Multiprocessor OS

part 2

COMP9242 – Advanced Operating Systems

Ihor Kuz (Kry10, UNSW) Ihor.kuz@unsw.edu.au

2023 T3 Week 10

Overview

Multiprocessor OS (Background and Review)

- How does it work? (Background)
- Scalability (Review)

Multiprocessor Hardware

- Contemporary systems (Intel, AMD, ARM, Oracle/Sun)
- Experimental (Intel, MS, Polaris)

OS Design for Multiprocessors

- Guidelines
- Design approaches
  - Divide and Conquer (Disco, Tesselation)
  - Reduce Sharing (K42, Corey, Linux, FlexSC, scalable commutativity)
  - No Sharing (Barrelfish, fos)
  - Deal with Heterogeneity (de facto OS)

Summary

Scalability

- 100+ cores
- Amdahl's law really kicks in

Heterogeneity

- Heterogeneous cores, memory, etc.
- Properties of similar systems may vary wildly (e.g. interconnect topology and latencies between different AMD platforms)

NUMA

- Also variable latencies due to topology and cache coherence

Cache coherence may not be possible

- Can't use it for locking
- Shared data structures require explicit work

Computer is a distributed system

- Message passing
- Consistency and Synchronisation
- Fault tolerance

OS DESIGN for Multiprocessors

Optimisation for Scalability

Reduce amount of code in critical sections

- Increases concurrency
- Fine grained locking
  - Lock data not code (big kernel lock vs fine-grained locking)
  - Tradeoff: more concurrency but more locking (and locking causes serialisation)
- Lock free data structures

Avoid expensive memory access

- Avoid uncached memory
- Access cheap (close) memory

Optimisation for Scalability

Reduce false sharing

- Pad data structures to cache lines

Reduce cache line bouncing

- Reduce sharing
- E.g: MCS locks use local data

Reduce cache misses

- Affinity scheduling: run process on the core where it last ran.
- Avoid cache pollution
  - Don’t evict all application cache when OS runs
  - Don’t evict all OS cache when app runs

OS Design Guidelines for Modern (and future) Multiprocessors

Avoid shared data

- Performance issues arise less from lock contention than from data locality

Explicit communication

- Regain control over communication costs (and predictability)
  - Cache coherence is expensive, and opaque
- Sometimes it's the only option

Tradeoff: parallelism vs synchronisation

- Synchronisation introduces serialisation
- Make concurrent threads independent: reduce critical sections & cache misses
- Aim for: embarrassingly parallel

Allocate for locality

- E.g. provide memory local to a core

Schedule for locality

- With cached data
- With local memory

Tradeoff: uniprocessor performance vs scalability

Design approaches

Divide and conquer

- Divide multiprocessor into smaller bits, use them as normal
- Using virtualisation
- Using exokernel

Reduced sharing

- Brute force & Heroic Effort
  - Find problems in existing OS and fix them
  - E.g Linux rearchitecting: BKL -> fine grained locking
- By design
  - Avoid shared data as much as possible

No sharing

- Computer is a distributed system
  - Do extra work to share!

Dealing with heterogeneity

Divide and Conquer

Disco

- Scalability is too hard!

Context:

- ca. 1995, large ccNUMA multiprocessors appearing
- Scaling OSes requires extensive modifications

Idea:

- Implement a scalable VMM
- Run multiple OS instances

VMM has most of the features of a scalable OS:

- NUMA aware allocator
- Page replication, remapping, etc.

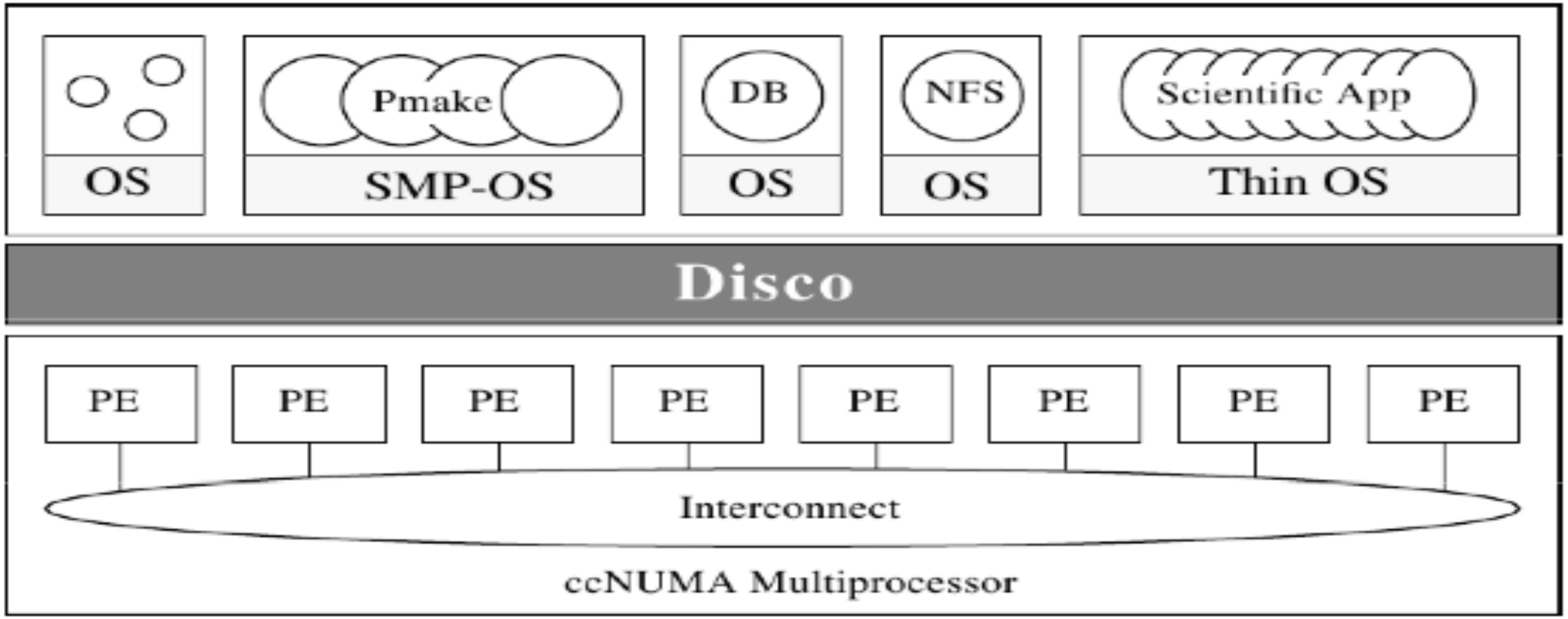
VMM substantially simpler/cheaper to implement

Modern incarnations of this

- Virtual servers (Amazon, etc.)
- Research (Cerberus)

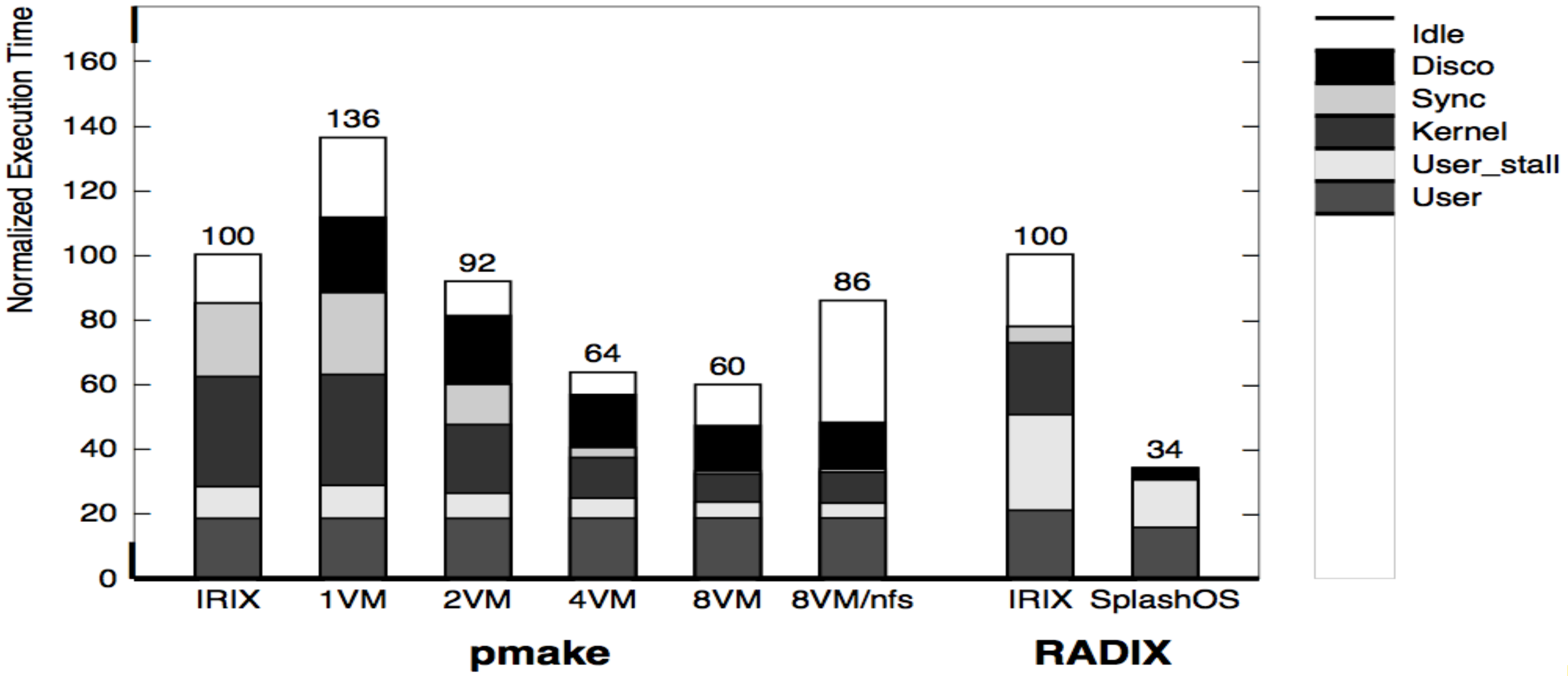
Running commodity OSes on scalable multiprocessors [Bugnion et al., 1997]
http://www-flash.stanford.edu/Disco/

Disco Architecture



[Bugnion et al., 1997]

Disco Performance



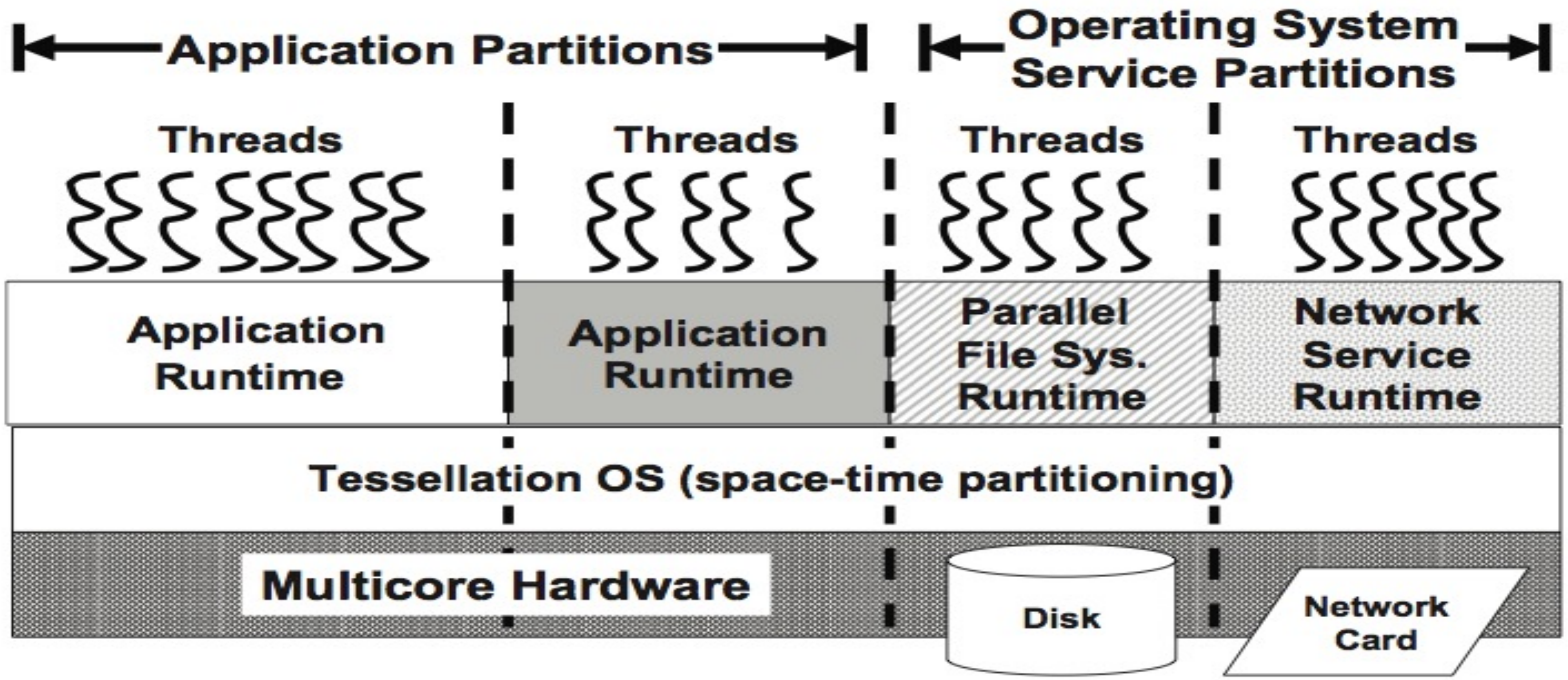
Space-Time Partitioning

Tessellation

- Space-Time partitioning
- 2-level scheduling

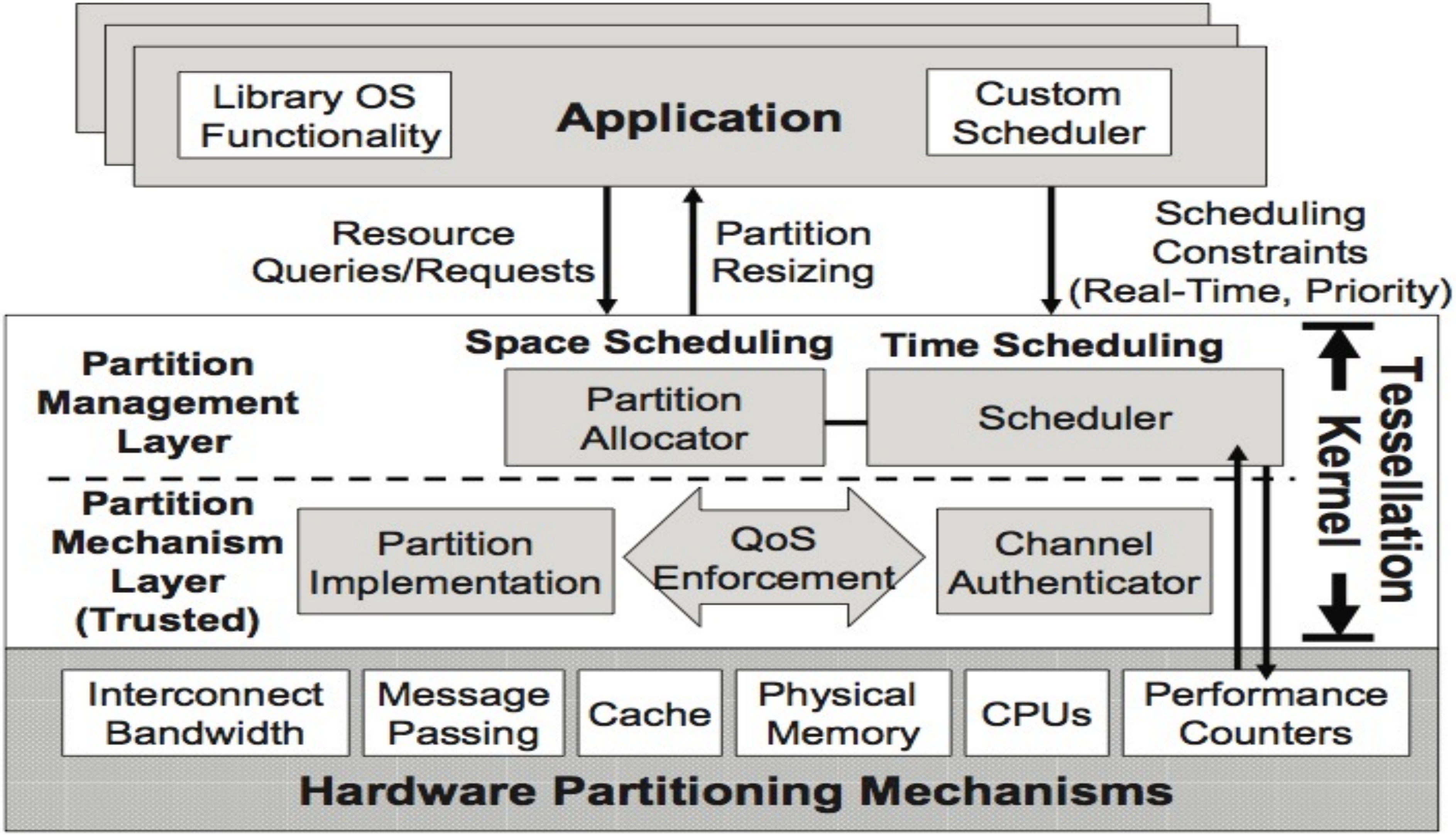
Context:

- 2009-… highly parallel multicore systems
- Berkeley Par Lab



Tessellation: Space-Time Partitioning in a Manycore Client OS [Liu et al., 2010]
http://tessellation.cs.berkeley.edu/

Tessellation



Reduce Sharing

K42

Context:
- 1997-2006: OS for ccNUMA systems
- IBM, U Toronto (Tornado, Hurricane)

Goals:
- High locality
- Scalability

Object Oriented
- Fine grained objects

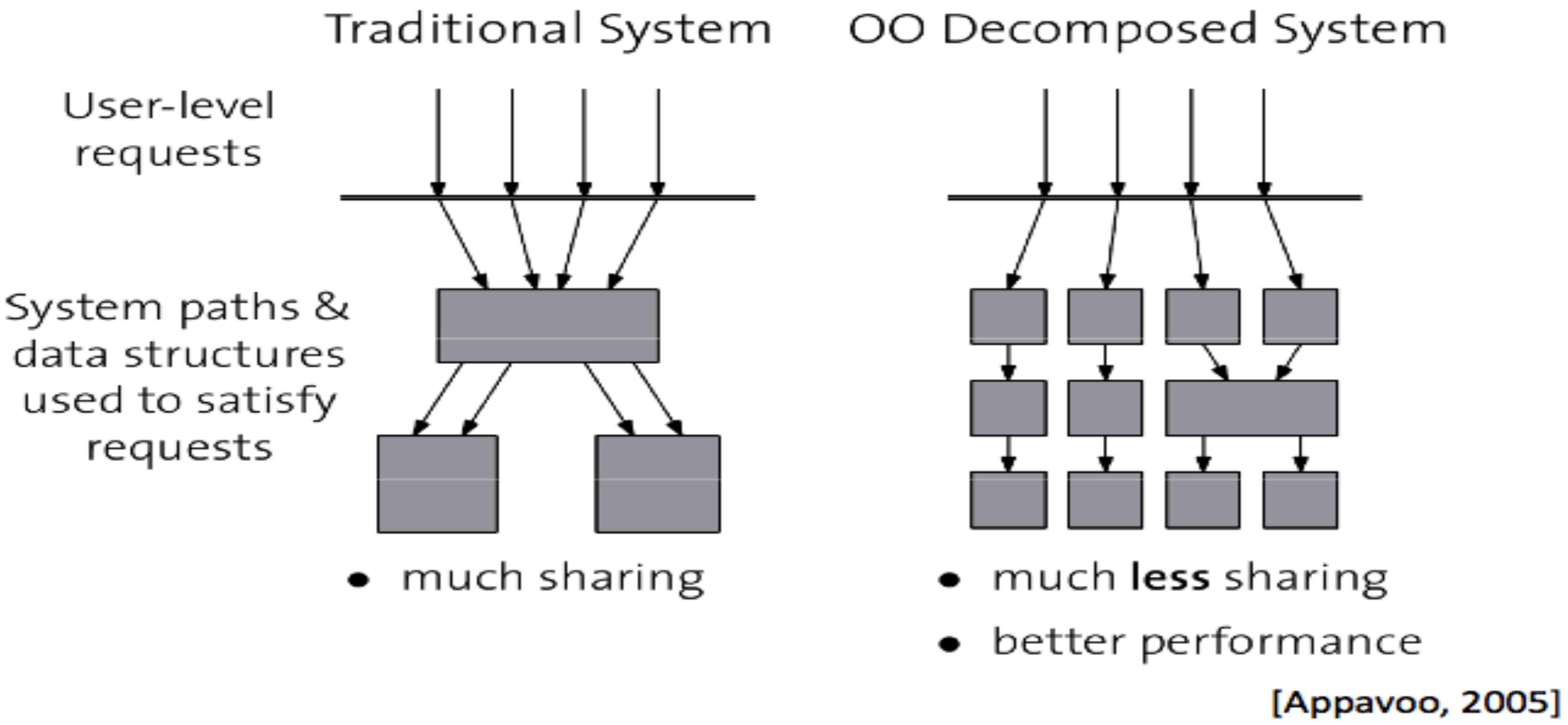
Clustered (Distributed) Objects
- Data locality

Deferred deletion (RCU)
- Avoid locking

NUMA aware memory allocator
- Memory locality

Clustered Objects, Ph.D. thesis [Appavoo, 2005]
http://www.research.ibm.com/K42/

K42: Fine-grained objects



K42: Clustered objects

Globally valid object reference

Resolves to

- Processor local representative

Sharing, locking strategy local to e

Transparency

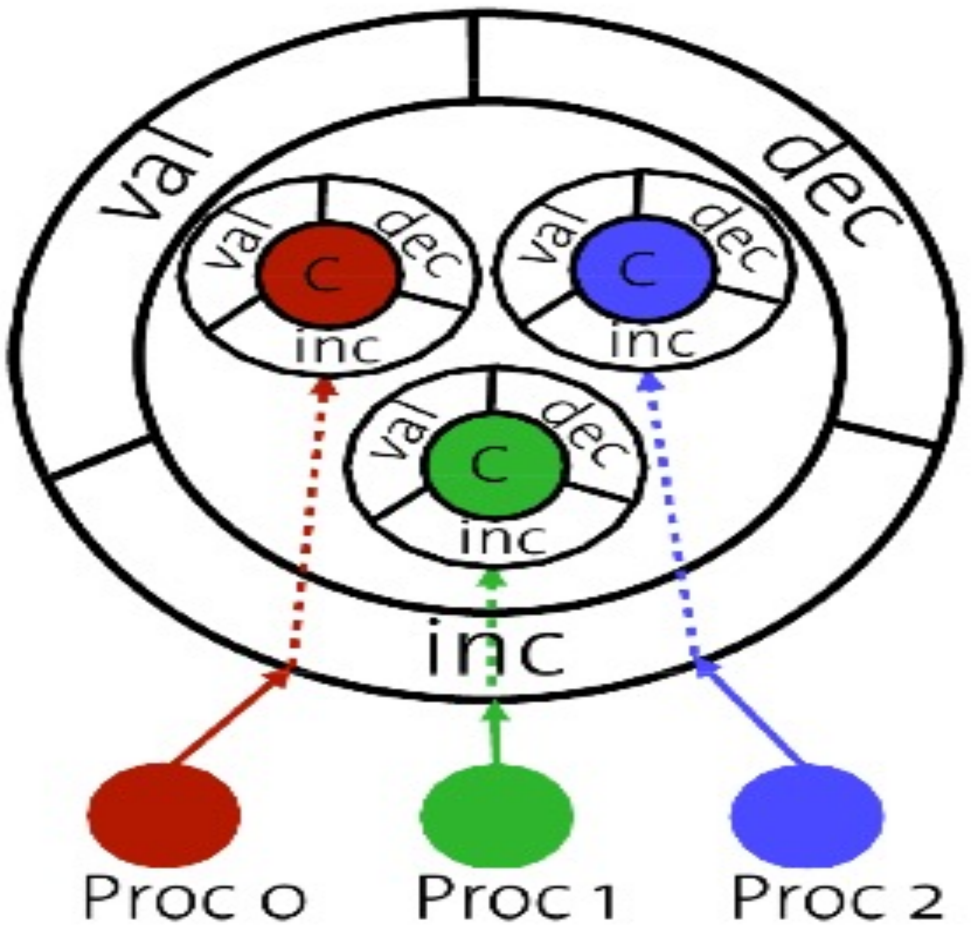
- Eases complexity
- Controlled introduction of locality

Shared counter:

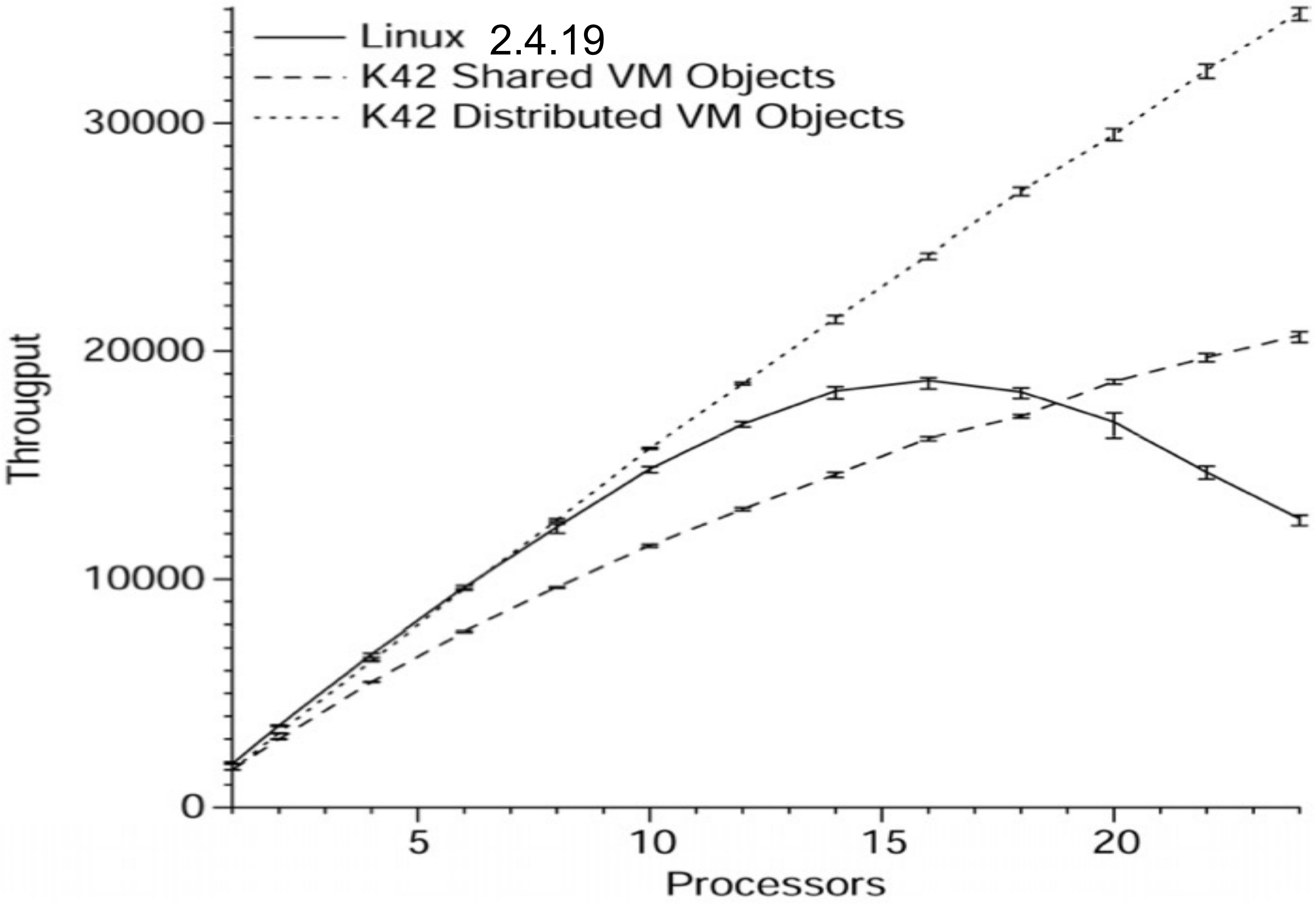
- *inc*, *dec*: local access
- *val*: communication

Fast path:

- Access mostly local structures



K42 Performance



Corey

Context

- 2008, high-end multicore servers, MIT

Goals:

- Application control of OS sharing

OS

- Exokernel-like, higher-level services as libraries
- By default only single core access to OS data structures
- Calls to control how data structures are shared

Address Ranges

- Control private per core and shared address spaces

Kernel Cores

- Dedicate cores to run specific kernel functions

Shares

- Lookup tables for kernel objects allow control over which object identifiers are visible to other cores.

Corey: An Operating System for Many Cores [Boyd-Wickizer et al., 2008]
http://pdos.csail.mit.edu/corey

Linux Brute Force Scalability

Context

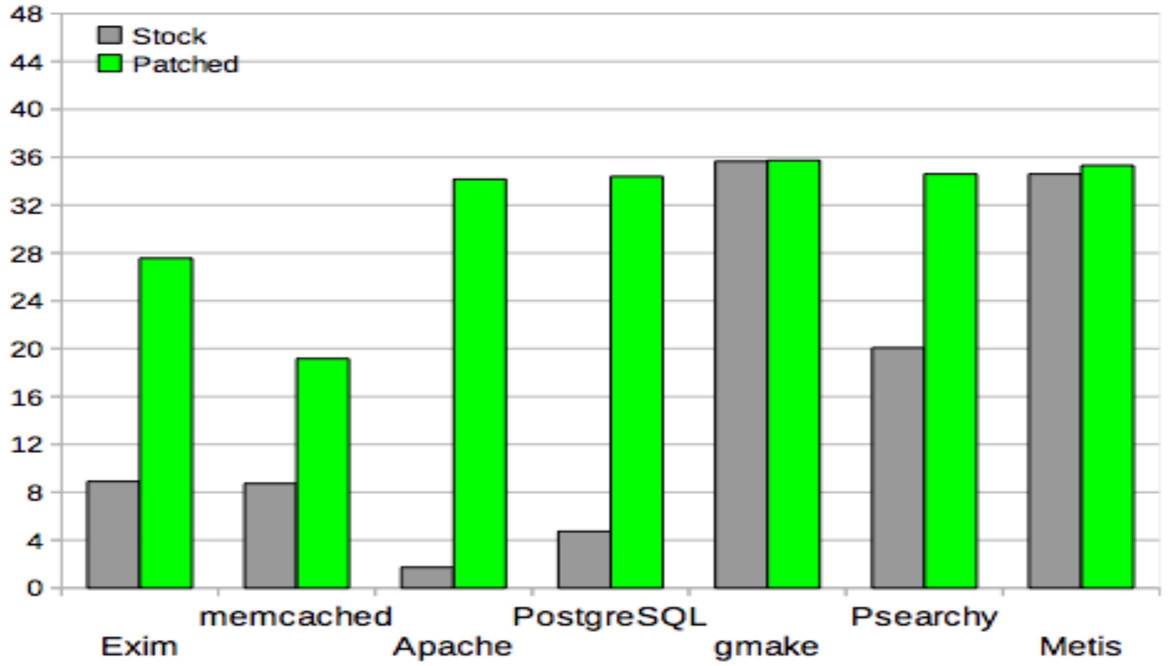
- 2010, high-end multicore servers, MIT

Goals:

- Scaling commodity OS

Linux scalability

- 2010 – scale Linux to 48 cores)



Y-axis: (throughput with 48 cores) / (throughput with one core)

An Analysis of Linux Scalability to Many Cores [Boyd-Wickizer et al., 2010]

Linux Brute Force Scalability

Apply lessons from parallel computing and past research

- sloppy counters,
- per-core data structs,
- fine-grained lock, lock free,
- cache lines
- 3002 lines of code changed

	memcached	Apache	Exim	PostgreSQL	gmake	Psearchy	Metis
Mount tables		X	X				
Open file table		X	X				
Sloppy counters	X	X	X				
inode allocation	X	X					
Lock-free dentry lookup		X	X				
Super pages							X
DMA buffer allocation	X	X					
Network stack false sharing	X	X		X			
Parallel accept		X					
Application modifications				X		X	X

Conclusion:

- no scalability reason to give up on traditional operating system organizations just yet.

Scalability of the API

Context

- 2013, previous multicore projects at MIT

Goals

- How to know if a system is really scalable?

Workload-based evaluation

- Run workload, plot scalability, fix problems
- Did we miss any non-scalable workload?
- Did we find all bottlenecks?

Is there something fundamental that makes a system non-scalable?

- The interface might be a fundamental bottleneck

The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors [Clements et al., 2013]

Scalable Commutativity Rule

The Rule

- *Whenever interface operations commute, they can be implemented in a way that scales*.

Commutative operations:

- Cannot distinguish order of operations from results
- Example:
  - Creat:
    - Requires that lowest available FD be returned
    - Not commutative: can tell which one was run first

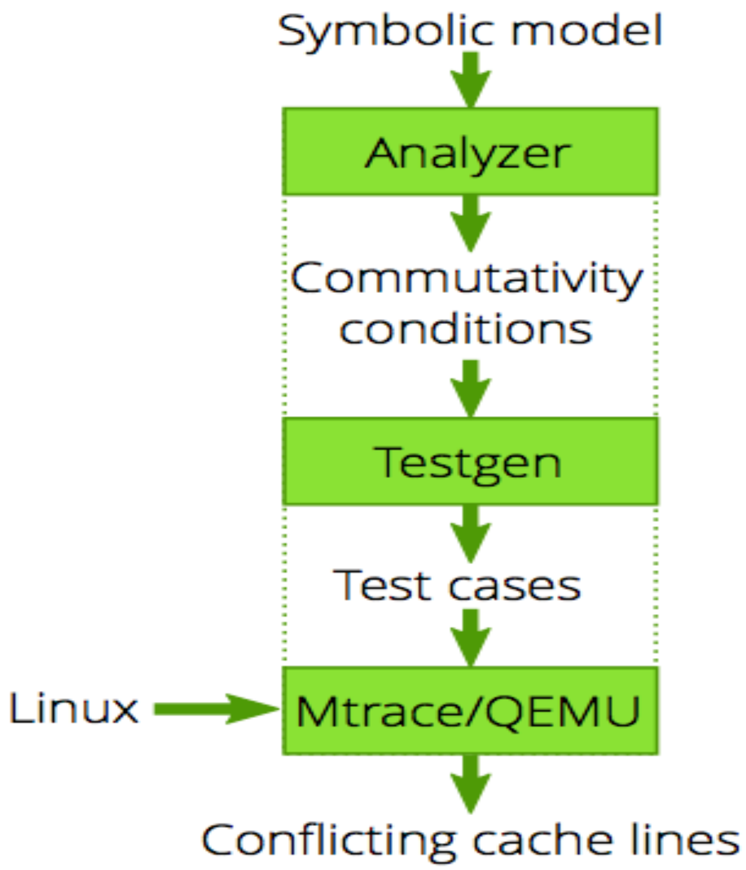
Why are commutative operations scalable?

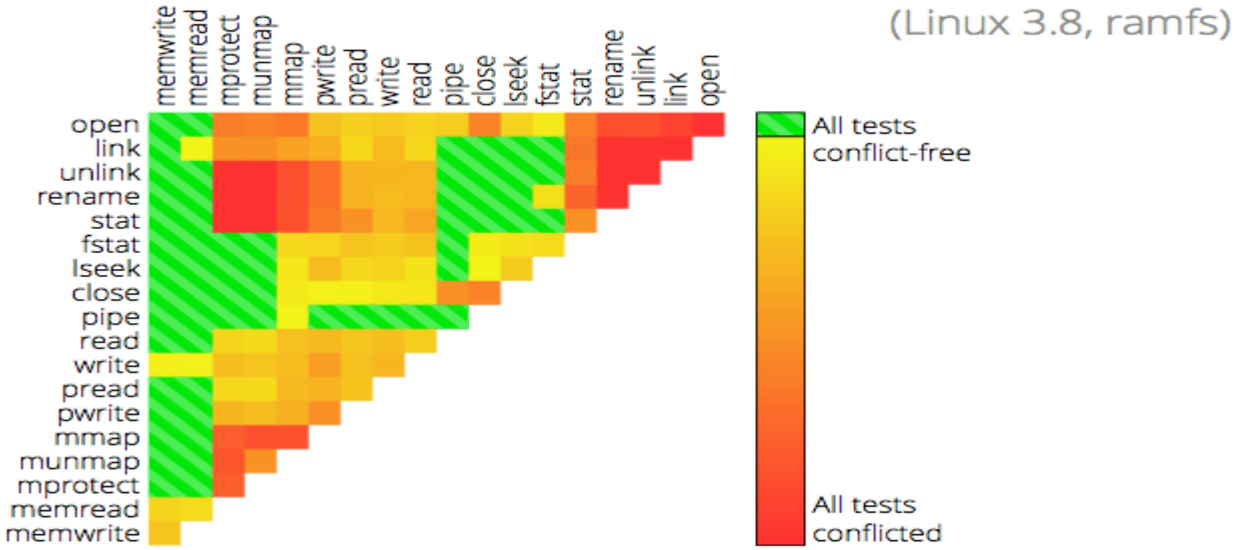
- results independent of order ⇒ communication is unnecessary
- without communication, no conflicts

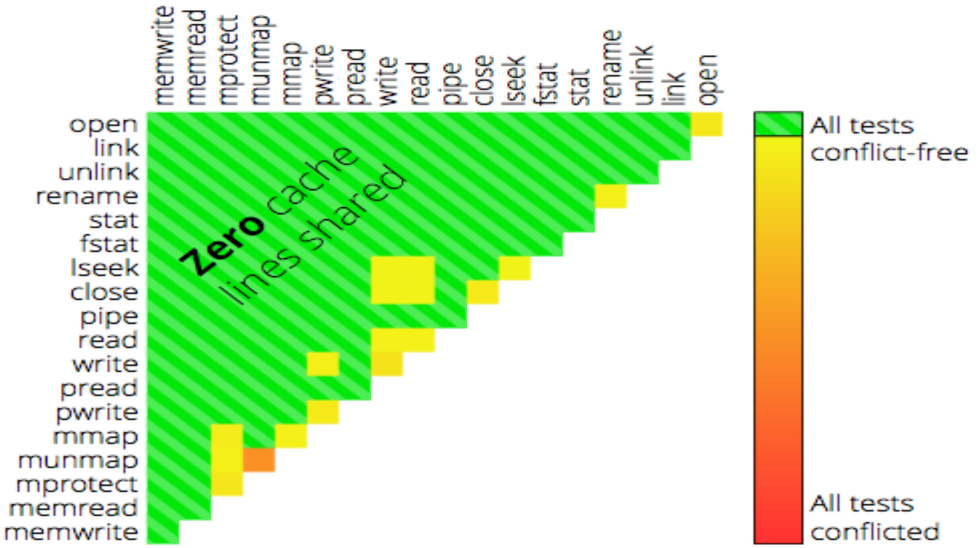
Informs software design process

- Design: design guideline for scalable interfaces
- Implementation: clear target
- Test: workload-independent testing

Commuter: An Automated Scalability Testing Tool







(sv6)

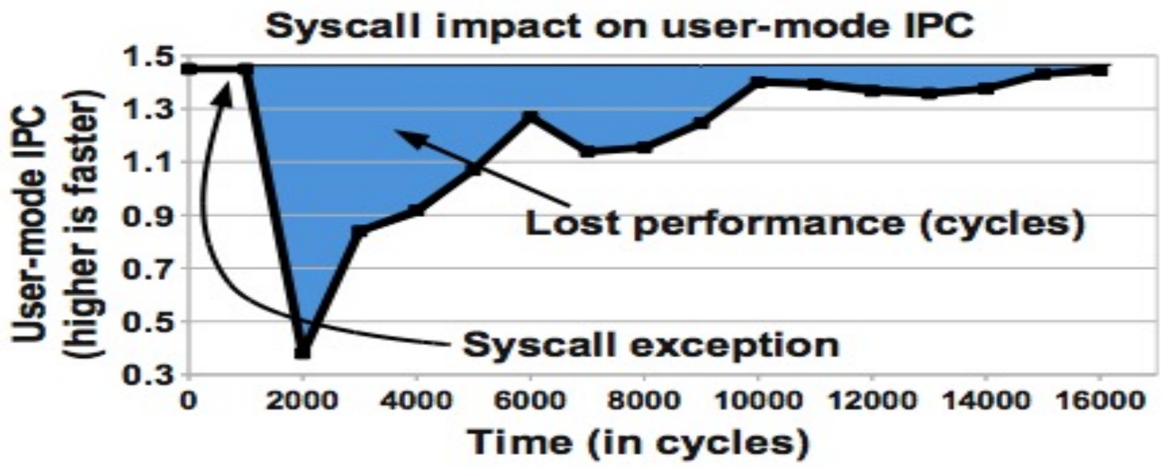
FlexSC

Context:

- 2010, commodity multicores
- U Toronto

Goal:

- Reduce context switch overhead of system



Syscall context switch:

- Usual mode switch overhead
- But: cache and TLB pollution!

Syscall	Instructions	Cycles	IPC	i-cache	d-cache	L2	L3	d-TLB
stat	4972	13585	0.37	32	186	660	2559	21
pread	3739	12300	0.30	32	294	679	2160	20
pwrite	5689	31285	0.18	50	373	985	3160	44
open+close	6631	19162	0.34	47	240	900	3534	28
mmap+munmap	8977	19079	0.47	41	233	869	3913	7
open+write+close	9921	32815	0.30	78	481	1462	5105	49

FlexSC: Flexible System Call Scheduling with Exception-Less System Calls [Soares and Stumm., 2010]

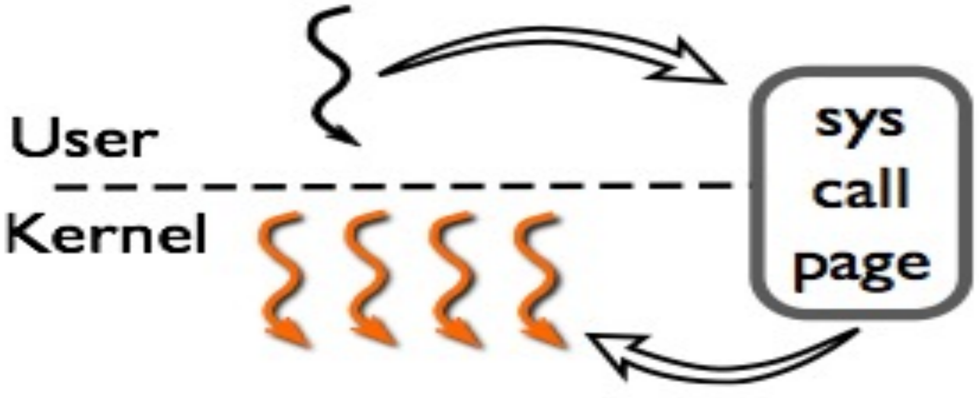
FlexSC

Asynchronous system calls

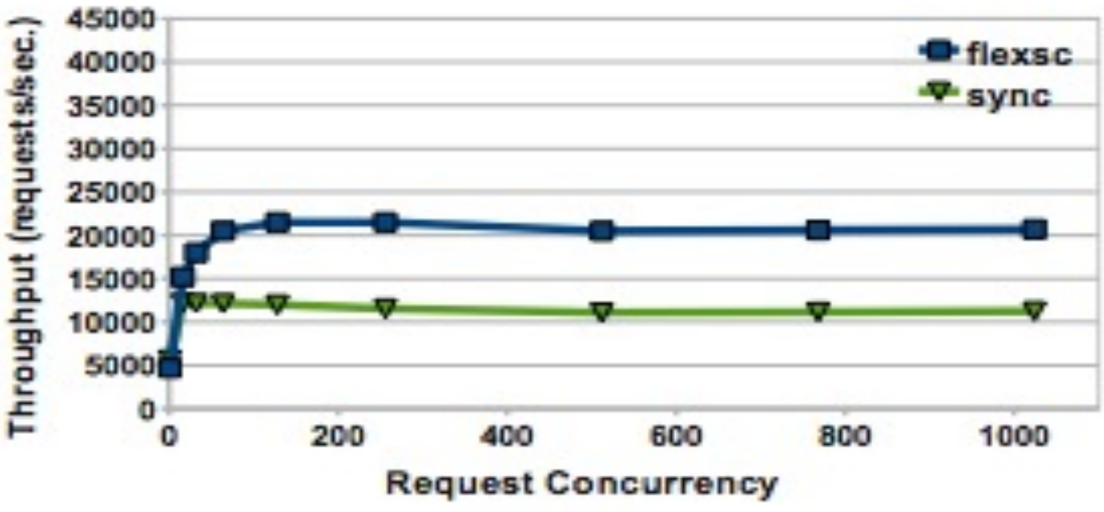
- Batch system calls
- Run them on dedicated cores

FlexSC-Threads

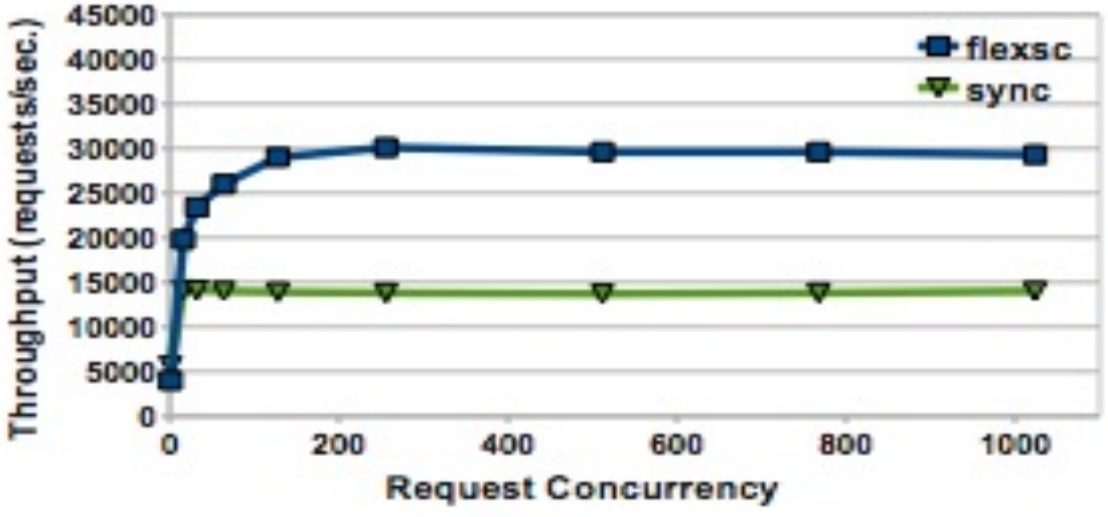
- M on N
- M >> N



FlexSC Results

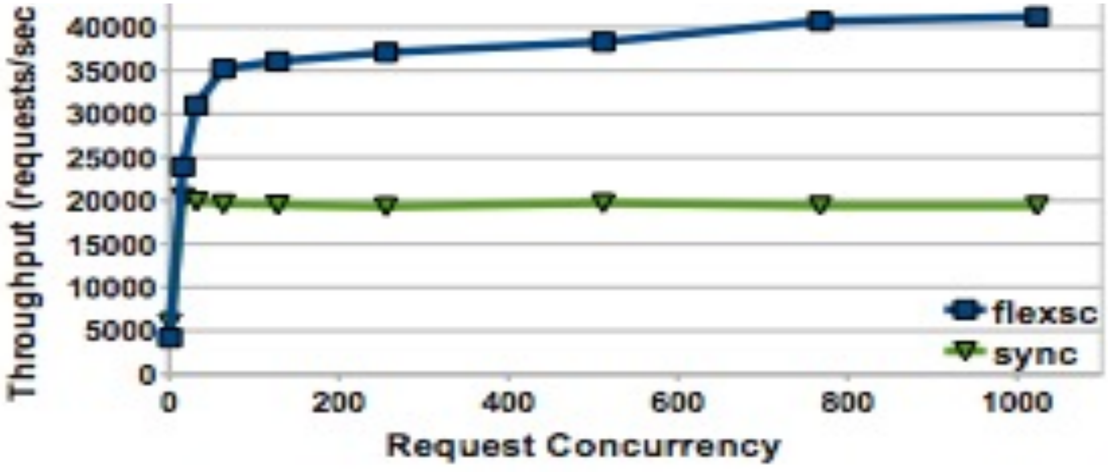


(a) 1 Core



(b) 2 Cores

Apache
FlexSC: batching,
sys call core redirect

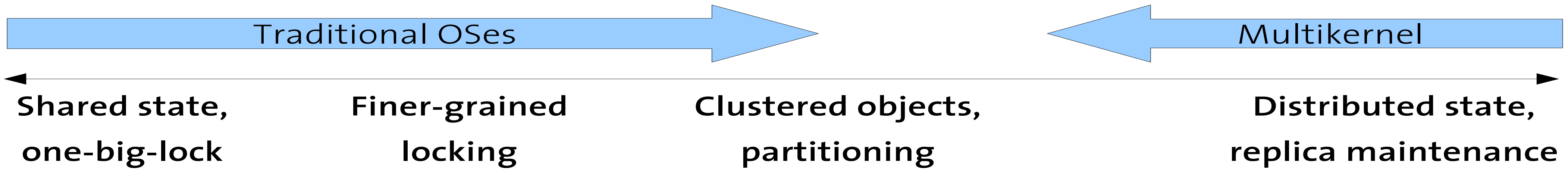


(c) 4 Cores

No sharing

Multikernel

- Barrelfish
- fos: factored operating system



The Multikernel: A new OS architecture for scalable multicore systems [Baumann et al., 2009]
http://www.barrelfish.org/

Barrelfish

Context:

- 2007 large multicore machines appearing
- 100s of cores on the horizon
- NUMA (cc and non-cc)
- ETH Zurich and Microsoft

Goals:

- Scale to many cores
- Support and manage heterogeneous hardware

Approach:

- Structure OS as *distributed system*

Design principles:

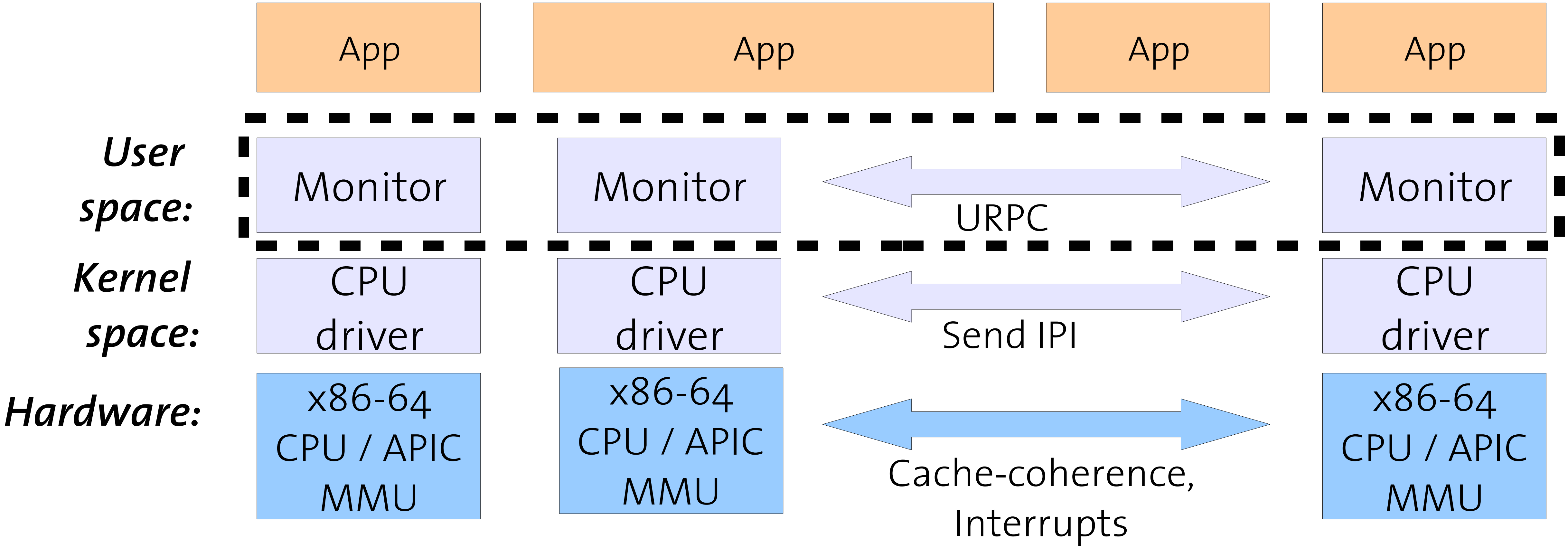
- Interprocessor communication is explicit
- OS structure hardware neutral
- State is replicated

Microkernel

- Similar to seL4: capabilities

The Multikernel: A new OS architecture for scalable multicore systems
[Baumann et al., 2009]   http://www.barrelfish.org/

Barrelfish



App
App
App
App
User space:
Monitor
Monitor
URPC
Monitor
Kernel space:
CPU driver
CPU driver
Send IPI
CPU driver
Hardware:
x86-64 CPU / APIC MMU
x86-64 CPU / APIC MMU
Cache-coherence, Interrupts
x86-64 CPU / APIC MMU

Barrelfish: Replication

Kernel + Monitor:

- Only memory shared for message channels

Monitor:

- Collectively coordinate system-wide state

System-wide state:

- Memory allocation tables
- Address space mappings
- Capability lists

What state is replicated in Barrelfish

- Capability lists

Consistency and Coordination

- Retype: two-phase commit to globally execute operation in order
- Page (re/un)mapping: one-phase commit to synchronise TLBs

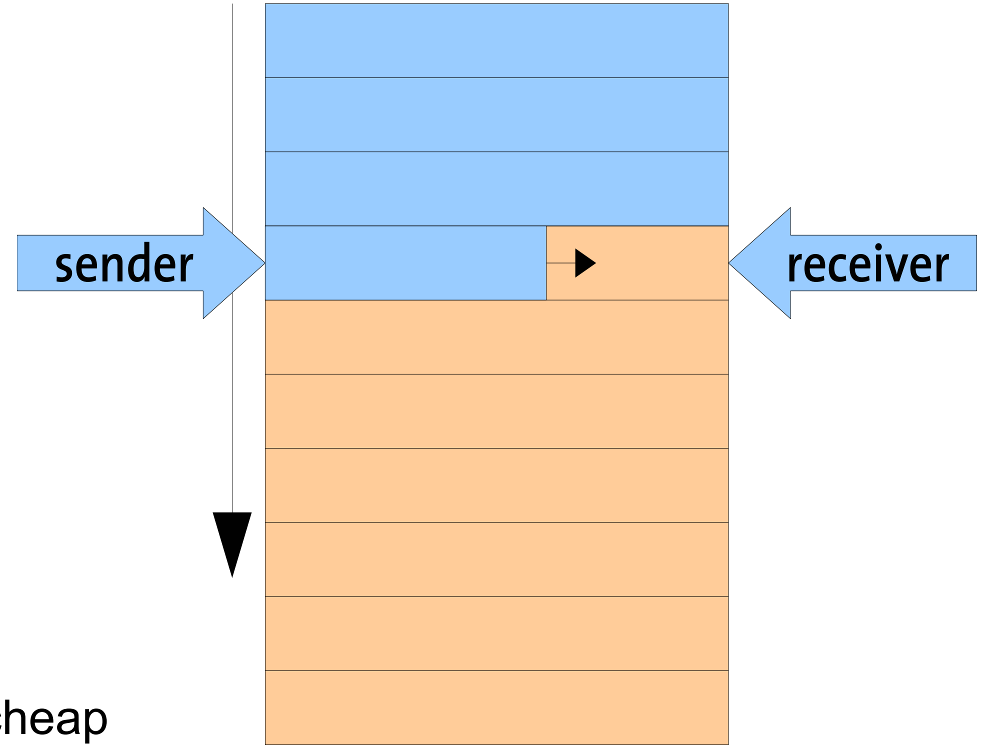
Barrelfish: Communication

Different mechanisms:

- Intra-core
  - Kernel endpoints
- Inter-core
  - URPC

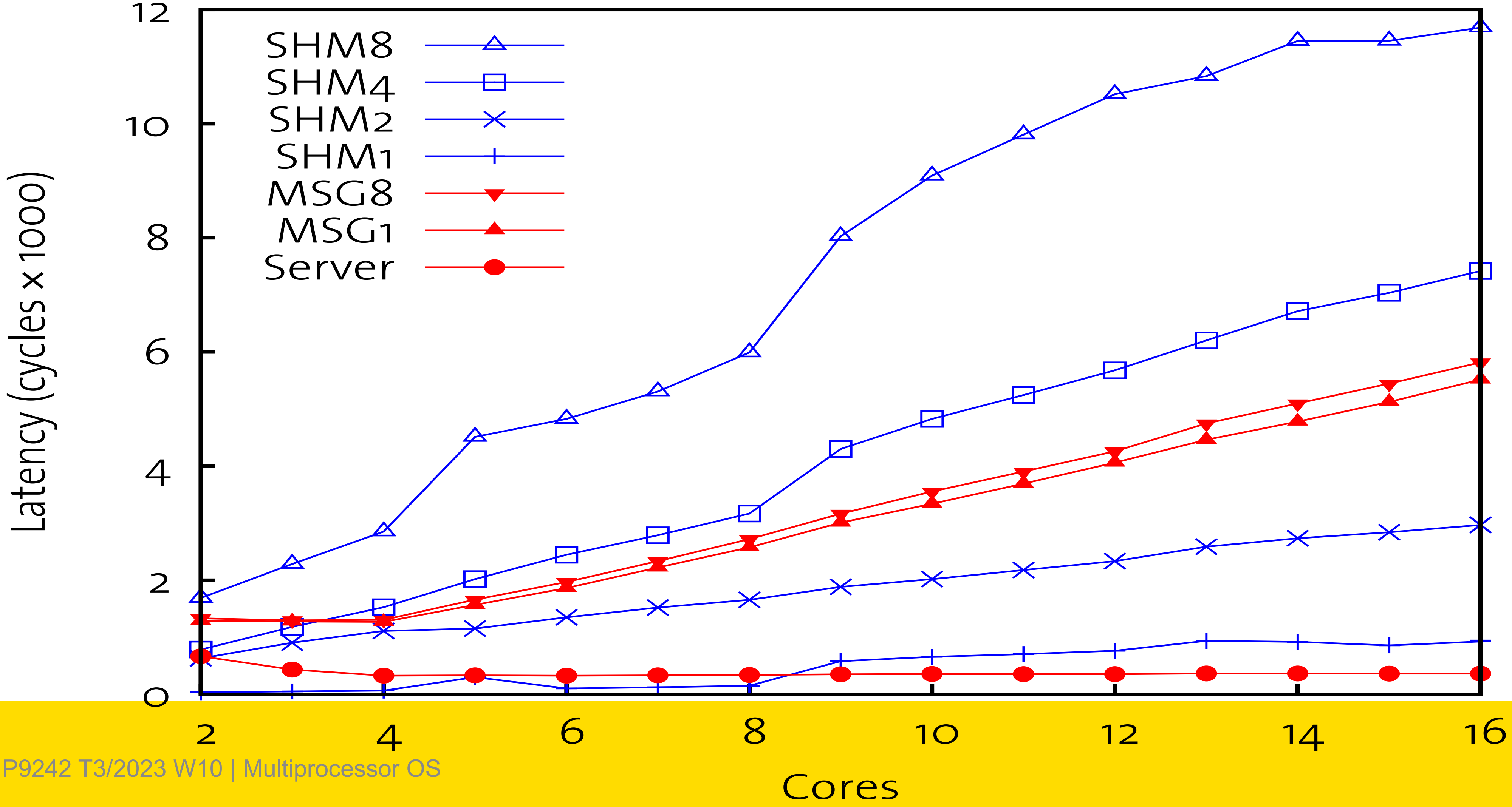
URPC

- Uses cache coherence + polling
- Shared bufffer
  - Sender writes a cache line
  - Receiver polls on cache line
  - (last word so no part message)
- Polling?
  - Cache only changes when sender writes, so poll is cheap
  - Switch to block and IPI if wait is too long.



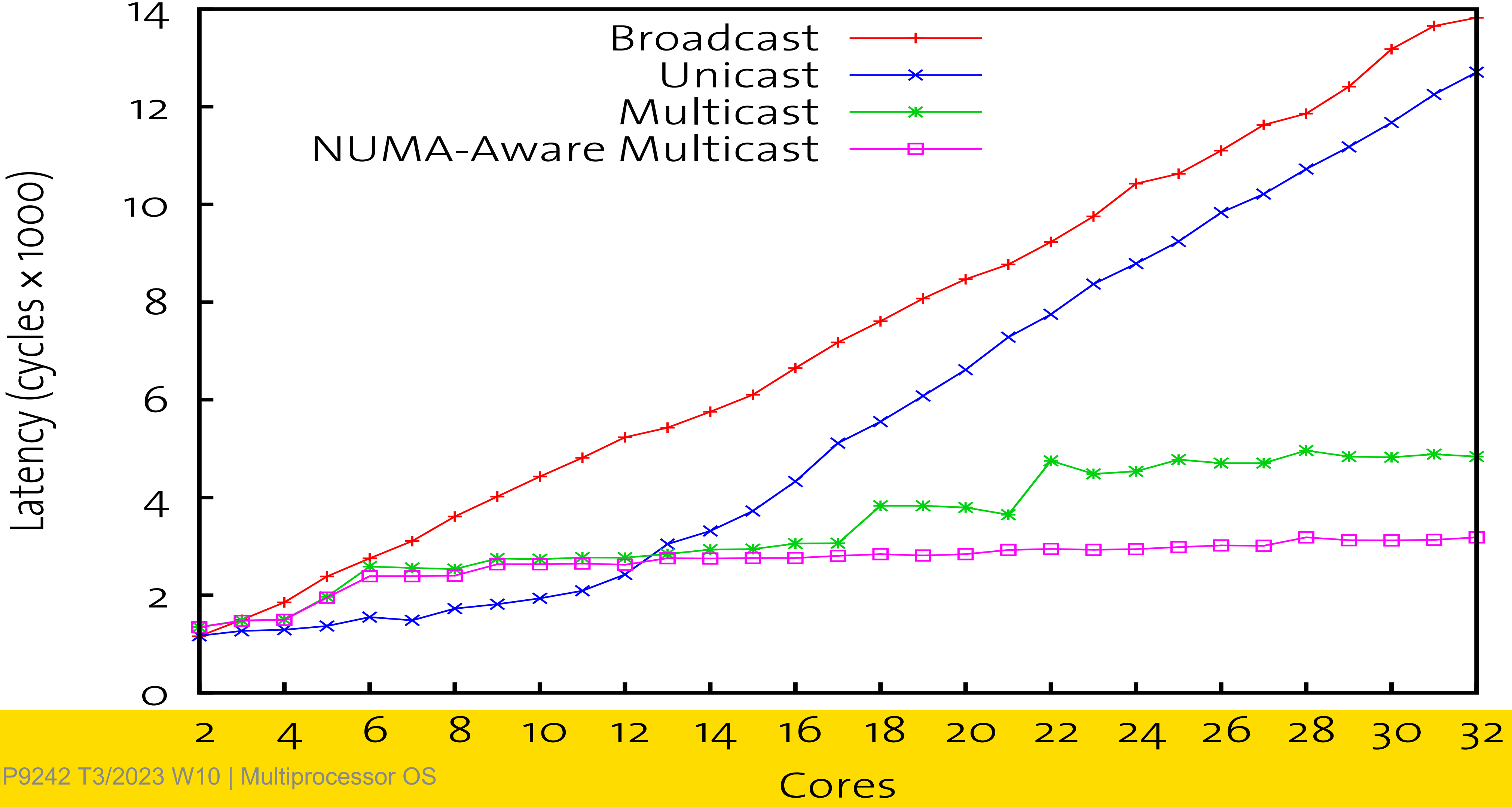
Barrelfish: Results

Message passing vs caching



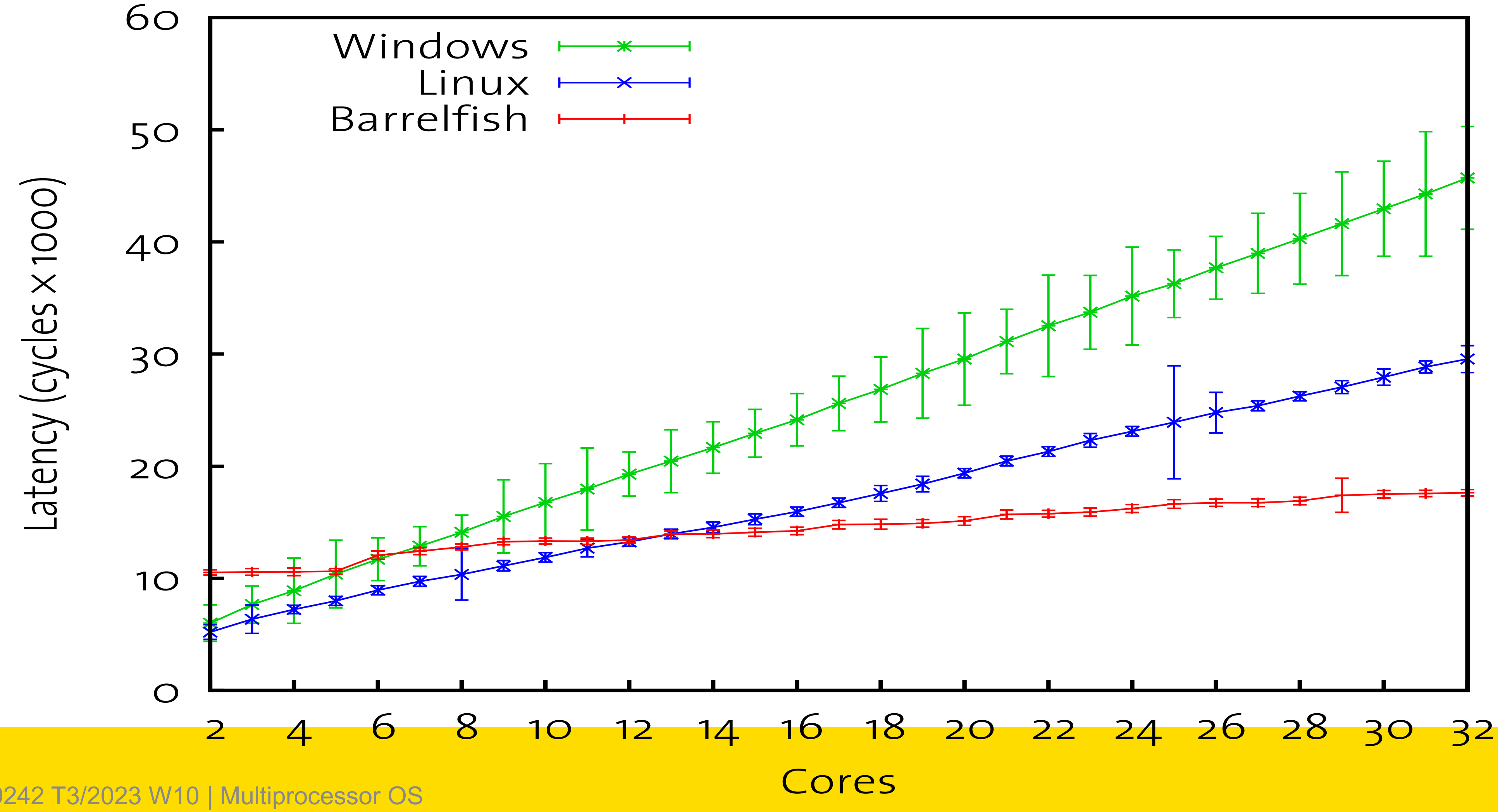
Barrelfish: Results

Broadcast vs Multicast



Barrelfish: Results

TLB shootdown



seL4

Context:

- 2013, 2022 UNSW/TS (+ Kry10, Proofcraft)
- Embedded/ARM multicore systems

Goals:

- Verified multicore kernel

Approach

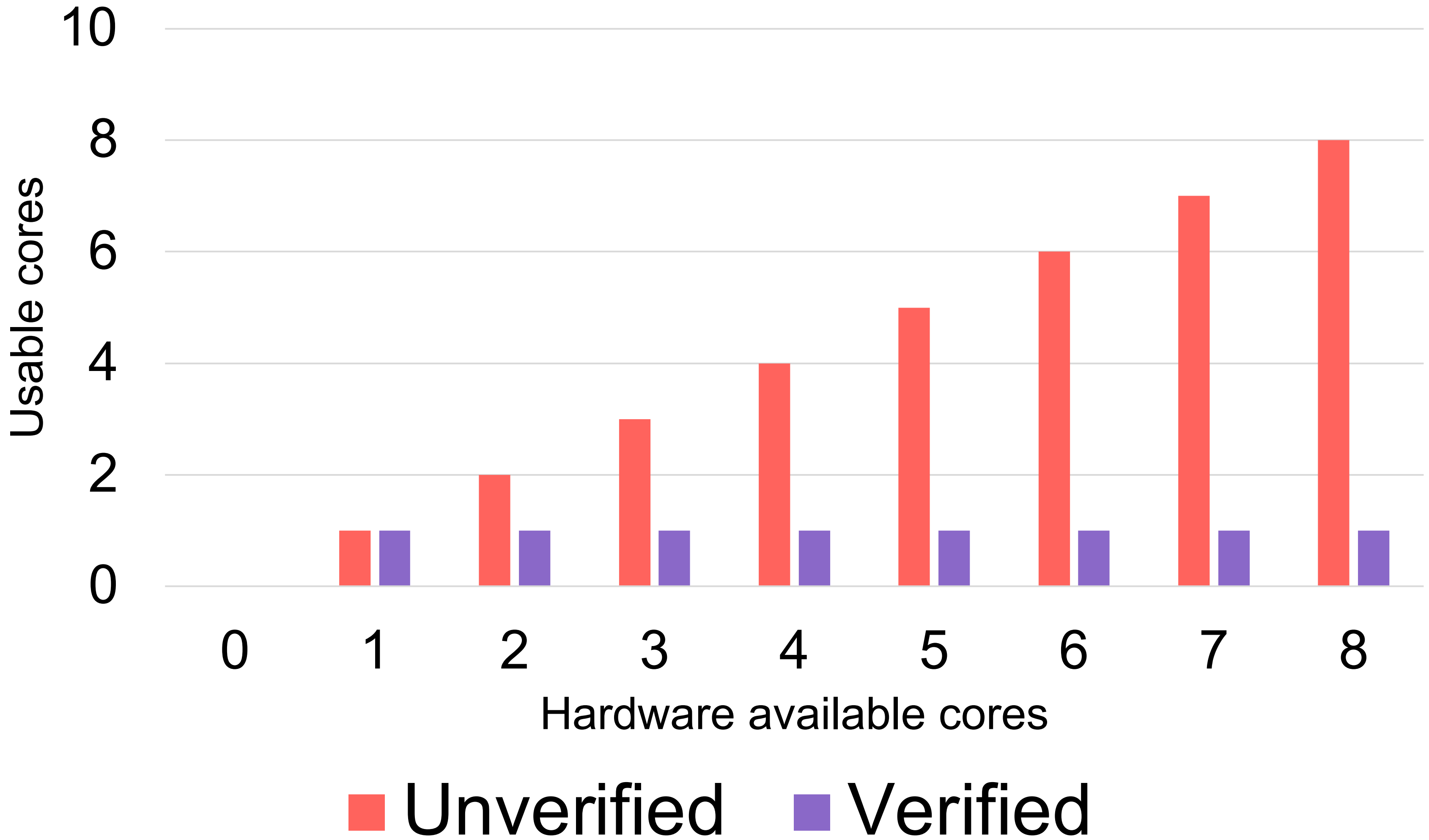
- Biglock SMP vs multikernel

Design Principles

- Divide and Conquer

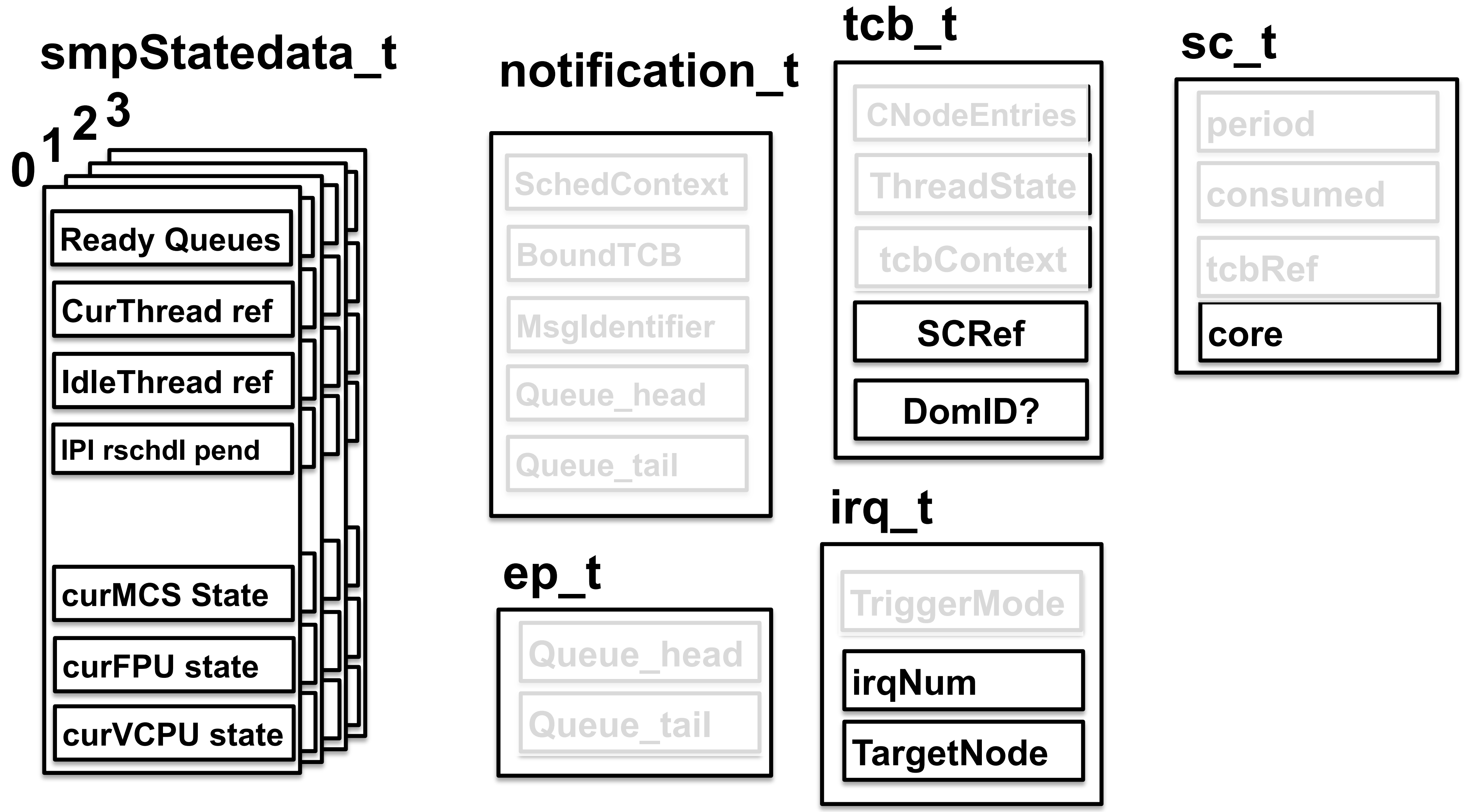
Multiprocessing on seL4 with verified kernels [McLeod, 2023]
https://sel4.systems/Foundation/Summit/2022/slides/d1_07_Multiprocessing_on_seL4_with_verified_kernels_Kent_Mcleod.pdf

Usable CPU count by kernel configuration



UNSW SYDNEY

seL4 SMP kernel (Big lock)



seL4 SMP Kernel (Big Lock)

SMP kernel has shared state

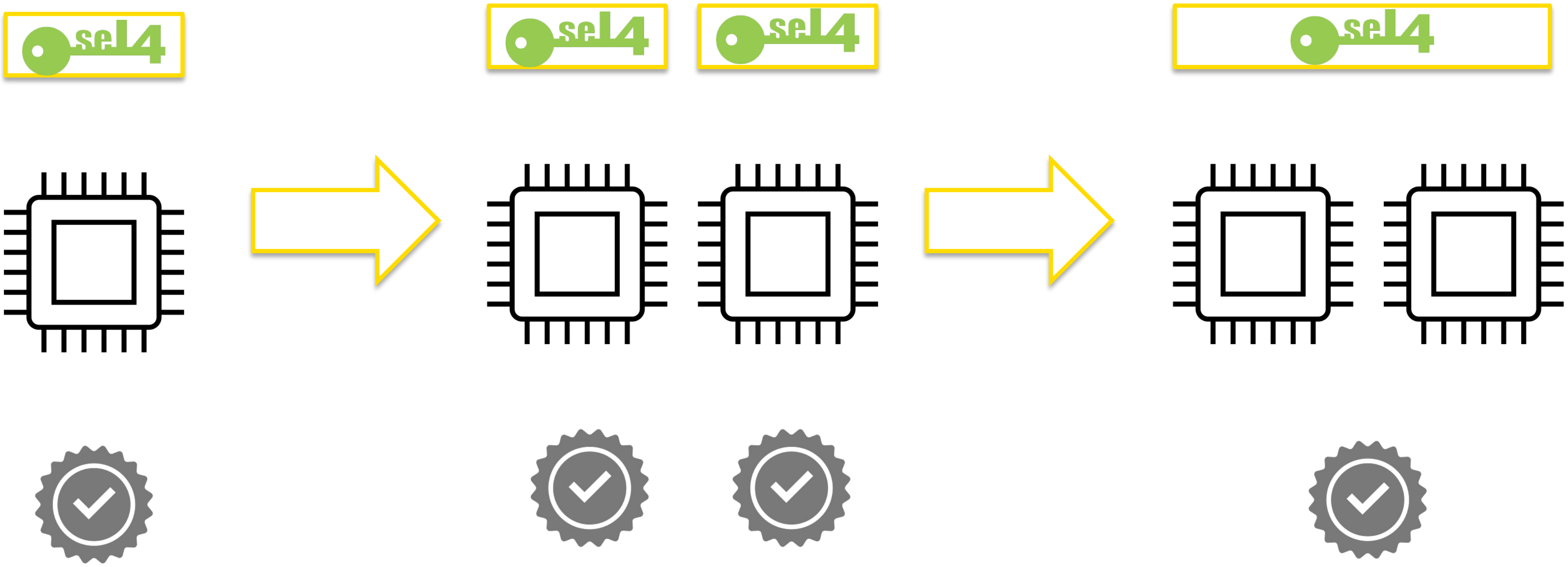
Concurrency in the kernel

Big kernel lock:

- Simplifies verification, but not by a lot initially
- Adds locking overhead to all kernel operations

Non-negligible code changes for implementing SMP design

(Re)Introducing: Partitioned multikernel



Unicore

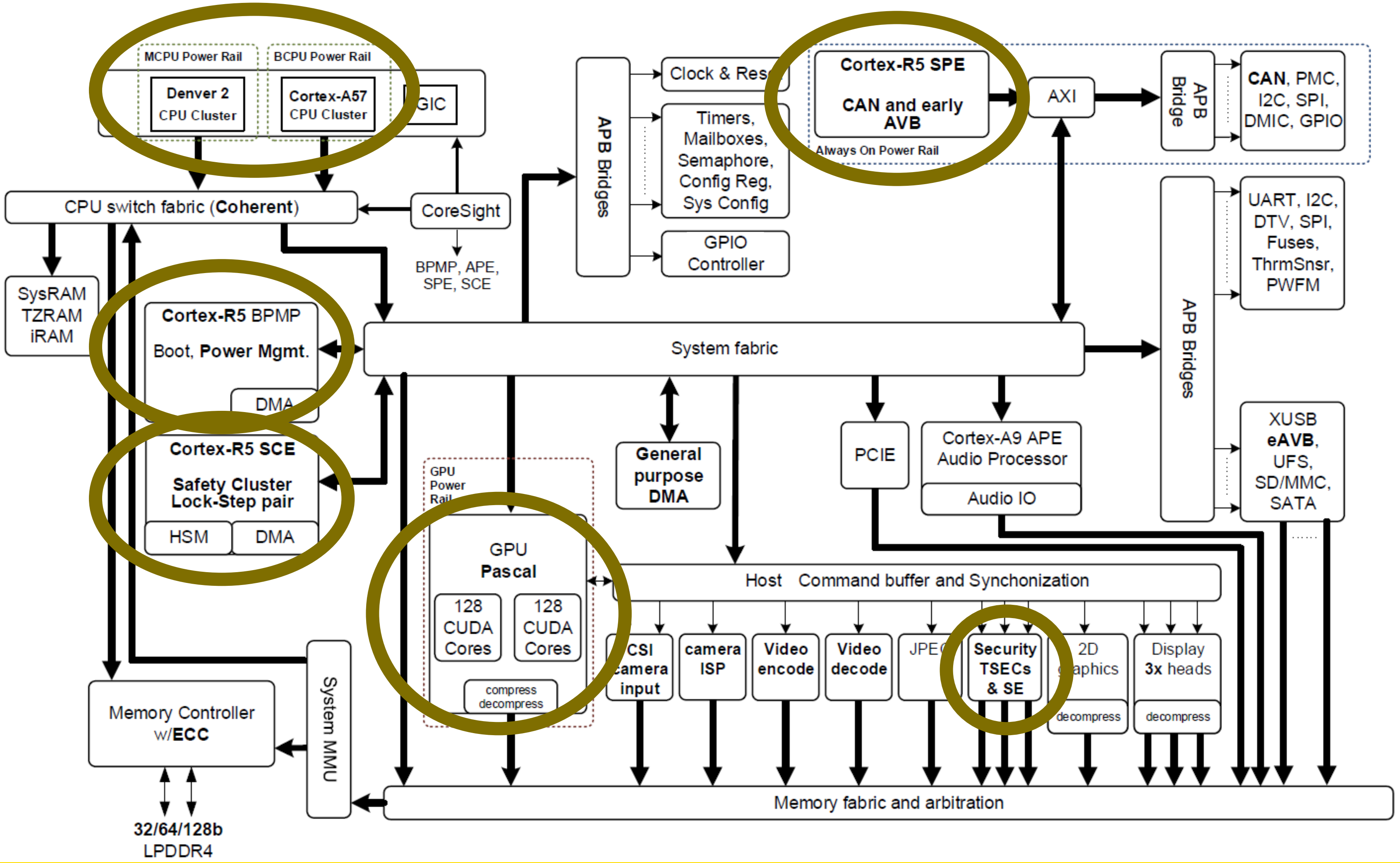
Multikernel (AMP)

SMP

What are the trade-offs?

	Multikernel	SMP
Kernel State	Partitioned	Shared
Concurrency in Kernel	No - better verification	Yes - hard to verify
Cross-core communications	Implemented at userlevel	Implemented by kernel

Dealing with Heterogeneity



De Facto OS

Context

- 2020+: highly heterogeneous SoC
- ETH Zurich

Goals

- Define a *de facto OS*
- All the memory accesses and privileges on a SoC

Approach

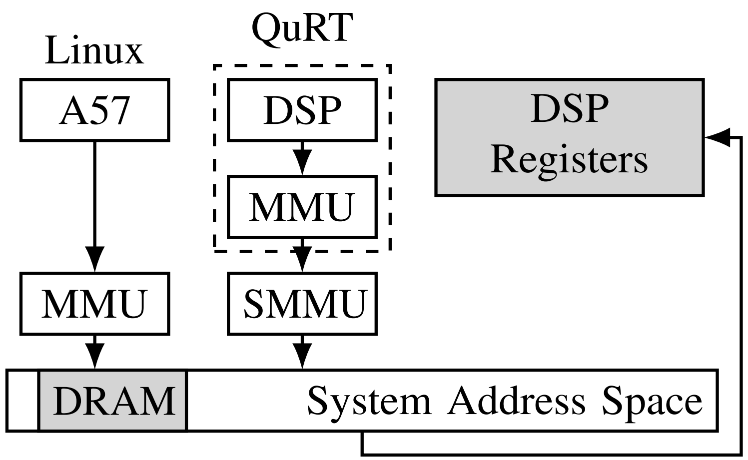
- Model the hardware and software
- Analyse it to determine trust requirements and properties

Putting out the Hardware Dumpster Fire [Fiedler et al, 2023]
https://sigops.org/s/conferences/hotos/2023/papers/fiedler.pdf

Heterogeneous SoCs – the problem

Cross-SoC Attacks

- Untrustworthy devices/peripherals
- Trusted by OS and other devices



Example: QualPWN

- over-the-air compromise of DSP
- DSP asks Linux driver to map all of physical memory for it through SMMU

How it normally works:

- Linux driver -> DSP: use this address for DMA
- DSP -> Linux driver: give me SMMU mappings for DMA

Exploit

- DSP -> Linux driver: asks for malicious SMMU mappings

Problem

- Trust driver(s) to filter out bad mappings…

Exploiting Qualcomm WLAN and Modem Over the Air [Gong et al, 2019]
https://i.blackhat.com/USA-19/Thursday/us-19-Pi-Exploiting-Qualcomm-WLAN-And-Modem-Over-The-Air-wp.pdf
mmapx: uniform memory protection in a heterogeneous world [Achermann et al, 2021]
https://people.inf.ethz.ch/troscoe/pubs/achermann-hotos-2021.pdf

Modelling the whole system

OS, isolation, and protection

- OS: provide protection and isolation between application programs
- Kernel (e.g. Linux, seL4) not the most privileged software on machine

De facto OS

- Consider HW (and firmware) that reads/writes to address spaces
  - DMA access: e.g. NICs, WiFi chips, video co-processors
  - Other (non-main memory) address spaces
- Formal specification (Sockeye3):
  - directed graph: nodes = address spaces, edges = translation between address spaces
  - Context: can generate memory operations (CPU, GPU, DMA engine, etc.)
  - Translation regions: contains metadata that configures translation operations
  - Component: complex behaviour = Rust code

Analysis

De facto OS characteristics

- No design
- Many parts cannot be changed

Goals

- Make security and correctness claims about de facto OS
- Understand how to Improve a real-world de facto OS

Analysis

- Compute overlaps between “victim” context and other contexts (critical regions)
  - (i.e. which agents can read and write which RAM regions and control registers)
  - -> integrity, confidentiality violations
- What trust assumptions need to change (and how) to remove violations?

Status

- i.MX8 8X model
- Stay tuned…

Summary

Summary

Trends in multicore

- Scale (100+ cores)
- NUMA
- No cache coherence
- Distributed system
- Heterogeneity

OS design guidelines

- Avoid shared data
- Explicit communication
- Locality

Approaches to multicore OS

- Partition the machine (Disco, Tessellation)
- Reduce sharing (K42, Corey, Linux, FlexSC, scalable commutativity)
- No sharing (Barrelfish, fos)
- Dealing with heterogeneity (de facto OS)