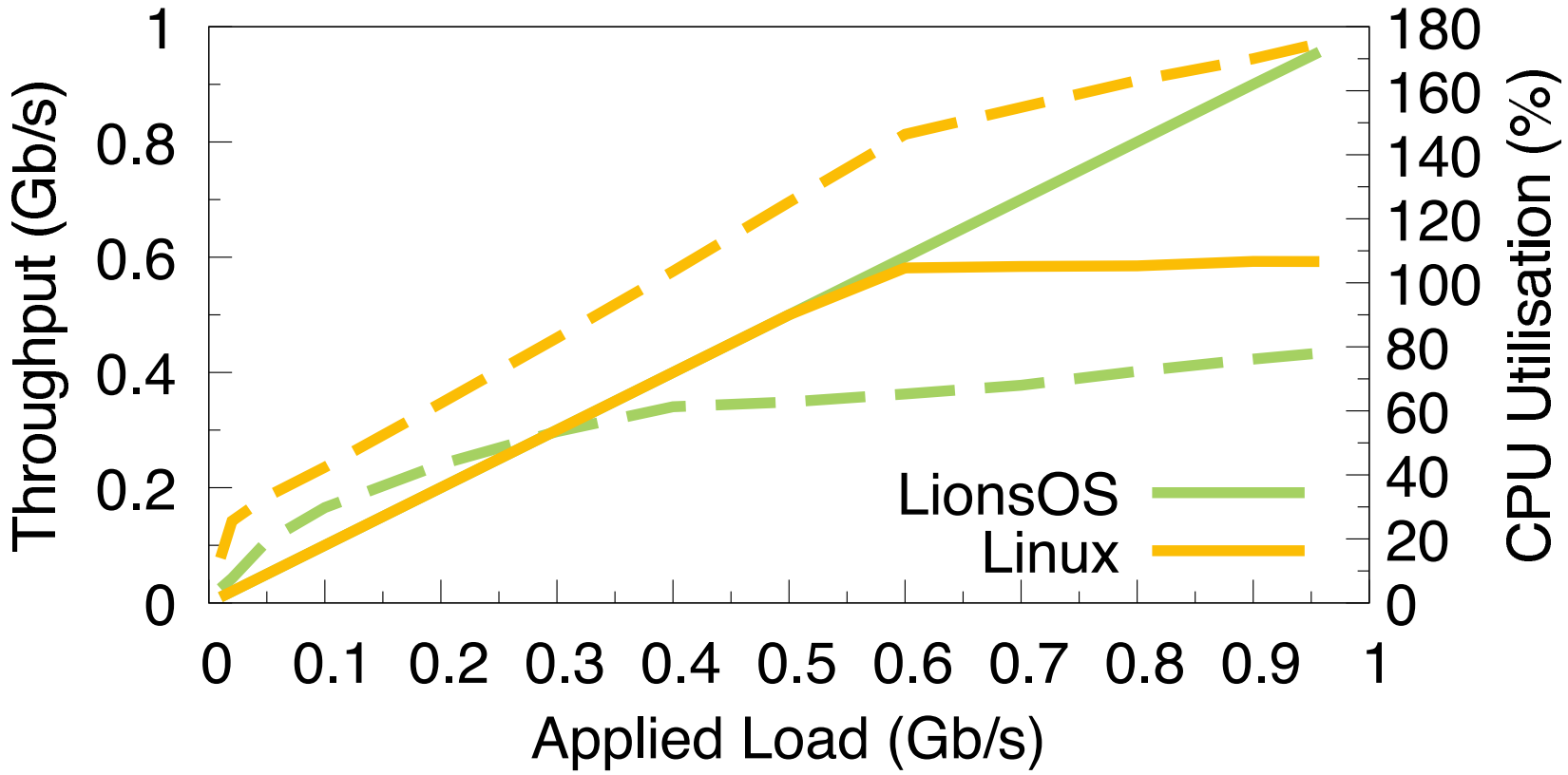
Performance: Round-Trip Times



© Gernot Heiser 2019–25 – CC BY 4.0



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

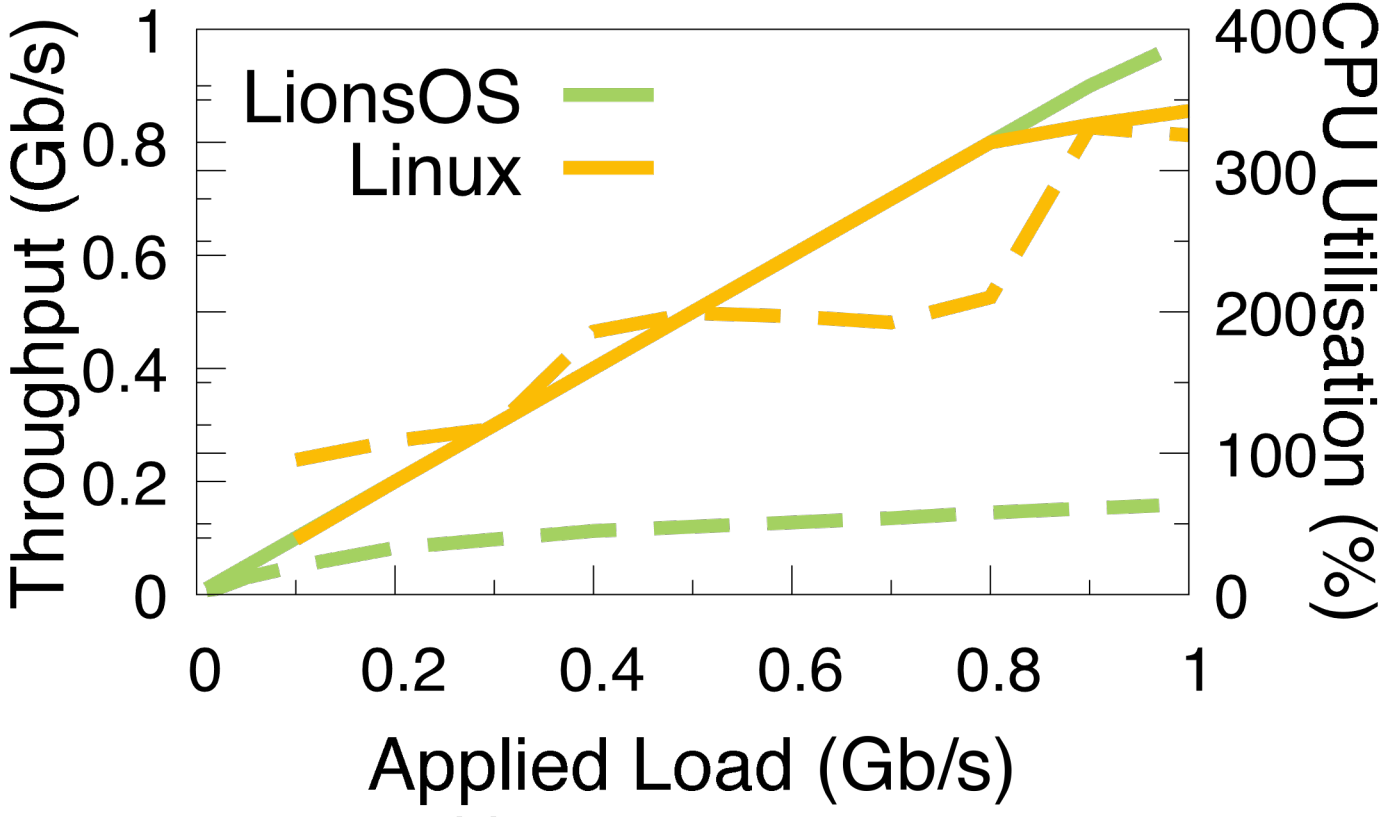


Multicore configuration

© Gernot Heiser 2019–25 – CC BY 4.0



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

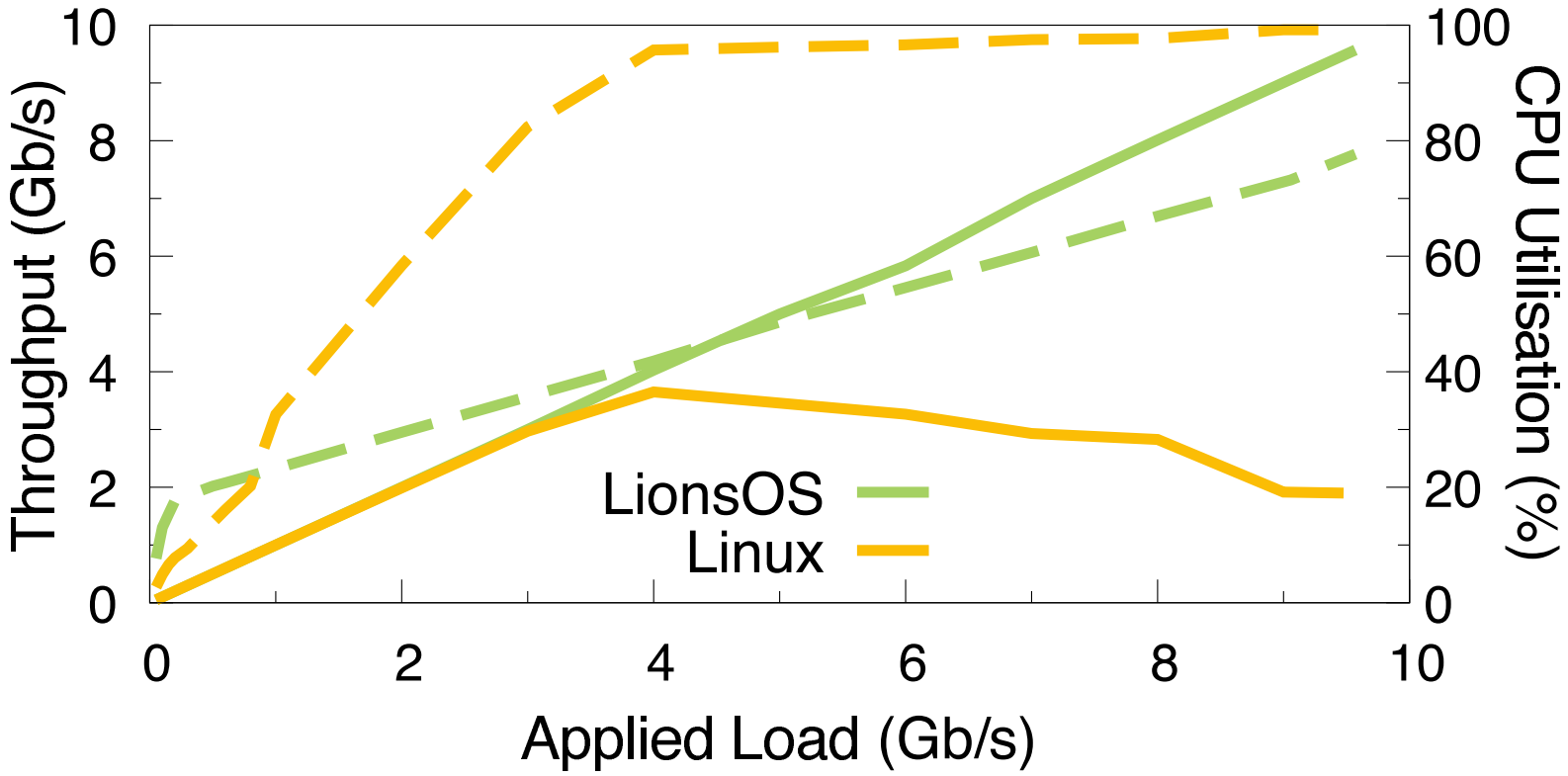


Multicore configuration, **Linux multiple clients**

© Gernot Heiser 2019–25 – CC BY 4.0



Performance: x86, 10Gb/s Eth, UDP

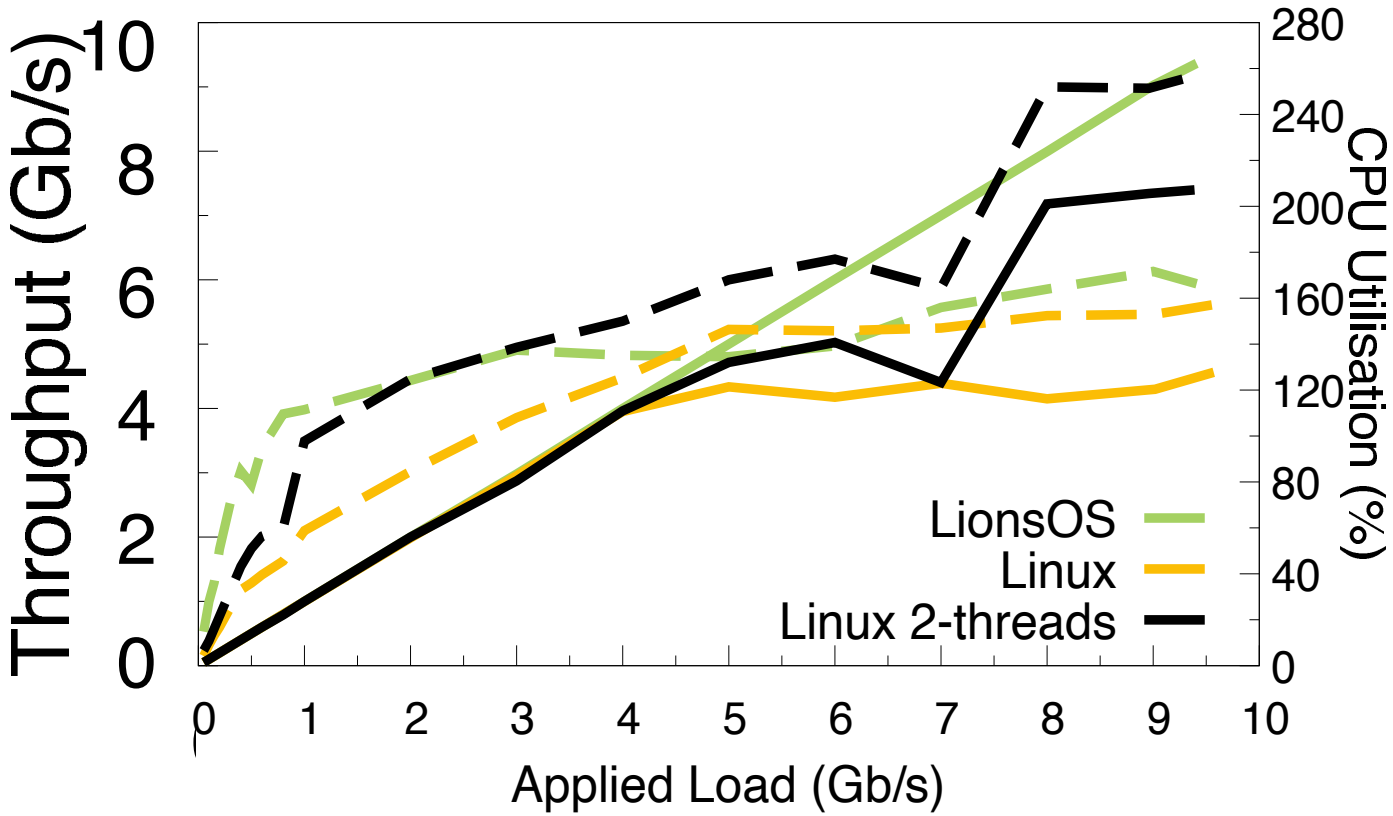


Single-core configuration

© Gernot Heiser 2019–25 – CC BY 4.0



Performance: x86, 10Gb/s Eth, UDP



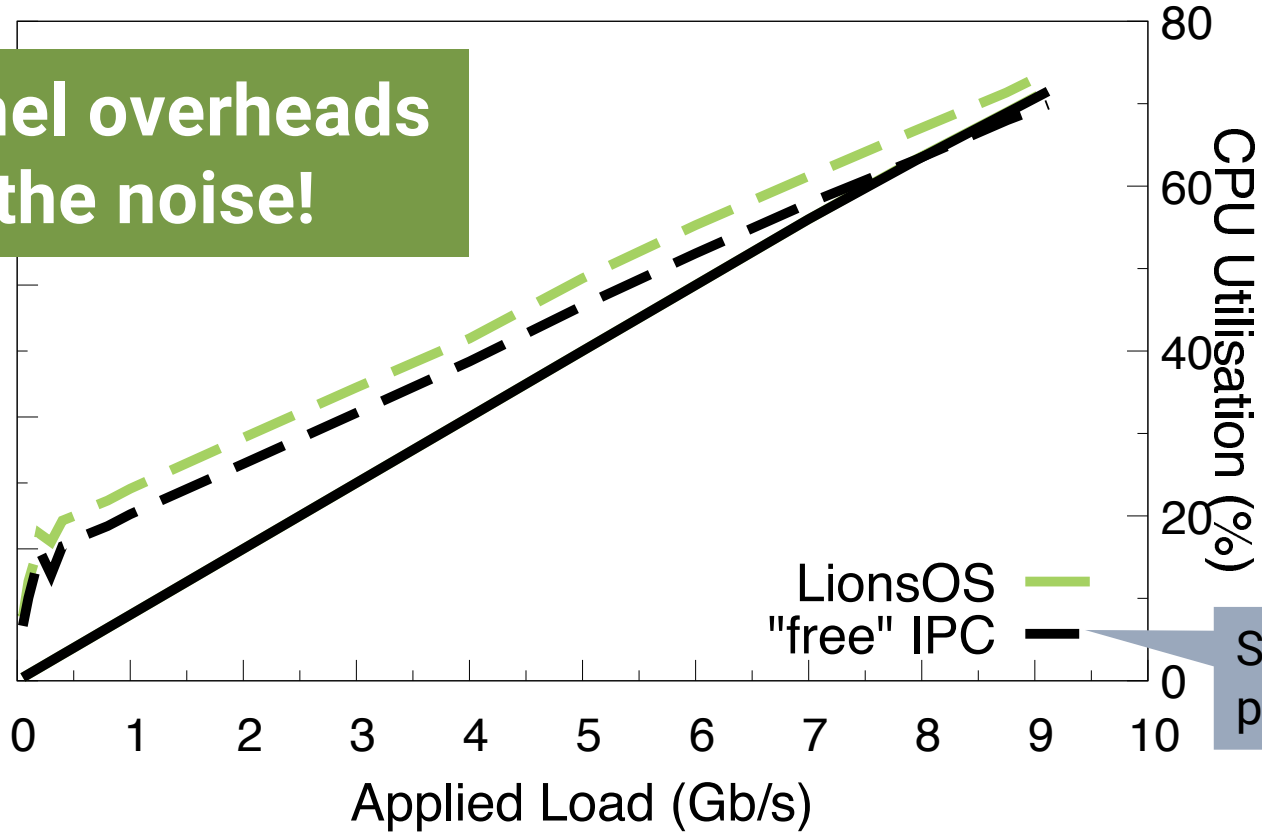
Multicore configuration

© Gernot Heiser 2019–25 – CC BY 4.0



Syscall cost simulations (x86)

Microkernel overheads are in the noise!



Subtract 500cy per kernel entry

Single-core configuration

© Gernot Heiser 2019–25 – CC BY 4.0



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

© Gernot Heiser 2019–25 – CC BY 4.0



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
- …

Radical Simplicity!

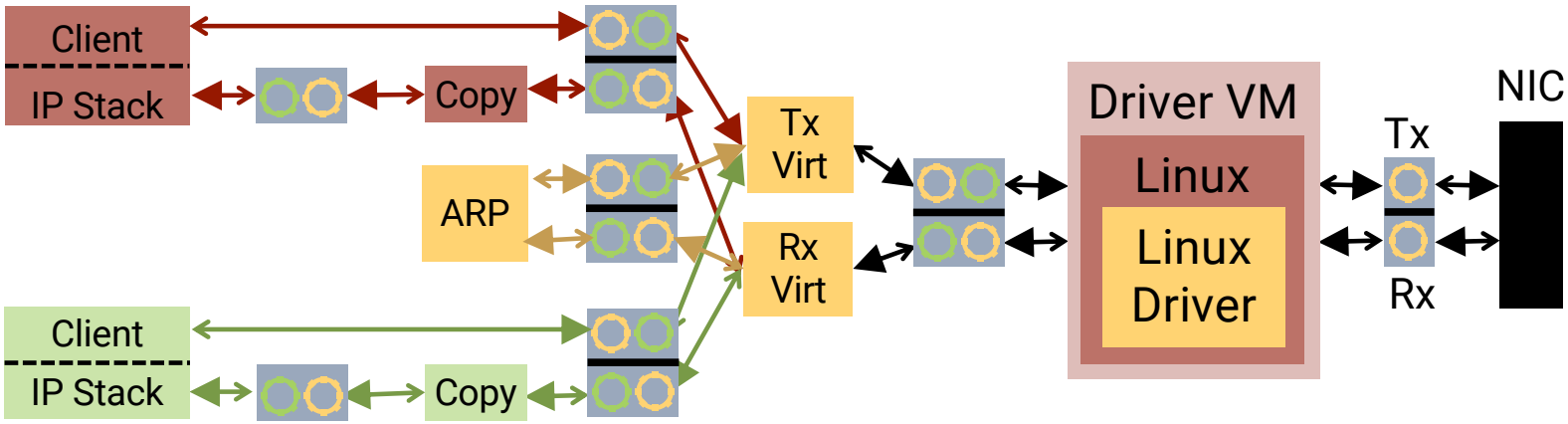
Static **architecture**, mostly static resource management

© Gernot Heiser 2019–25 – CC BY 4.0



LionsOS Driver VMs

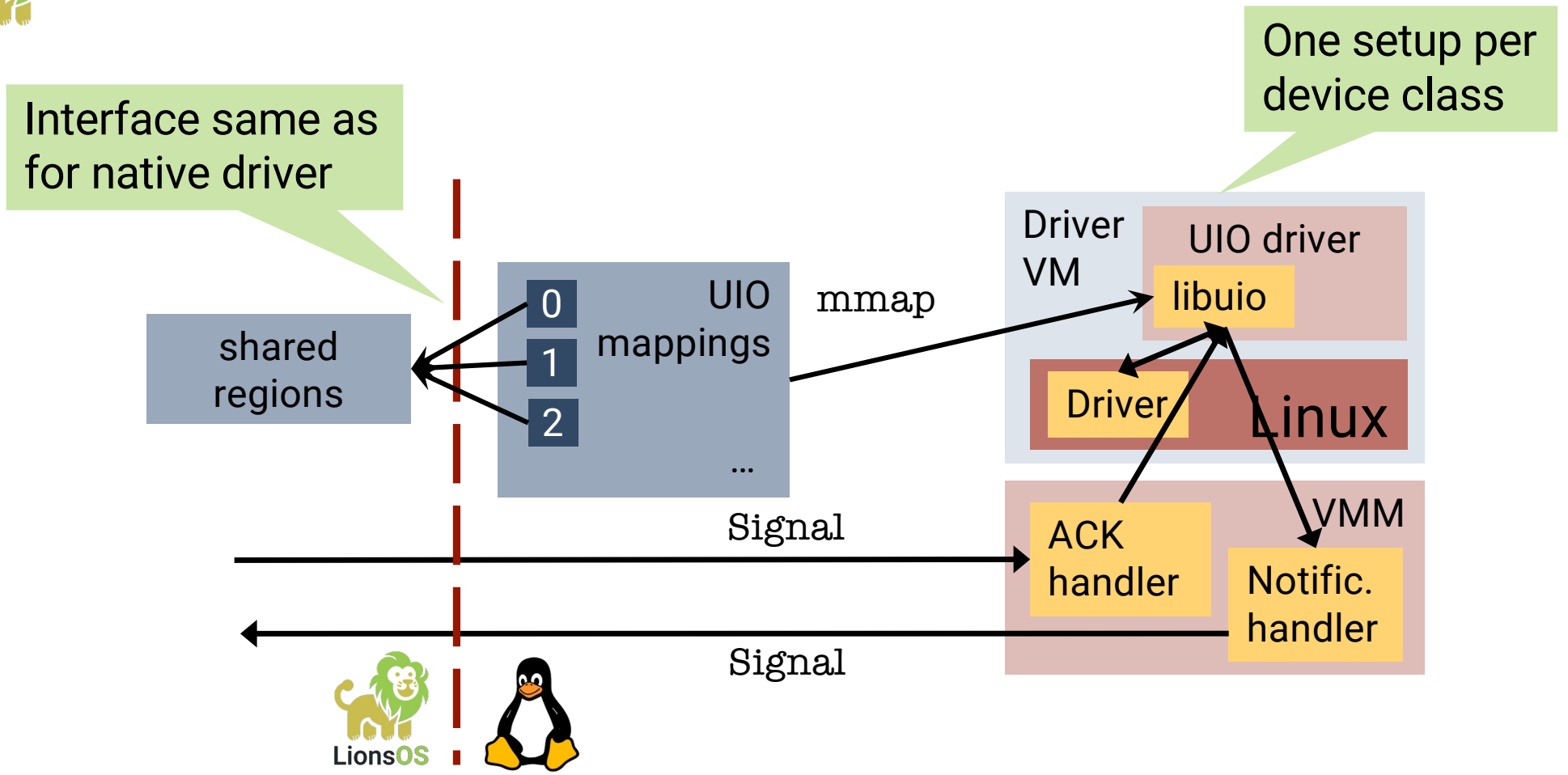
- Transparently use per-device driver VM instead of native driver
- Re-use unmodified Linux driver



Approach pioneered by LeVasseur&Uhlig, OSDI'04

© Gernot Heiser 2019–25 – CC BY 4.0

LionsOS Driver VMs



© Gernot Heiser 2019–25 – CC BY 4.0

Driver-VM Cost

In progress: using same setup to develop LionsOS modules under Linux

Driver	Kernel	RAM Disk	Runtime	Total
Default	29 MiB	6.7 MiB	70 MiB	106 MiB
Audio	3 MiB	2.4 MiB	18 MiB	23 MiB
Block	3 MiB	0.05 MiB	12 MiB	15 MiB

Optimised

Effort:
- Few days to set up UIO driver
- Total ≈ 2 weeks / device class

No performance data yet (but expect it to be bad)

© Gernot Heiser 2019–25 – CC BY 4.0

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, …
- Verified libmicrokit, Ethernet driver, Tx Virt
- Runs Linux, Windows in VMs



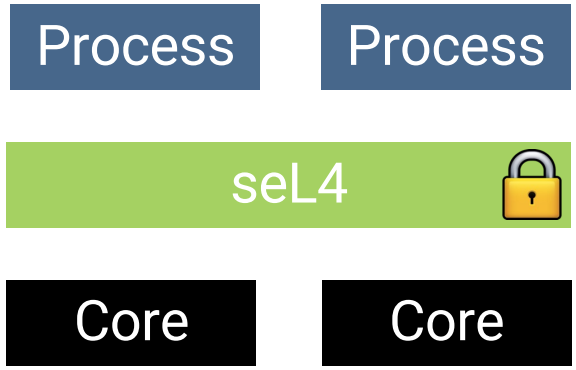
© Gernot Heiser 2019–25 – CC BY 4.0

seL4 on Multicore

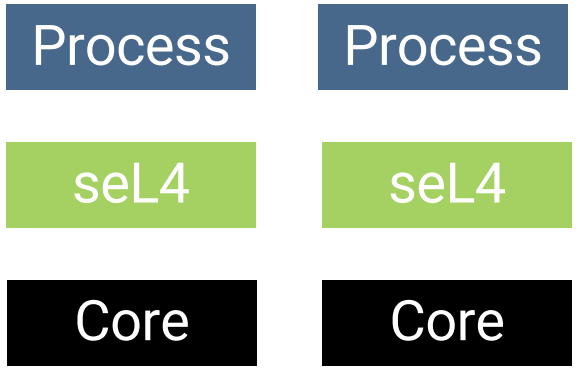
© Gernot Heiser 2019–25 – CC BY 4.0

SMP vs Multikernel

Symmetric multi-processing (SMP)



Multikernel



© Gernot Heiser 2019–25 – CC BY 4.0

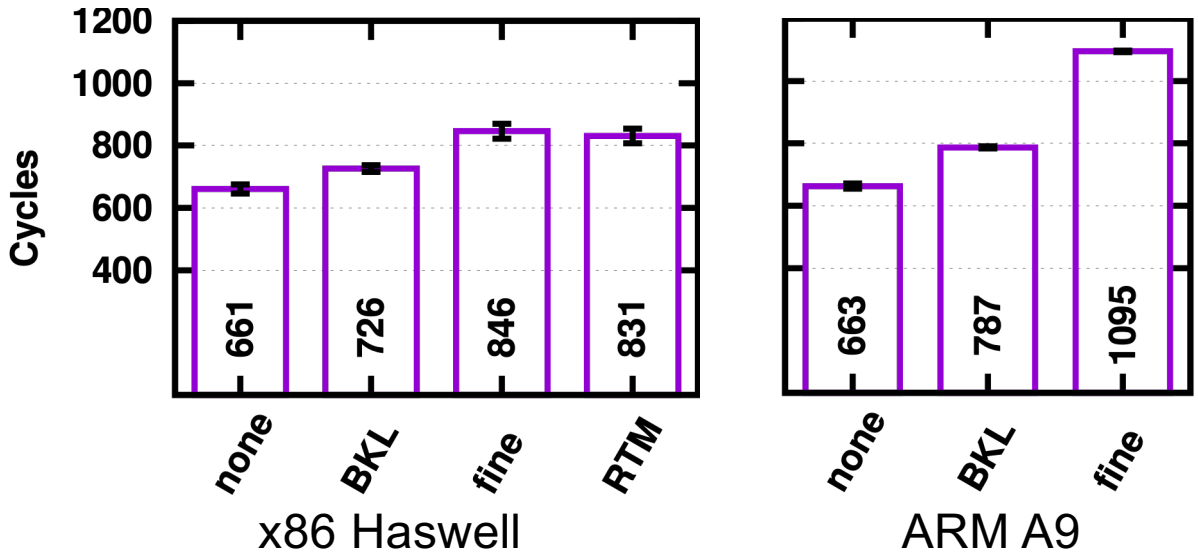
11 Years Ago: BKL vs FGL

For a Microkernel, a Big Lock Is Fine

APSys'15

Sean Peters, Adrian Danis, Kevin Elphinstone, Gernot Heiser

NICTA and UNSW Australia

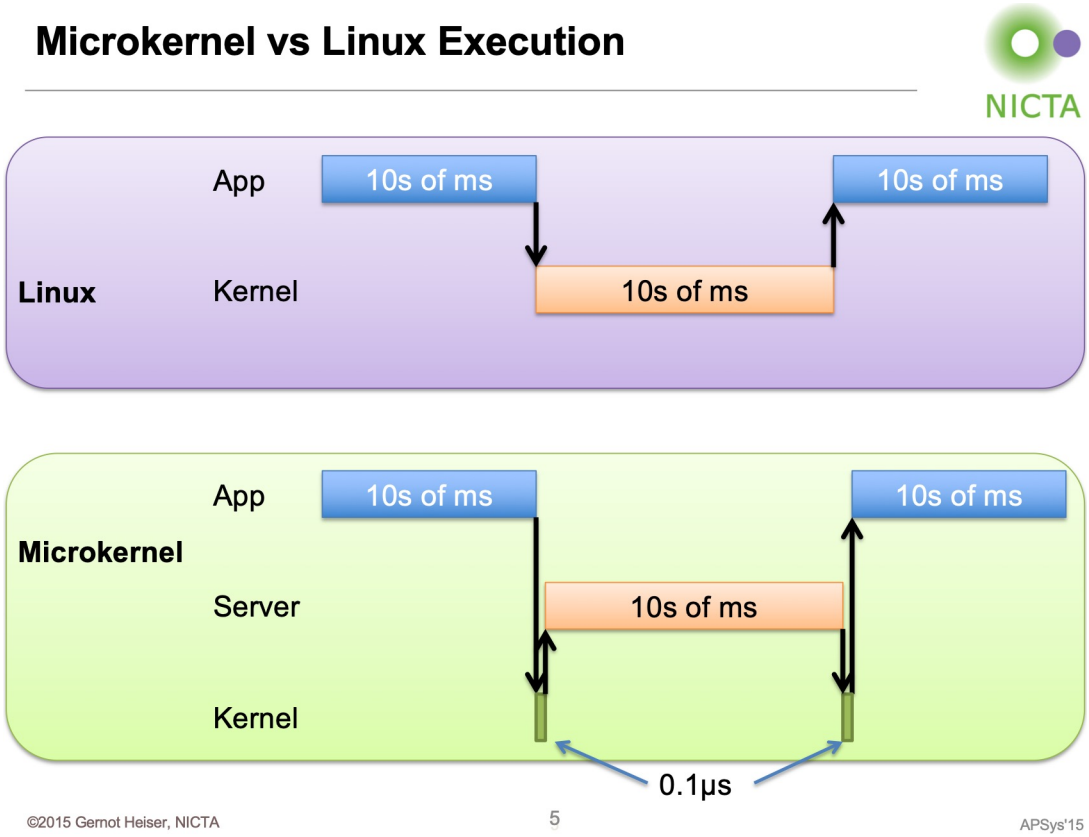


Findings:

- Locks are expensive
  - Big kernel lock (BKL) adds 10% to syscall
  - Fine-grained locking (FGL) adds 20–70%

© Gernot Heiser 2019–25 – CC BY 4.0

11 Years Ago: Observations



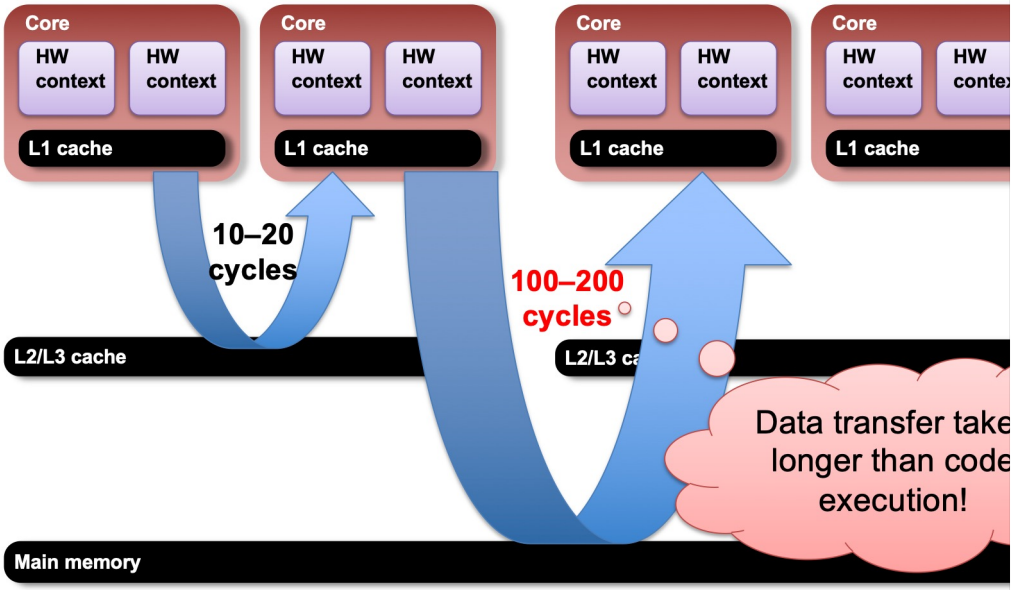
Microkernels are different!

- Syscalls are ≈ 0.1µs
- Linux syscalls ≈ 10s ms

© Gernot Heiser 2019–25 – CC BY 4.0

11 Years Ago: Assertions

Cache Line Migration Latencies



Microkernel Multicore Design

Assertion 1: Minimise locks, not locked code

- Amount of locked code is small anyway, 100–200 instructions
- Corresponds to medium-grained locks in Linux
- Cost of locks is within an OoM of kernel execution time
- Kernel times are short ⇒ contention is low

Assertion 2: Don't share mirokernel data without shared cache

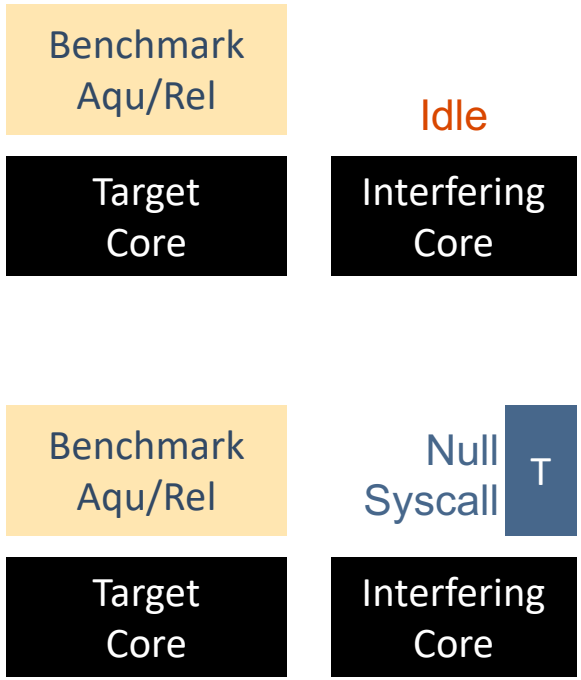
- Migrating only a few cache lines takes longer than rest of syscall

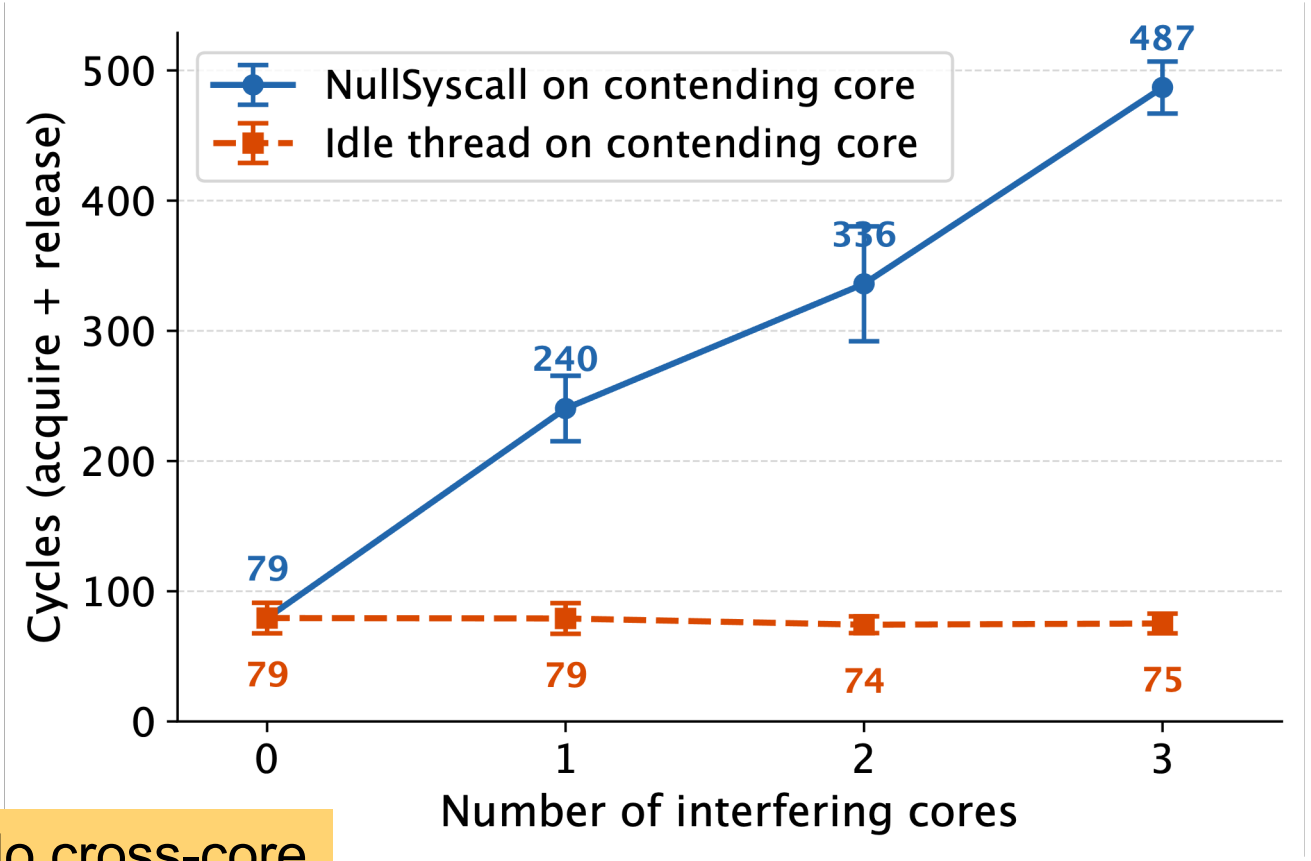
Assertion 3: Big lock will perform for closely-coupled cores

- Shared caches presently have moderate core counts
- Big lock in a *well-designed* microkernel will scale there

© Gernot Heiser 2019–25 – CC BY 4.0

Let's Revisit the BLK

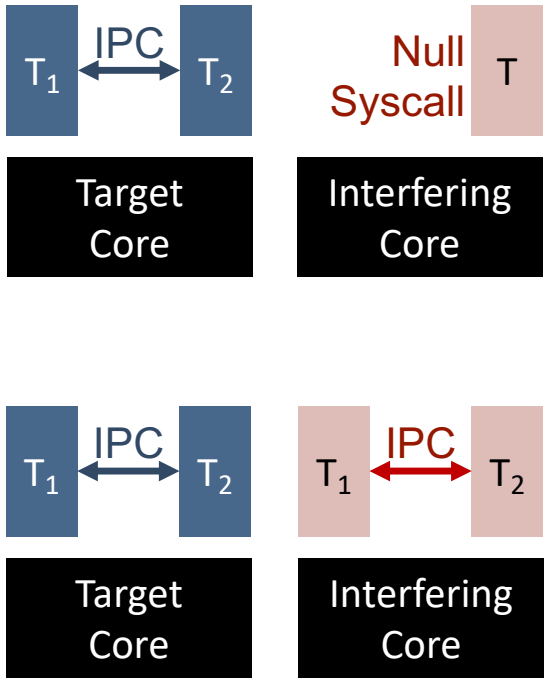


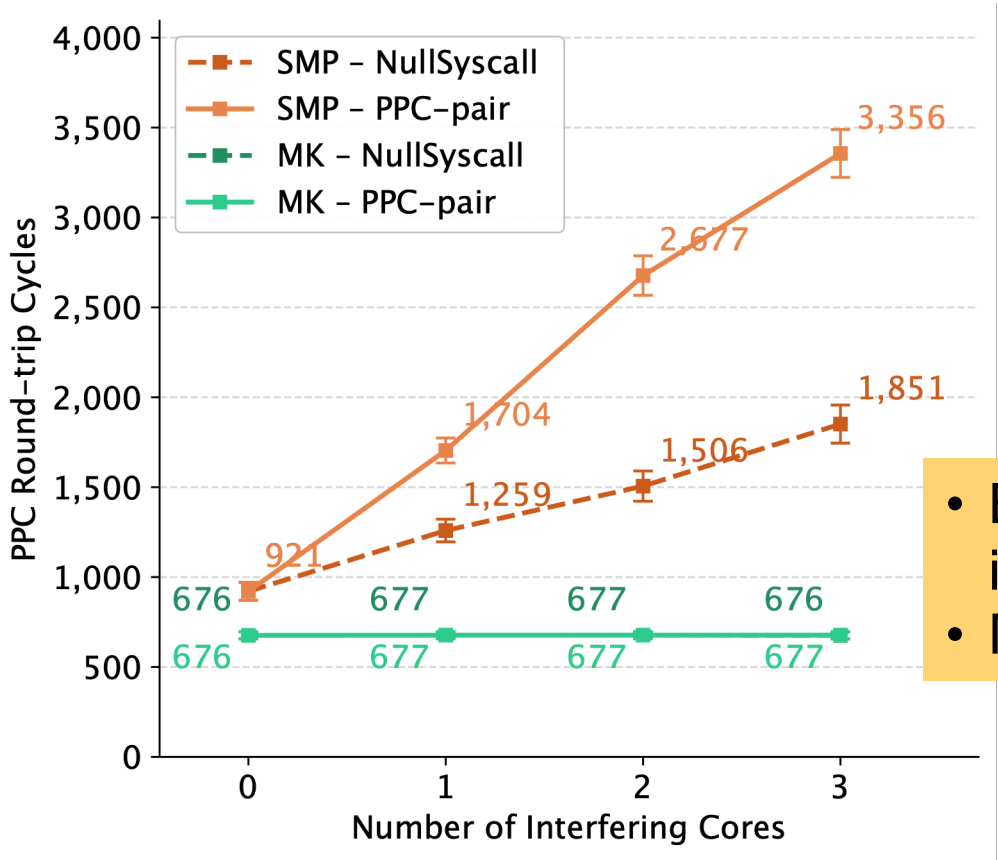


No cross-core operations!

© Gernot Heiser 2019–25 – CC BY 4.0

Ping-Pong Interference Benchmark





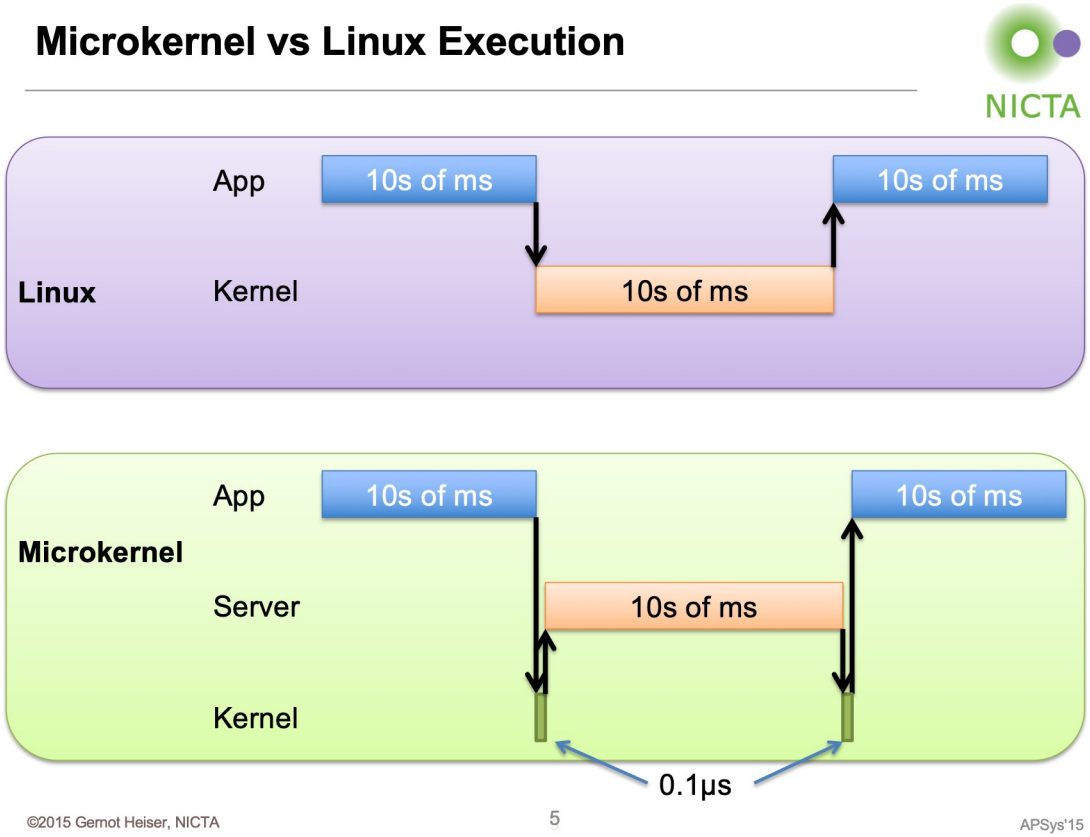
- BKL: strong interference
- Multikernel none

© Gernot Heiser 2019–25 – CC BY 4.0

Observations

- None of this is surprising
- APSys'15 argument:
  - Syscalls are short, with sufficient usermode execution will not interfere
  - SMP is the preferred choice for cores sharing an L2 cache

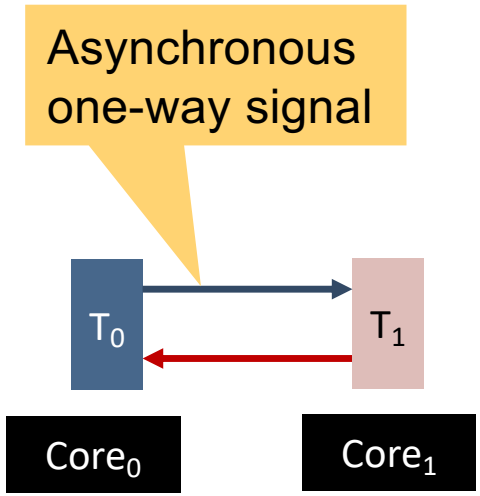
Really?



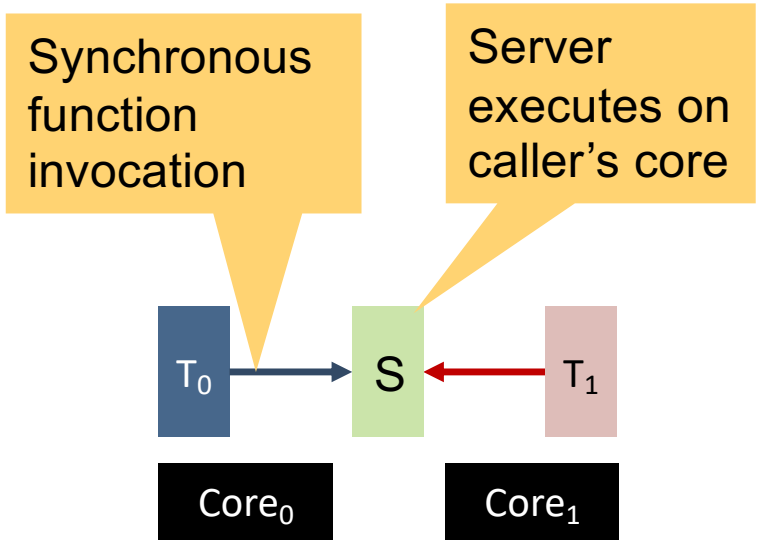
© Gernot Heiser 2019–25 – CC BY 4.0

How About Cross-Core Operations?

Two relevant cross-core operations:



Cross-core Notifications

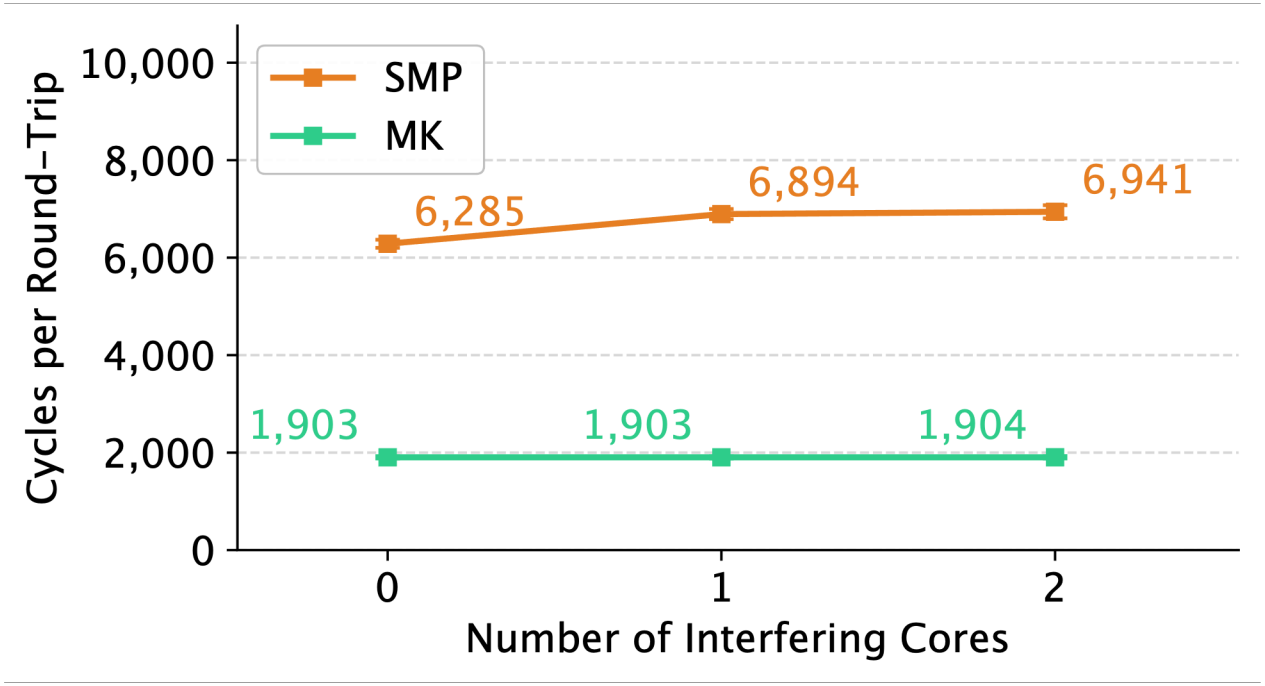


Shared passive servers

© Gernot Heiser 2019–25 – CC BY 4.0

Cross-Core Notifications

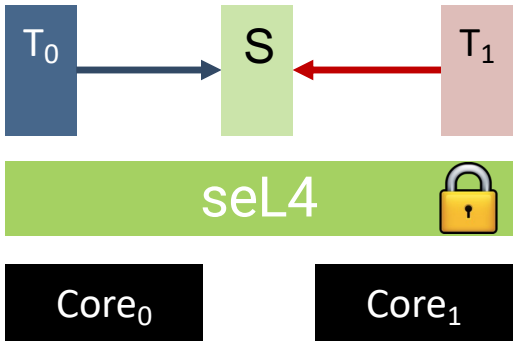
Two cores ping-pong Notifications



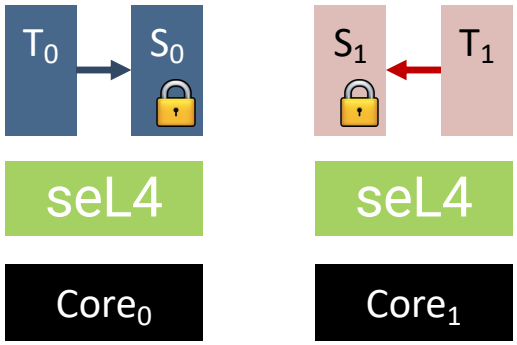
Doing everything in the kernel is **more** expensive than in userland!

© Gernot Heiser 2019–25 – CC BY 4.0

Passive Server on Multikernel



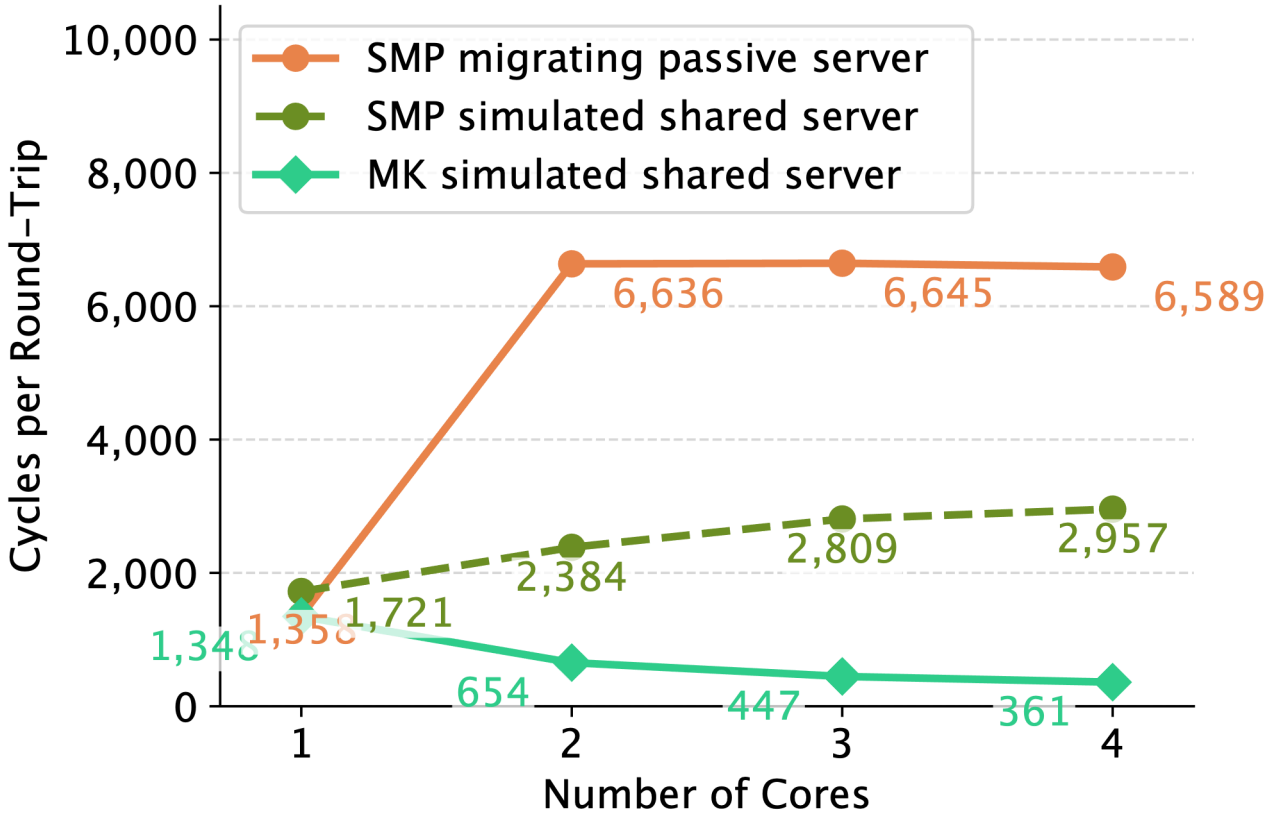
SMP



Multikernel

© Gernot Heiser 2019–25 – CC BY 4.0

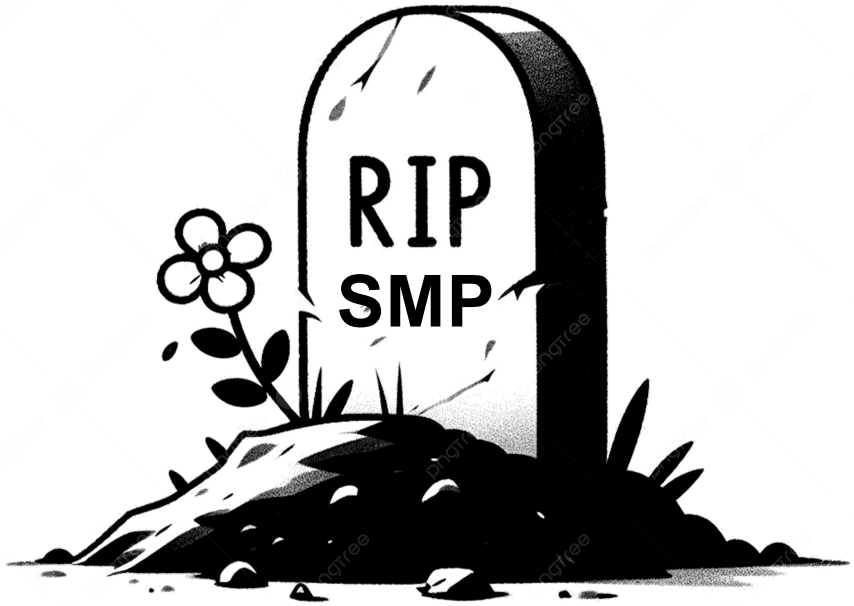
Passive-Server Performance



Cores compete for server
⇒ frequent migrations

Minimality principle:
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.
[Liedtke SOSP'95]

© Gernot Heiser 2019–25 – CC BY 4.0



© Gernot Heiser 2019–25 – CC BY 4.0

Reminders

- Please complete the myExperience Survey
- Exam preparation session: Wed 25/11 at 4pm in CIVL-G1
- Honours theses at Trustworthy Systems
  https://trustworthy.systems/students/theses
- **John Lions CS Honours Award for thesis in OS**
  https://www.scholarships.unsw.edu.au/scholarships/id/1757
  Deadline: 30 November for T1/27!

© Gernot Heiser 2019–25 – CC BY 4.0