



# Multiprocessor OS

COMP9242 – Advanced Operating Systems

Ihor Kuz | [ihor.kuz@data61.csiro.au](mailto:ihor.kuz@data61.csiro.au)

T2/2019 Week 10

[www.data61.csiro.au](http://www.data61.csiro.au)



**UNSW**  
AUSTRALIA

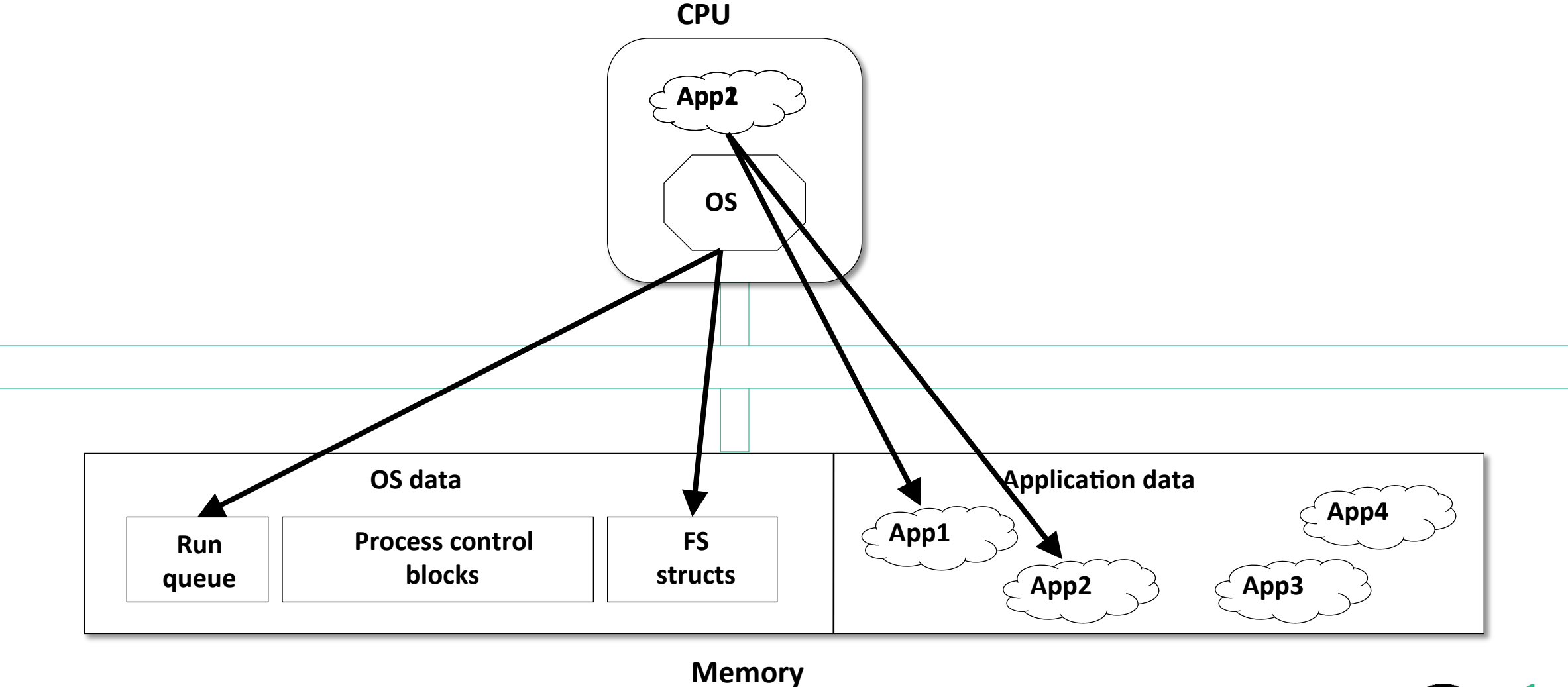
# Overview

- Multiprocessor OS (Background and Review)
  - How does it work? (Background)
  - Scalability (Review)
- Multiprocessor Hardware
  - Contemporary systems (Intel, AMD, ARM, Oracle/Sun)
  - Experimental and Future systems (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)

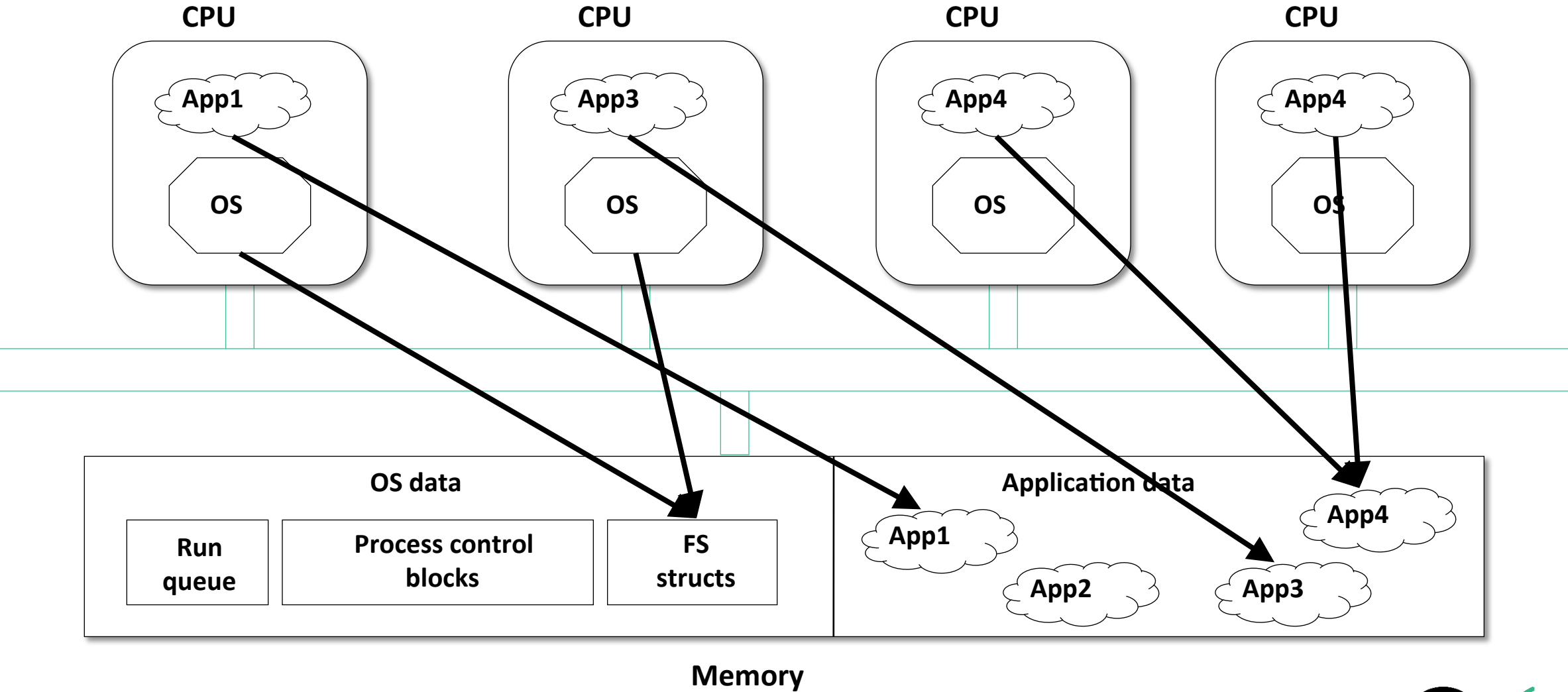
# Multiprocessor OS



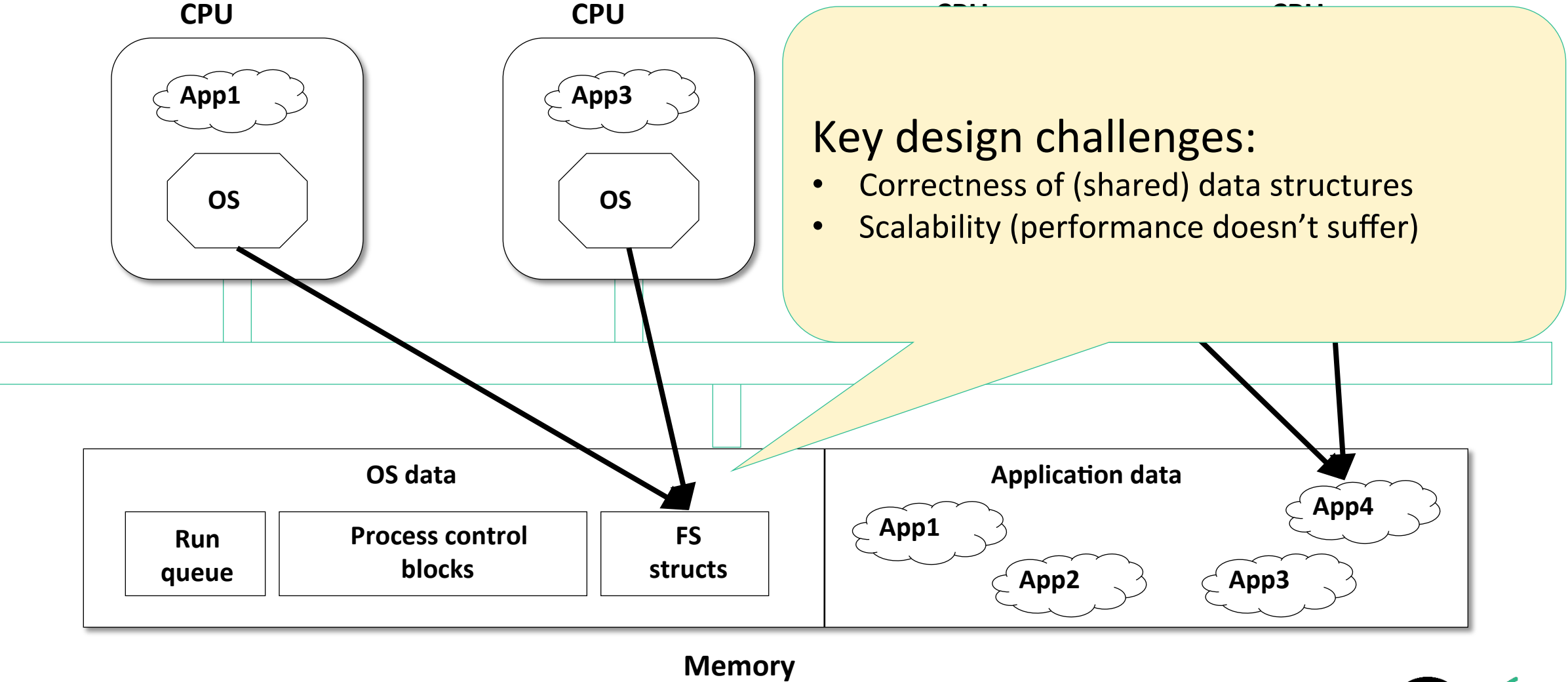
# Uniprocessor OS



# Multiprocessor OS



# Multiprocessor OS

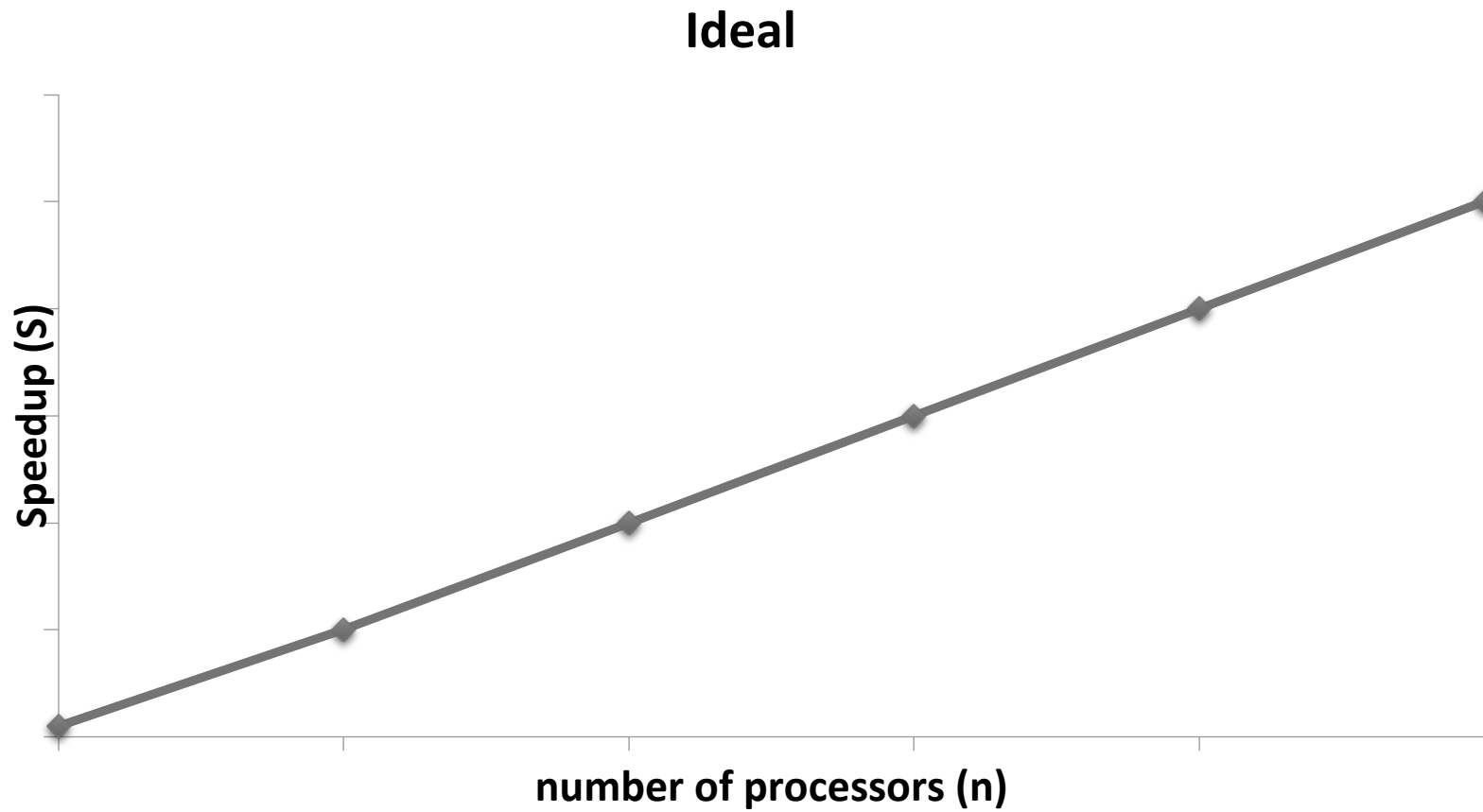


# Correctness of Shared Data

- Concurrency control
  - Locks
  - Semaphores
  - Transactions
  - Lock-free data structures
- We know how to do this:
  - In the application
  - In the OS

# Scalability

Speedup as more processors added



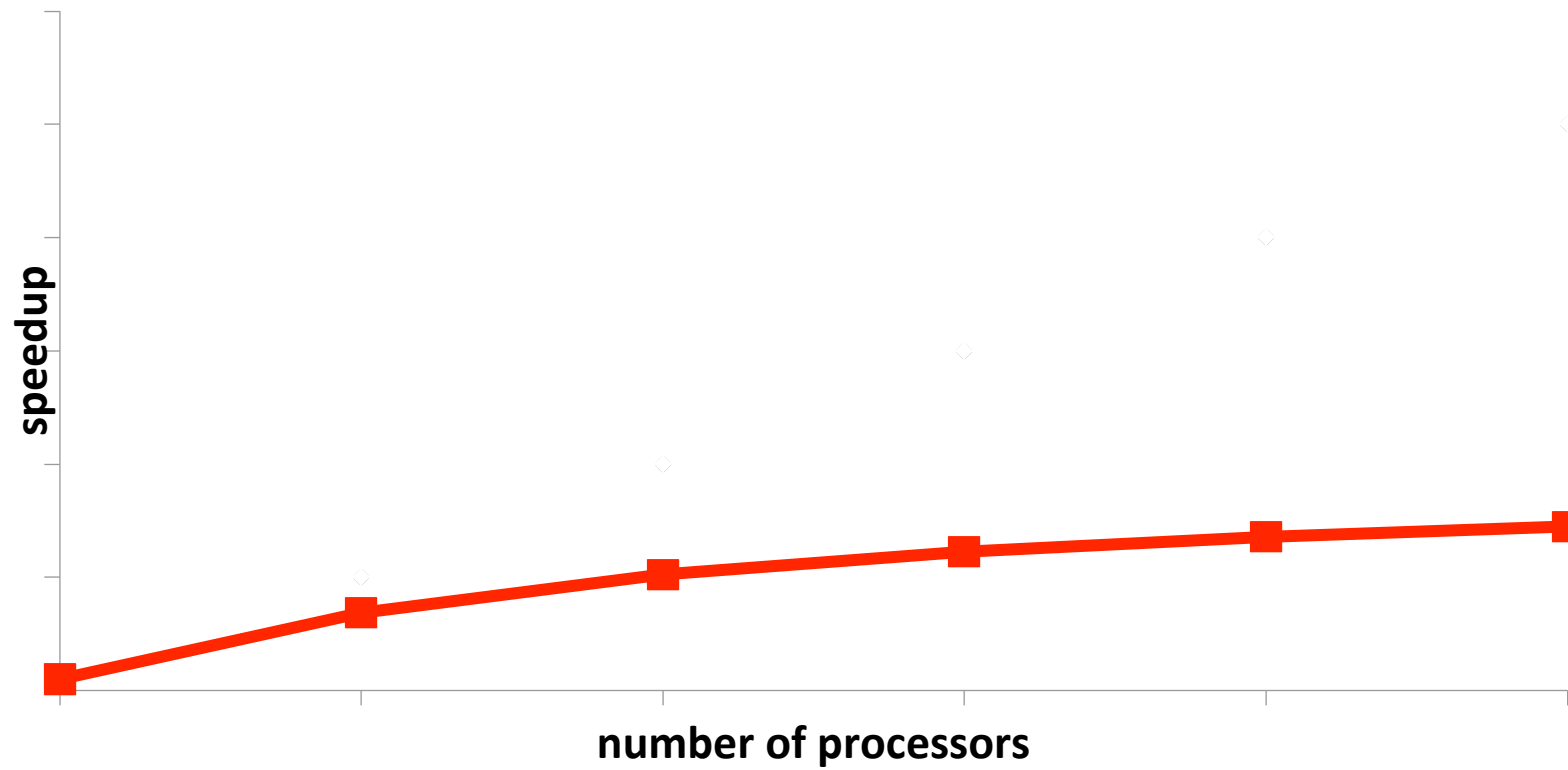
$$S(N) = \frac{T_1}{T_N}$$



# Scalability

Speedup as more processors added

Reality



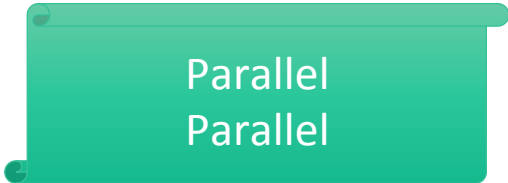
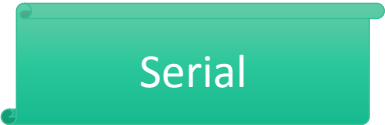
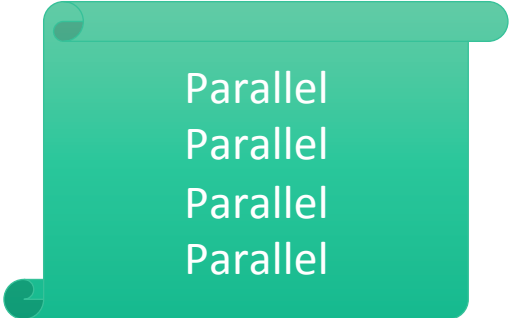
$$S(N) = \frac{T_1}{T_N}$$

# Scalability and Serialisation

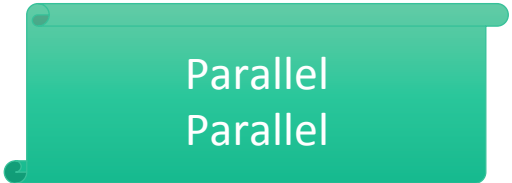
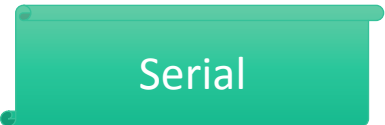
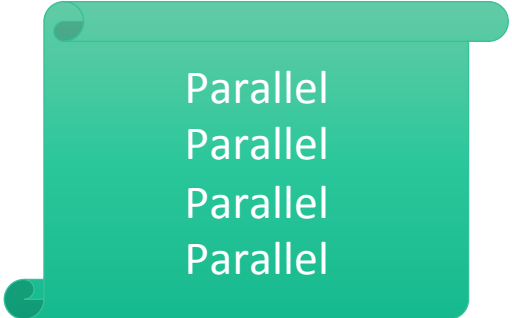
## Parallel Program



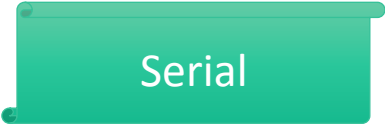
Processor 1



Processor 2



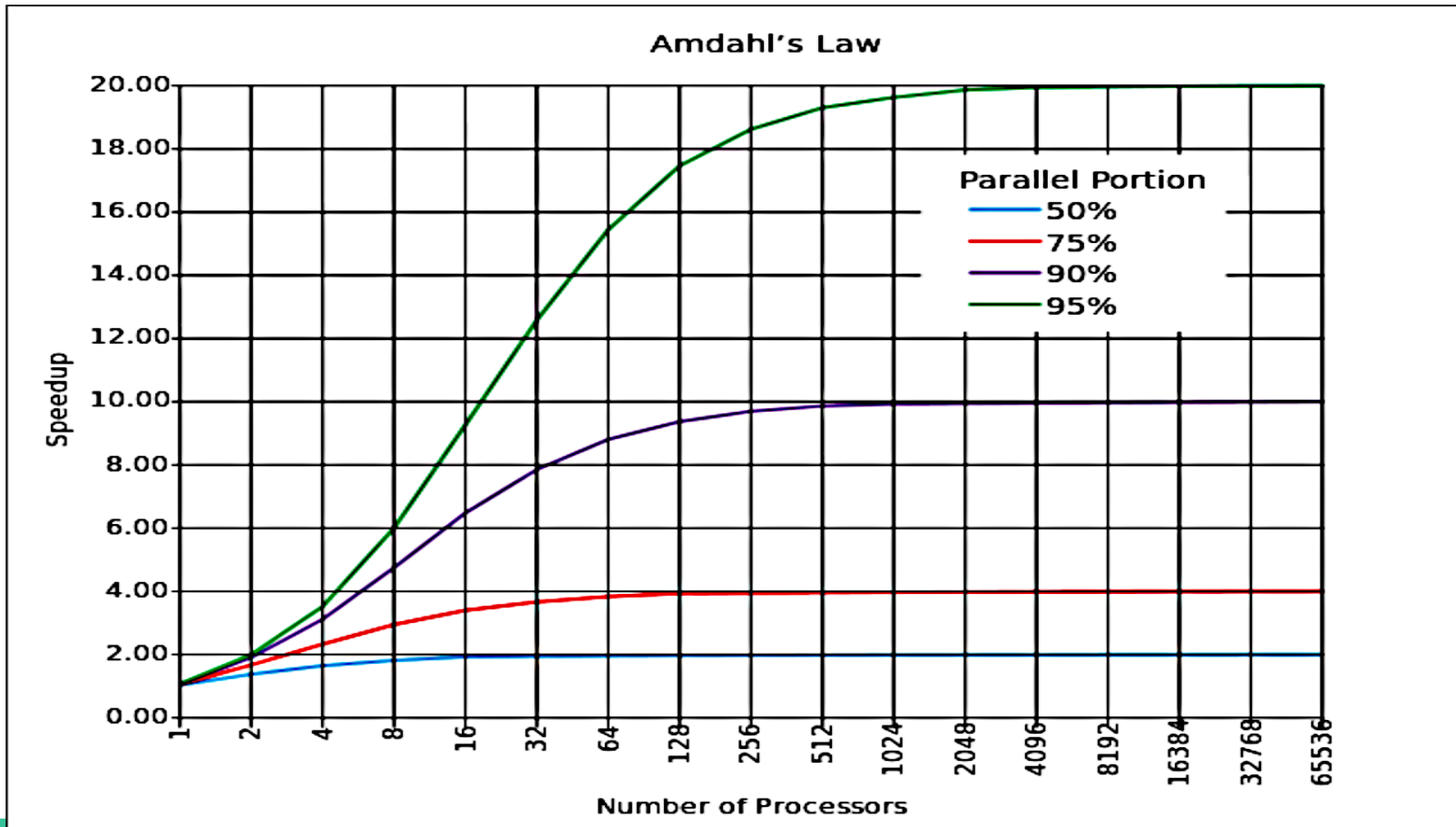
Processor 3



# Scalability and Serialisation

Remember Amdahl's law

- Serial (non-parallel) portion: when application not running on all cores
- Serialisation prevents scalability



$$T_1 = 1 = (1 - P) + P$$

$$T_N = (1 - P) + \frac{P}{N}$$

$$S(N) = \frac{T_1}{T_N} = \frac{1}{(1 - P) + \frac{P}{N}}$$

$$S(\infty) \rightarrow \frac{1}{(1 - P)}$$



# Serialisation

Where does serialisation show up?

- Application (e.g. access shared app data)
- OS (e.g. performing syscall for app) **How much time is spent in OS?**

## Sources of Serialisation

Locking (explicit serialisation)

- Waiting for a lock → **stalls self**
- Lock implementation:
  - Atomic operations lock bus → **stalls everyone waiting for memory**
  - Cache coherence traffic loads bus → **stalls others waiting for memory**

Memory access (implicit)

- Relatively high latency to memory → **stalls self**

Cache (implicit)

- **Processor stalled** while cache line is fetched or invalidated
- Affected by latency of interconnect
- Performance depends on data size (cache lines) and contention (number of cores)

# More Cache-related Serialisation

## False sharing

- Unrelated data structs share the same cache line
  - Accessed from different processors
- Cache coherence traffic and delay

## Cache line bouncing

- Shared R/W on many processors
  - E.g: bouncing due to locks: each processor spinning on a lock brings it into its own cache
- Cache coherence traffic and delay

## Cache misses

- Potentially direct memory access → stalls self
- When does cache miss occur?
  - Application accesses data for the first time, Application runs on new core
  - Cached memory has been evicted
    - Cache footprint too big, another app ran, OS ran

# Multiprocessor Hardware



# Multi-What?

- Terminology:
  - core, die (chip), package (module, processor, CPU)
- Multiprocessor, SMP
  - >1 separate processors, connected by off-processor interconnect
- Multithread, SMT
  - >1 hardware threads in a single processing core
- Multicore, CMP
  - >1 processing cores in a single die, connected by on-die interconnect
- Multicore + Multiprocessor
  - >1 multicore dies in a package (multi-chip module), on-processor interconnect
  - >1 multicore processors, off-processor interconnect
- Manycore
  - Lots (>100) of cores

# Interesting Properties of Multiprocessors

- Scale and Structure
  - How many cores and processors are there
  - What kinds of cores and processors are there
  - How are they organised (access to IO, etc.)
- Interconnect
  - How are the cores and processors connected
- Memory Locality and Caches
  - Where is the memory
  - What is the cache architecture
- Interprocessor Communication
  - How do cores and processors send messages to each other

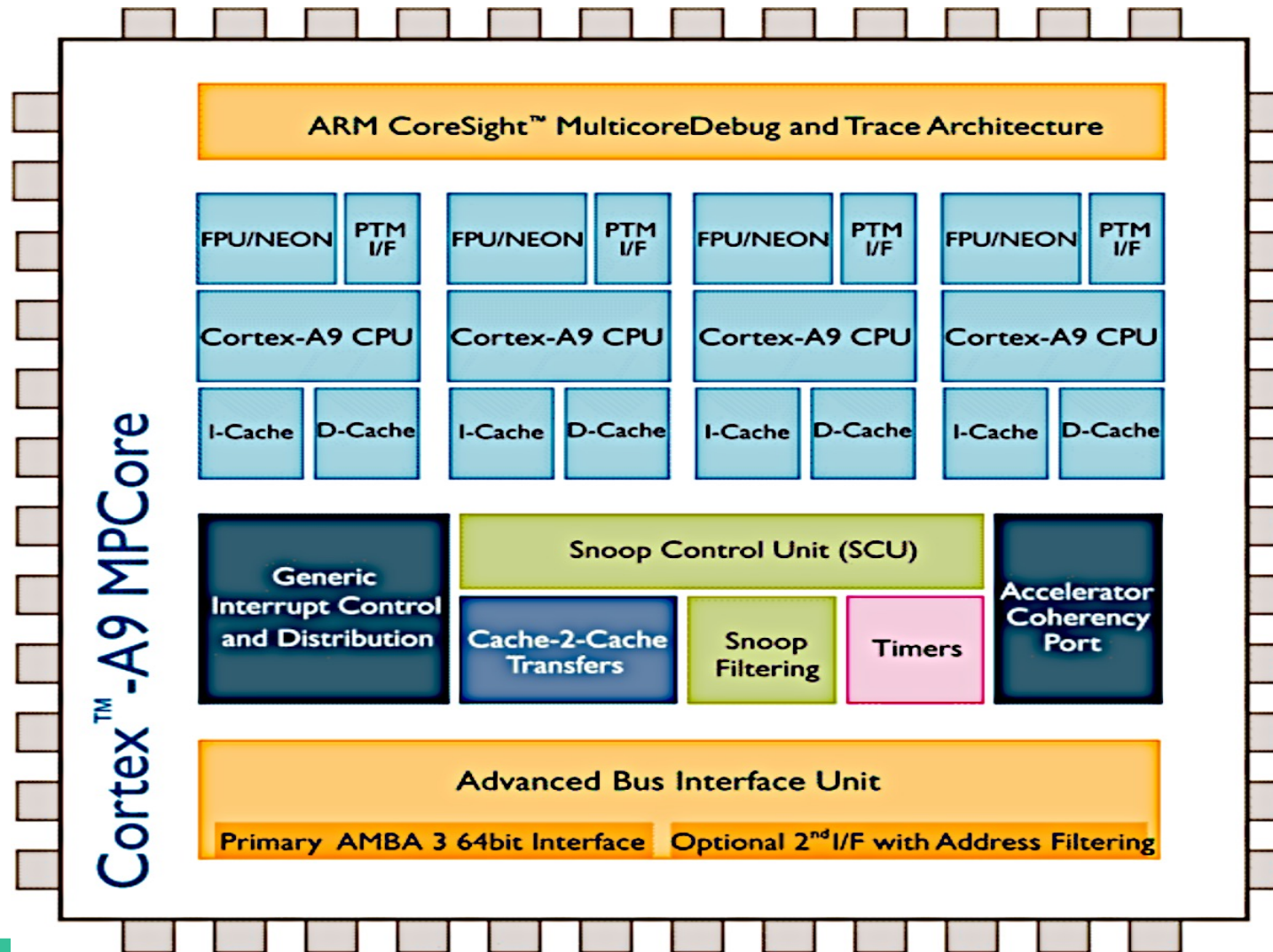


# Contemporary Multiprocessor Hardware

- Intel:
  - Nehalem, Westmere: 10 core, QPI
  - Sandy Bridge, Ivy Bridge: 5 core, ring bus, integrated GPU, L3, IO
  - Haswell (Broadwell): 18+ core, ring bus, transactional memory, slices (EP)
  - Skylake (SP): mesh architecture
- AMD:
  - K10 (Opteron: Barcelona, Magny Cours): 12 core, Hypertransport
  - Bulldozer, Piledriver, Steamroller (Opteron, FX)
    - 16 core, Clustered Multithread: module with 2 integer cores
  - Zen: on die NUMA: CPU Complex (CCX) (4 core, private L3)
  - Zen 2: chiplets (2xCCX) chiplets, IO die (incl mem controller)
- Oracle (Sun) UltraSparc T1,T2,T3,T4,T5 (Niagara), M5,M7
  - T5: 16 cores, 8 threads/core (2 simultaneous), crossbar, 8 sockets,
  - M8: 32 core, 8 threads, on chip network, 8 sockets, 5GHz
- ARM Cortex A9, A15 MPCore, big.LITTLE, DynamIQ
  - 4 -8 cores, big.LITTLE: A7 + A15, dynamIQ: A75 + A55

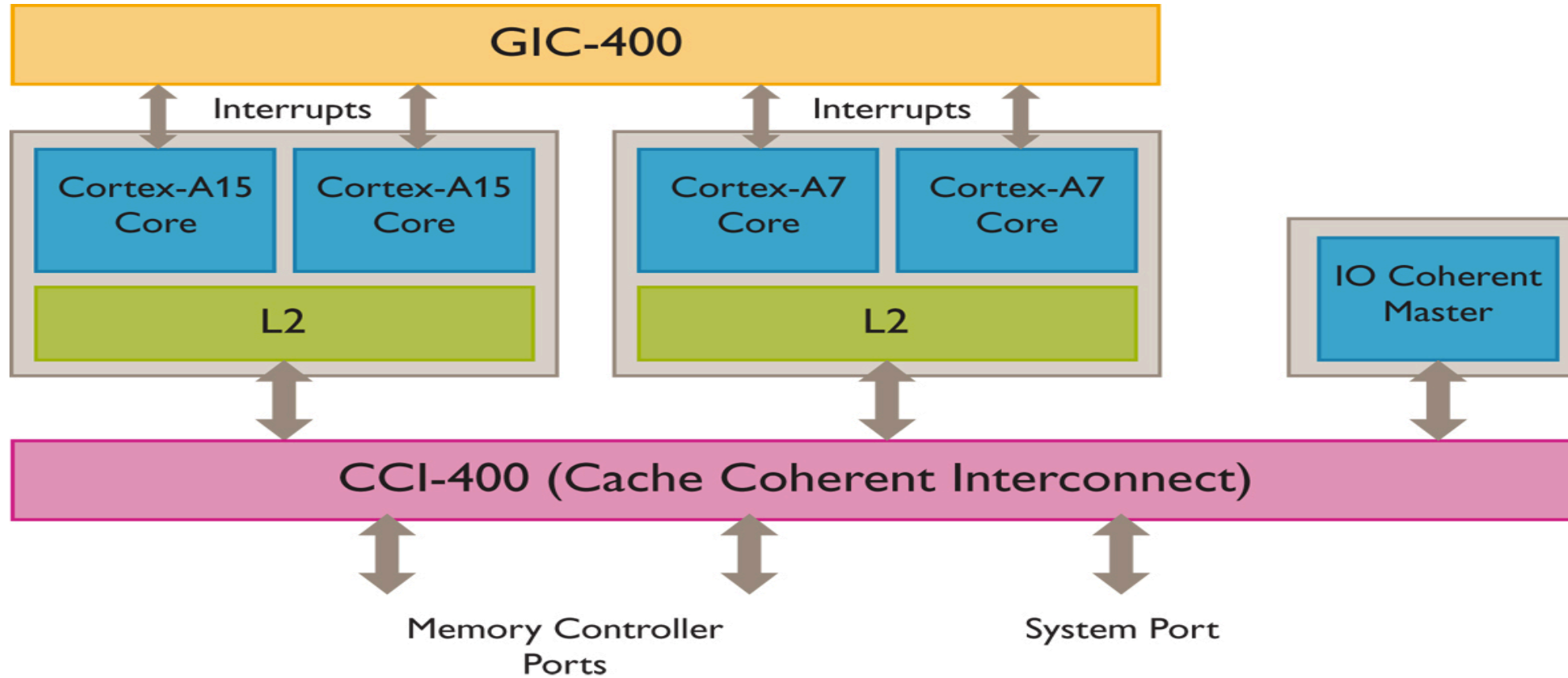
# Scale and Structure

- ARM Cortex A9



# Scale and Structure

- ARM big.LITTLE



# Scale and Structure

Conventional  
big.LITTLE

DynamiQ  
big.LITTLE



Quad  
Cortex-A53



1b+2L



1b+3L



Octa  
Cortex-A53



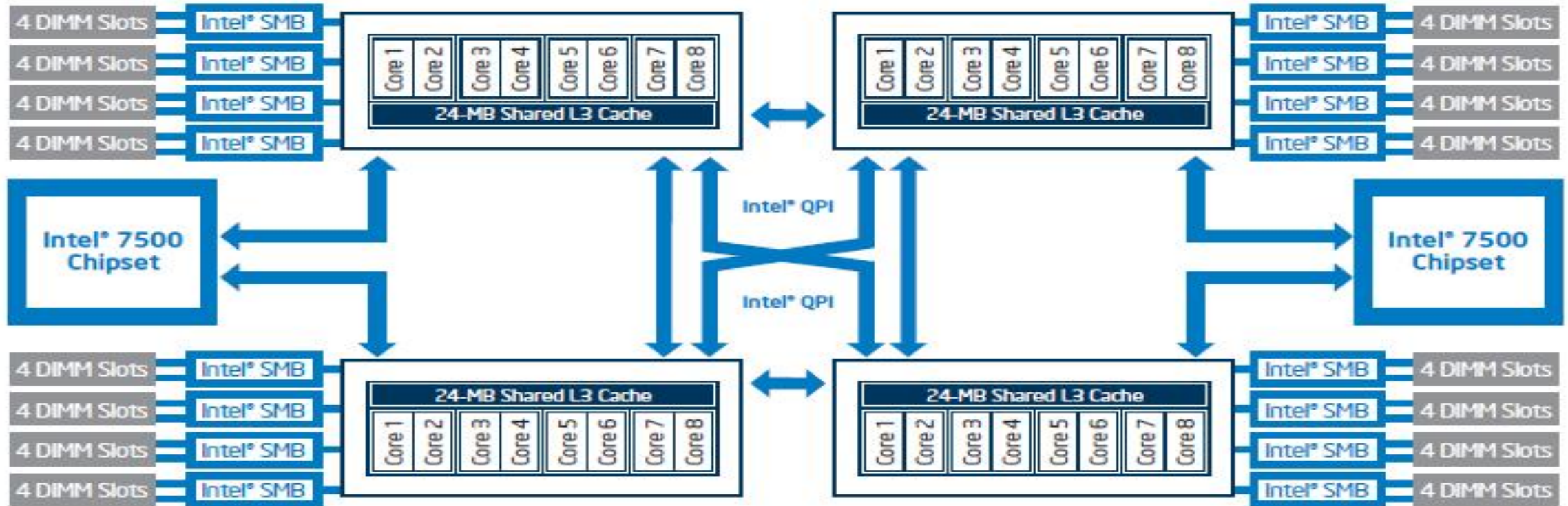
1b+4L



1b+7L

# Scale and Structure

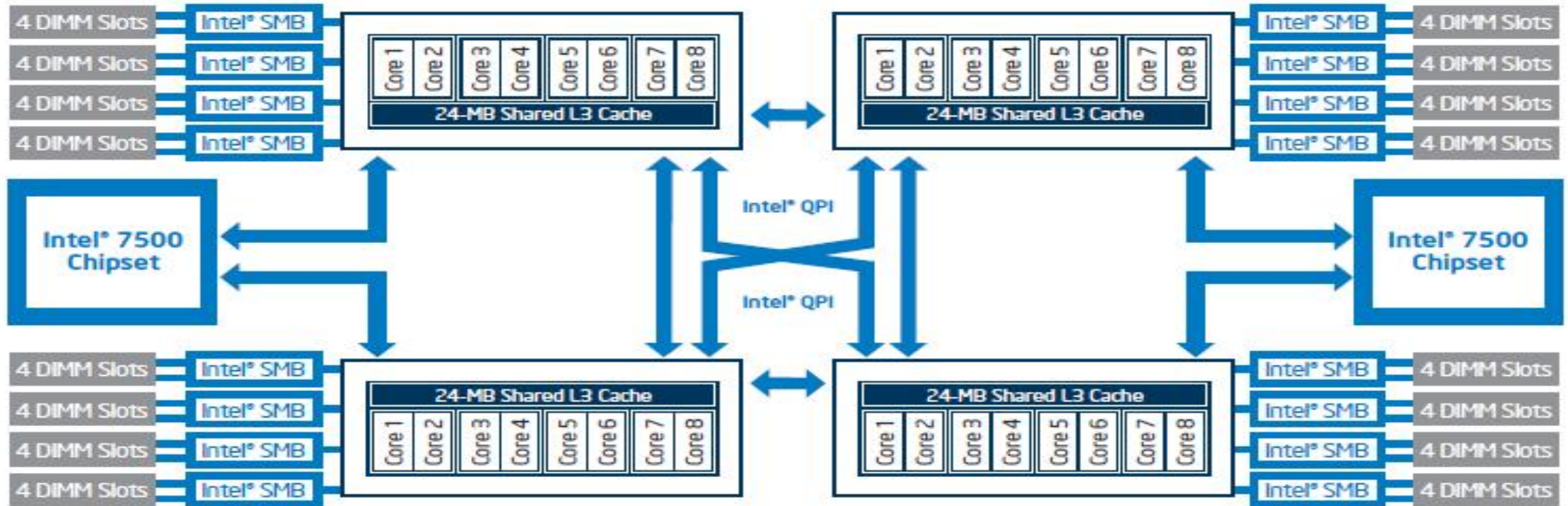
- Intel Nehalem





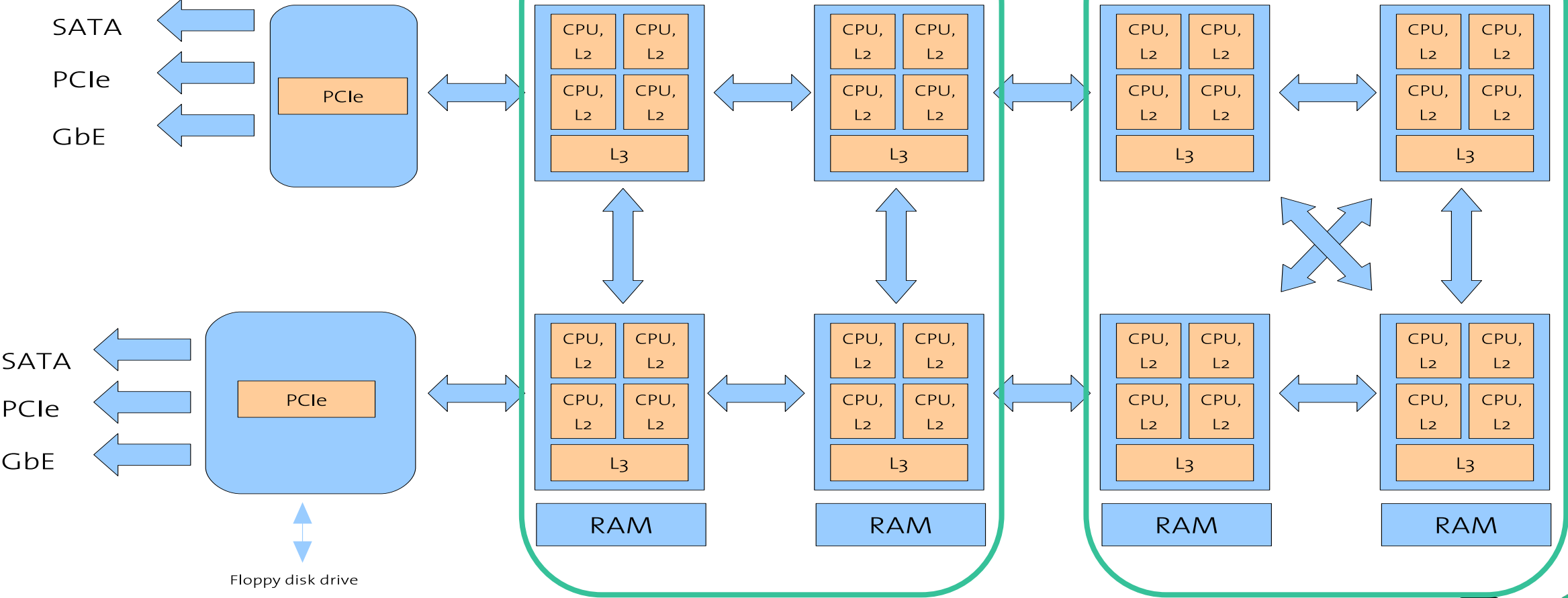
# Memory Locality and Caches

- NUMA (Non-Uniform Memory Access)

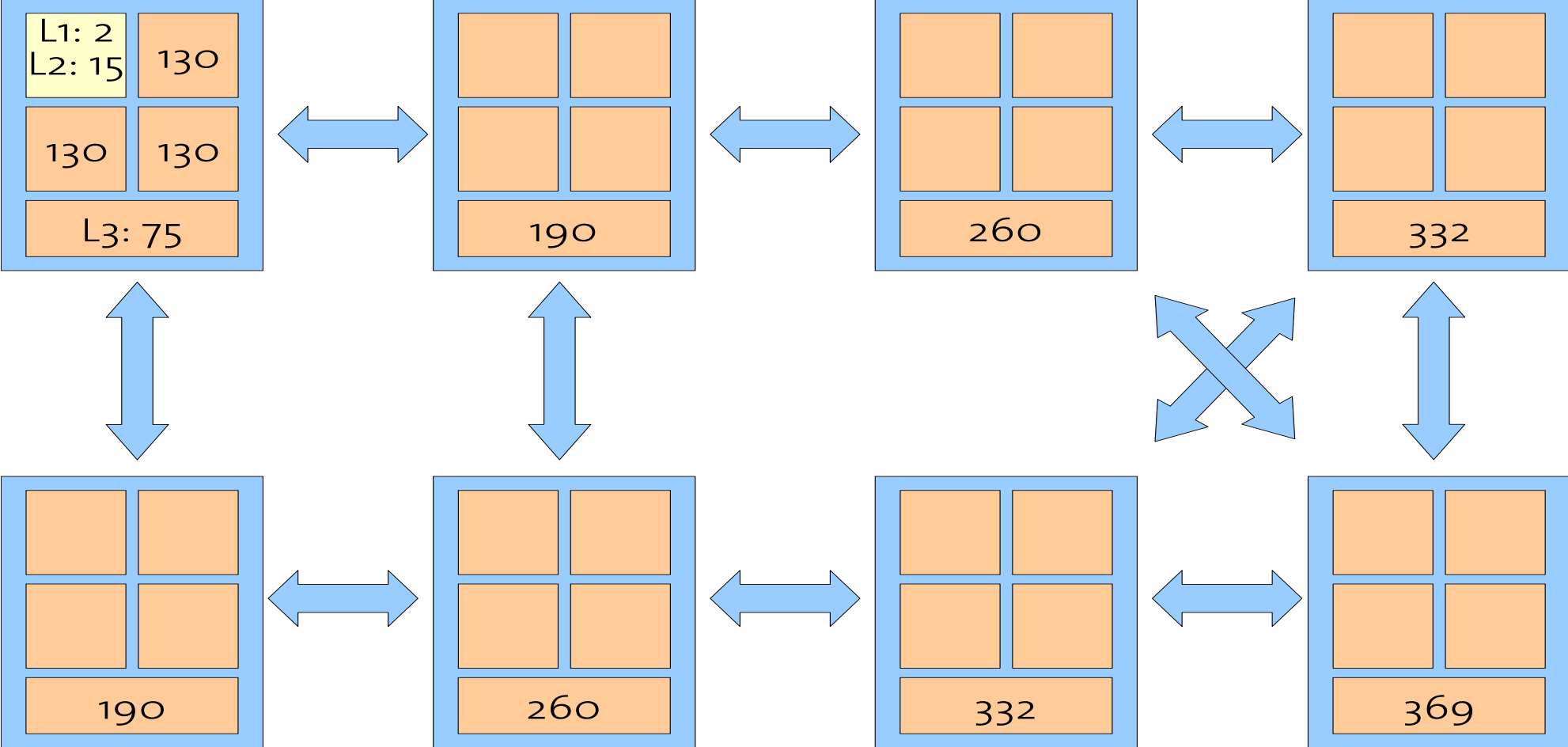


# Interconnect

- AMD Barcelona

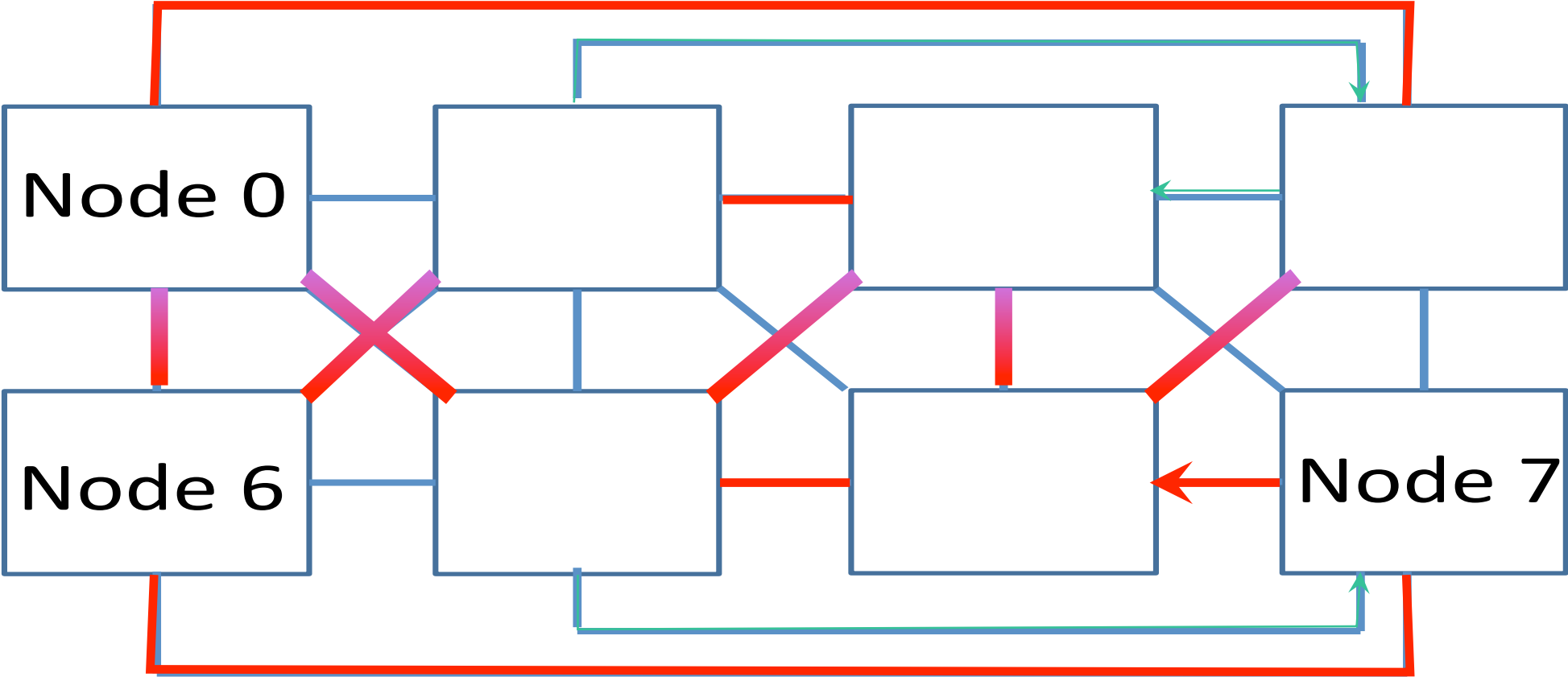


# Interconnect (Latency)





# Interconnect (Bandwidth)

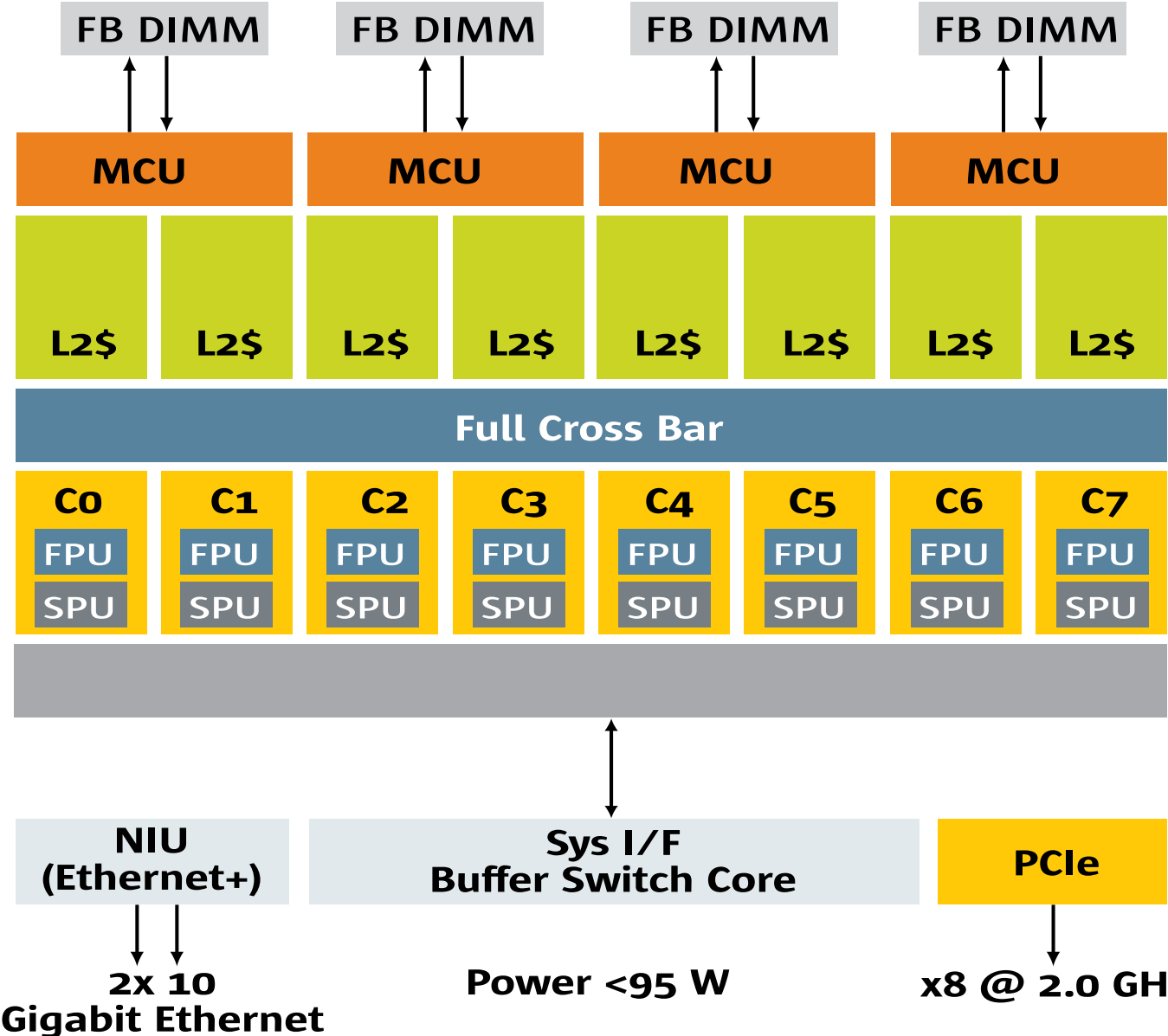


— 3GB/s      — 6GB/s      — 4GB/s-3GB/s      ← Unidirectional



# Interconnect

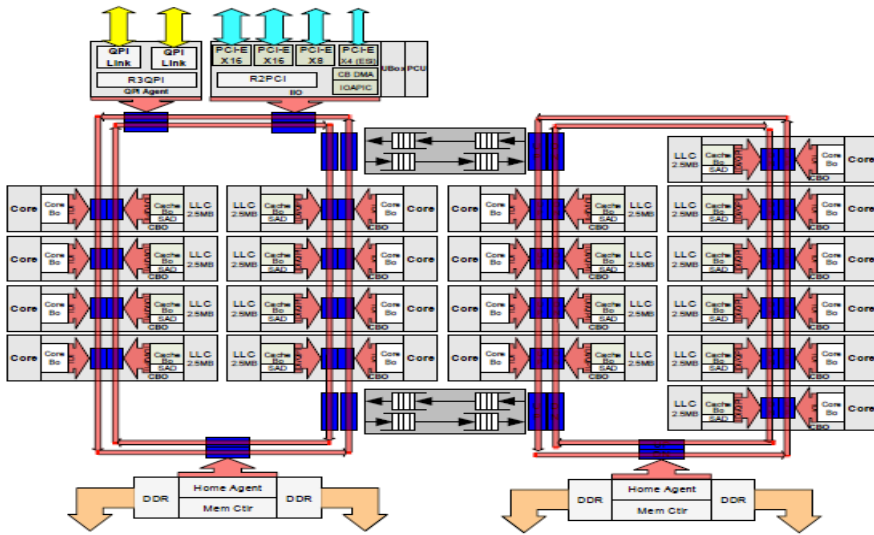
- Oracle Sparc T2



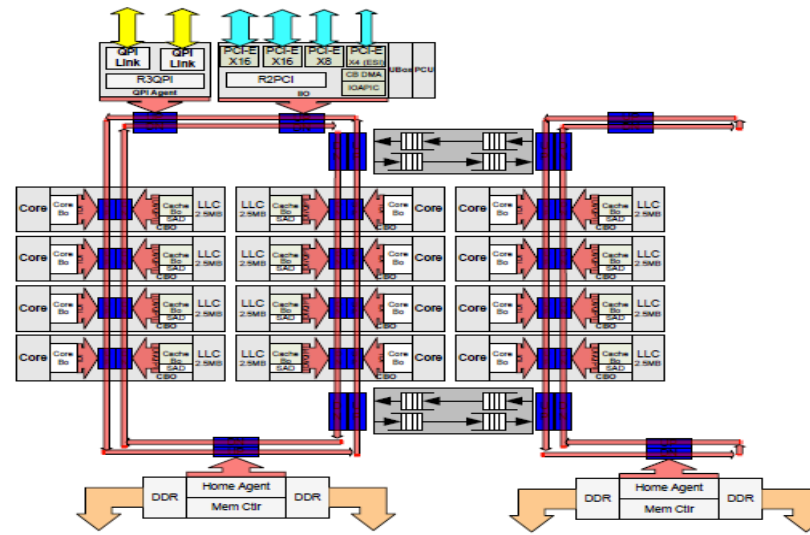
# Interconnect

## Haswell EP Die Configurations

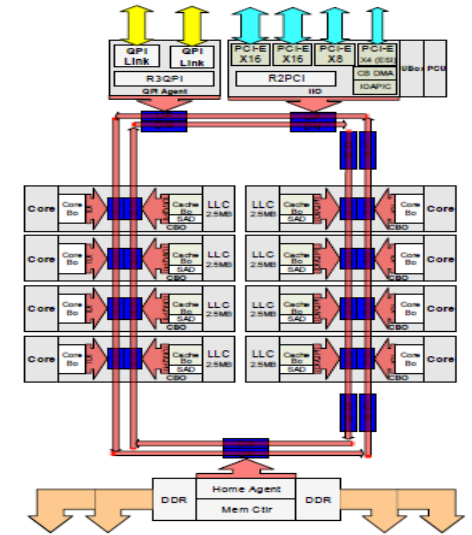
14-18 Core (HCC)



10-12 Core (MCC)



4-8 Core (LCC)



Not representative of actual die-sizes, orientation and layouts – for informational use only.

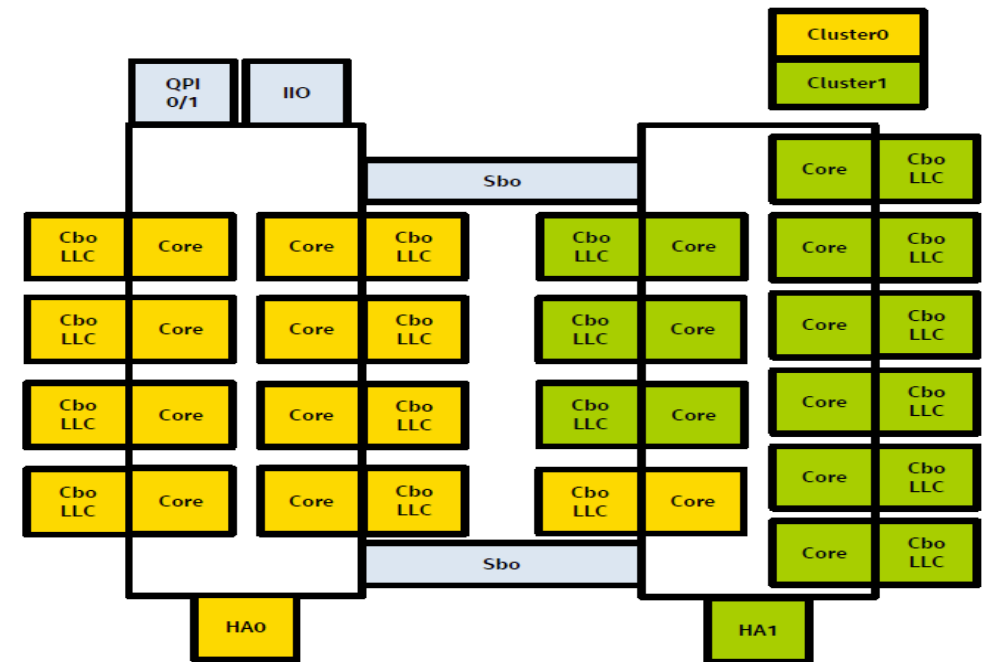
Chop	Columns	Home Agents	Cores	Power (W)	Transitors (B)	Die Area (mm <sup>2</sup> )
HCC	4	2	14-18	110-145	5.69	662
MCC	3	2	6-12	65-160	3.84	492
LCC	2	1	4-8	55-140	2.60	354

# Interconnect/Structure/Memory

## Cluster on Die (COD) Mode

- Supported on 1S & 2S SKUs with 2 Home Agents (10+ cores)
- In memory directory bits & directory cache used on 2S to reduce coherence traffic and cache-to-cache transfer latencies
- Targeted at NUMA optimized workloads where latency is more important than sharing across Caching Agents
  - Reduces average LLC hit and local memory latencies
  - HA sees most requests from reduced set of threads potentially offering higher effective memory bandwidth
- OS/VMM own NUMA and process affinity decisions

COD Mode for 18C E5-2600 v3

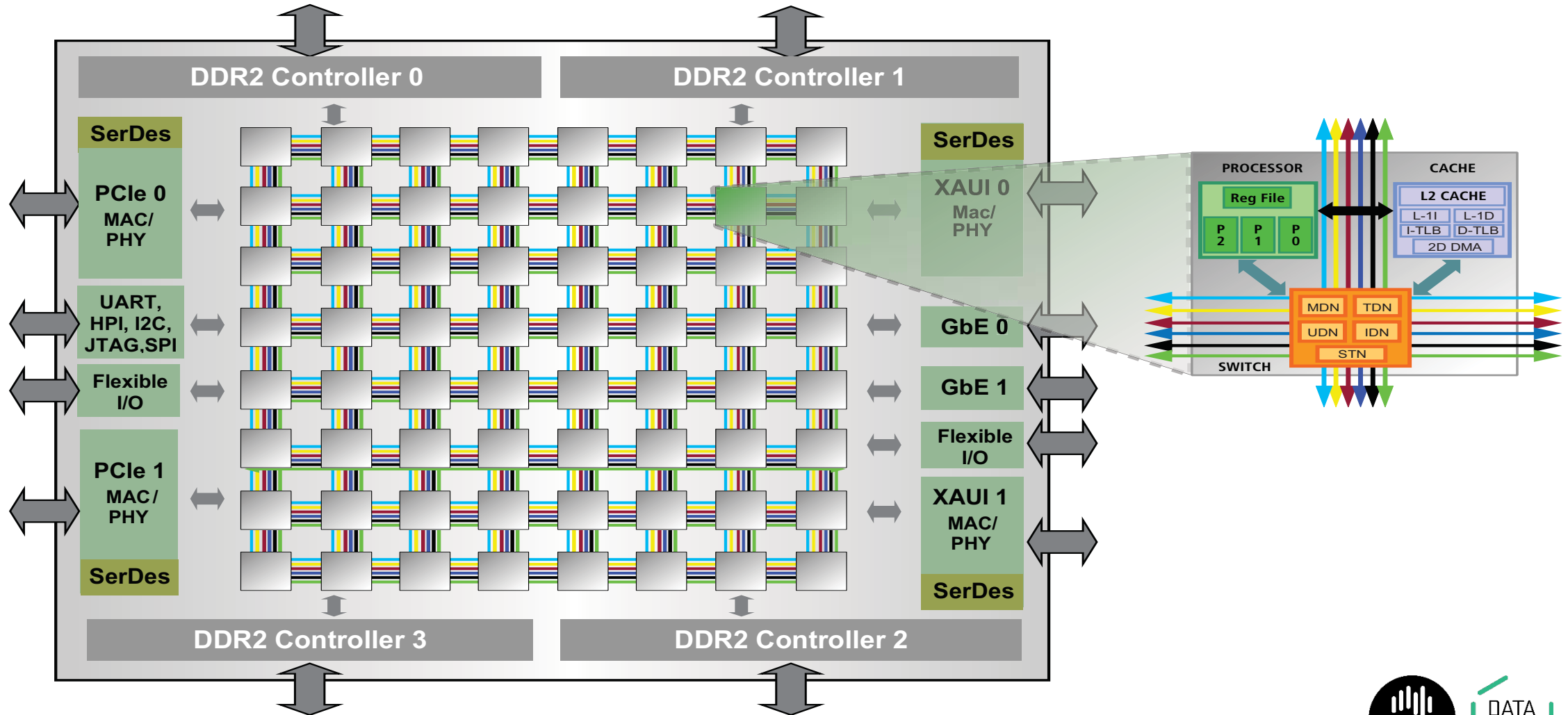


# Experimental/Future/Non-mainstream Multiprocessor Hardware

- Microsoft Beehive
  - Ring bus, no cache coherence
- Tiler (now Mellanox) Tile64, Tile-Gx
  - 100 cores, mesh network
- Intel Polaris
  - 80 cores, mesh network
- Intel SCC
  - 48 cores, mesh network, no cache coherency
- Intel MIC (Multi Integrated Core)
  - Knight's Corner/Landing - Xeon Phi
  - 60+ cores, ring bus/mesh

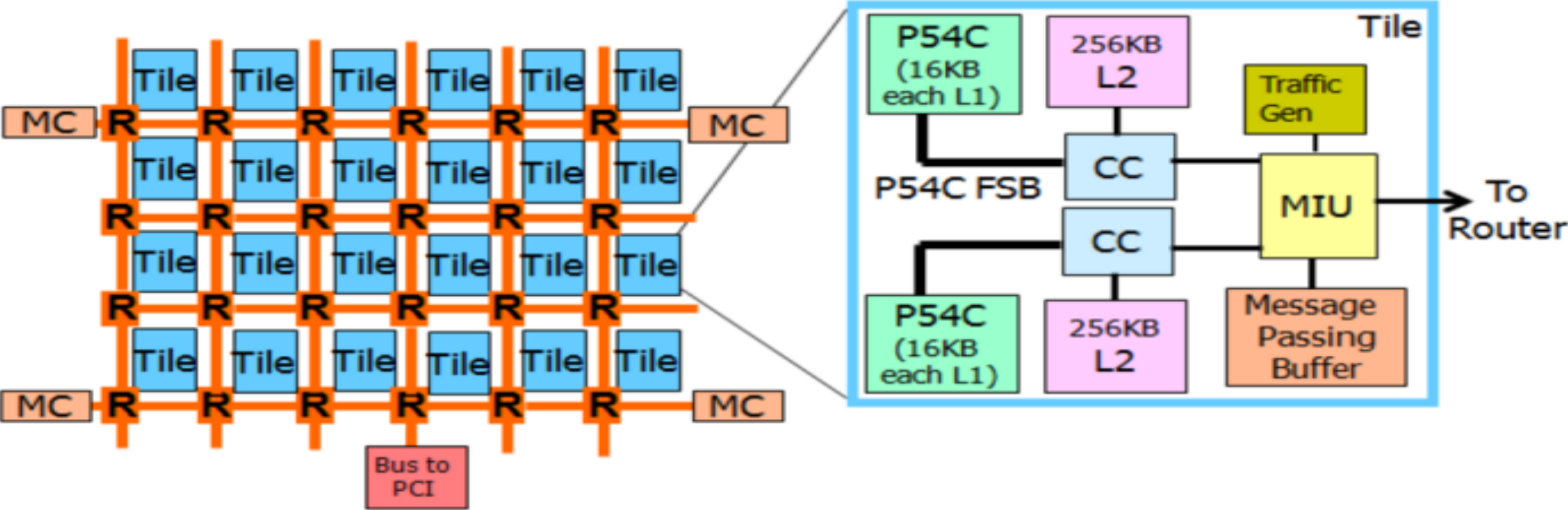
# Scale and Structure

- Tiler Tile64 (newest: Mellanox TILE-Gx), Intel Polariss



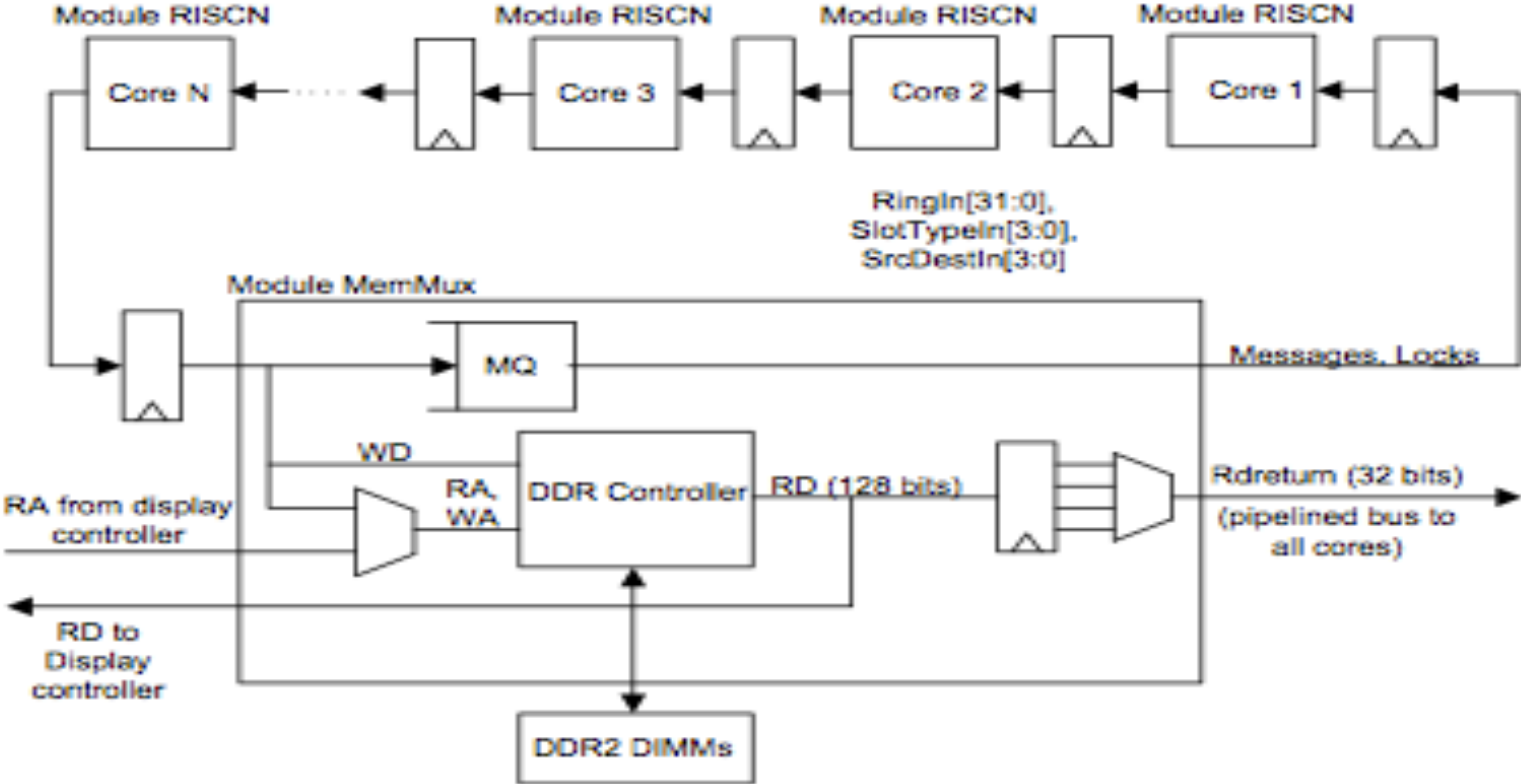
# Cache and Memory and IPC

- Intel SCC



# Interprocessor Communication

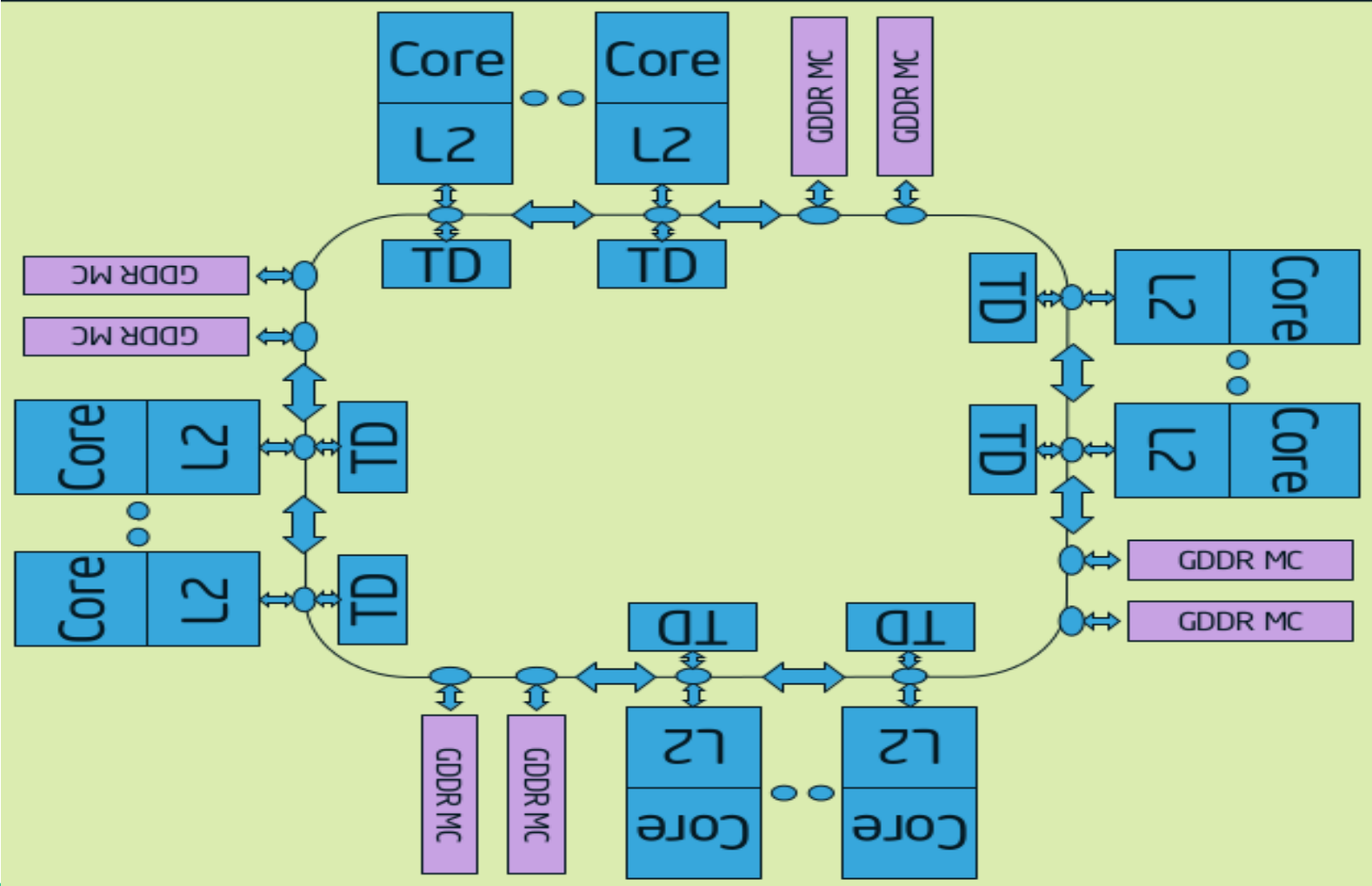
- Beehive



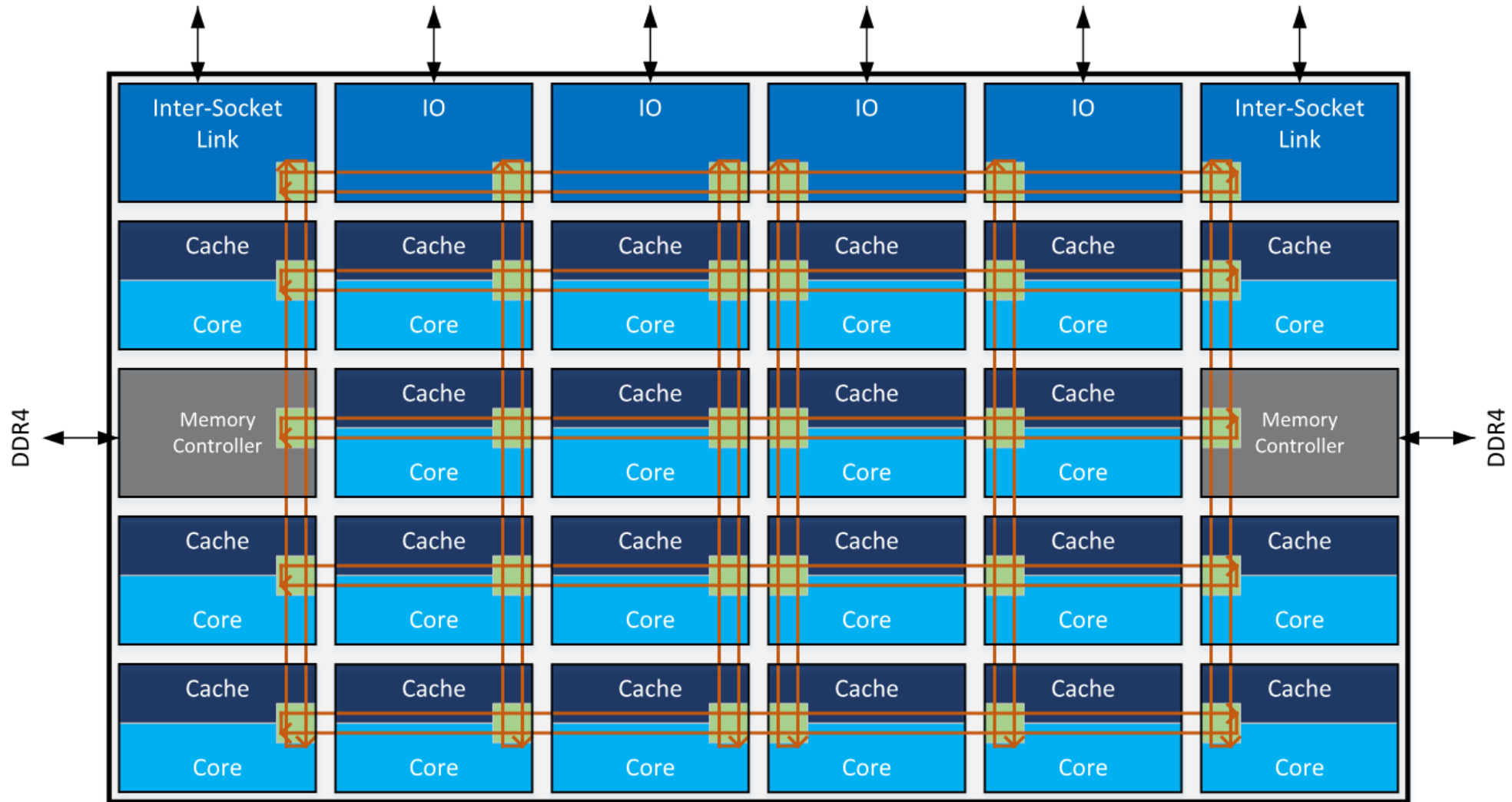


# Interconnect

- Intel MIC (Multi Integrated Core) (Knight's Corner/Landing - Xeon Phi)



# Skylake SP



# 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
    - 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

# 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)
  - Sometimes it's the only option
- Tradeoff: parallelism vs synchronisation
  - Synchronisation introduces serialisation
  - Make concurrent threads independent: reduce crit sections & cache misses
- 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!

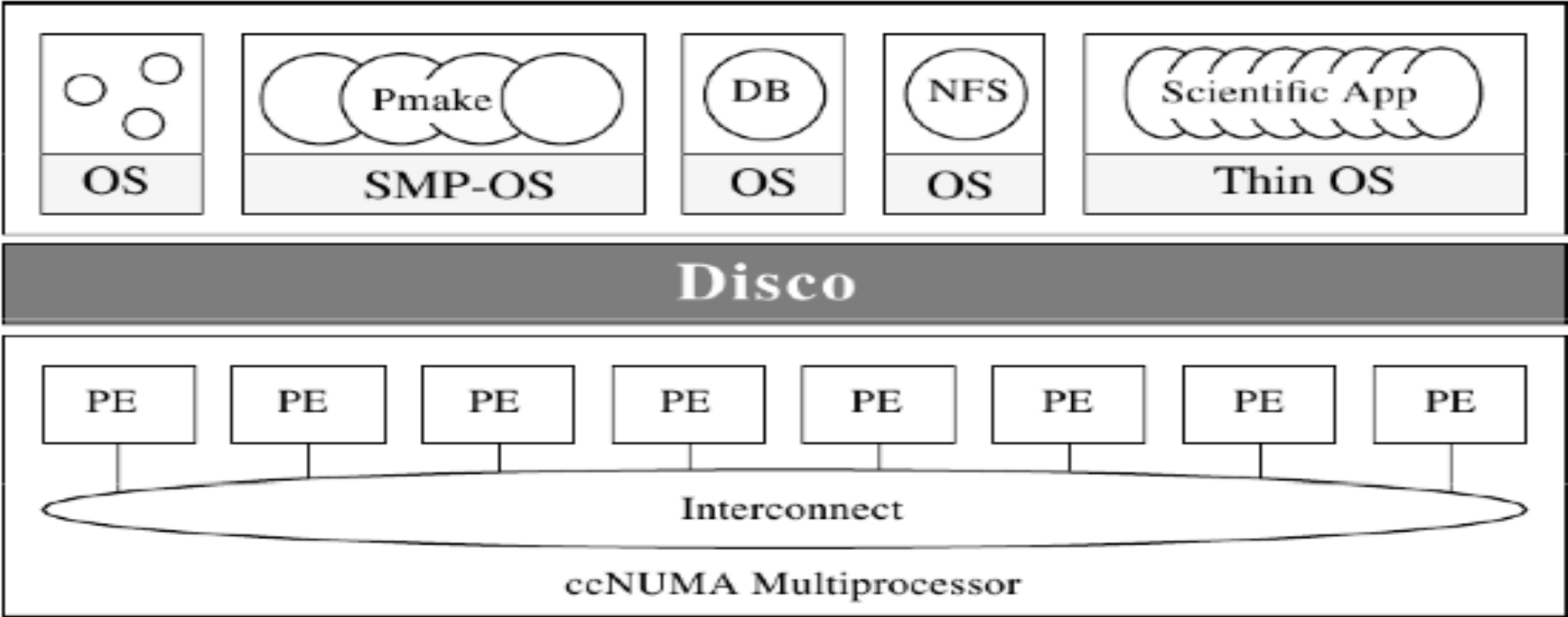


# Divide and Conquer

## Disco

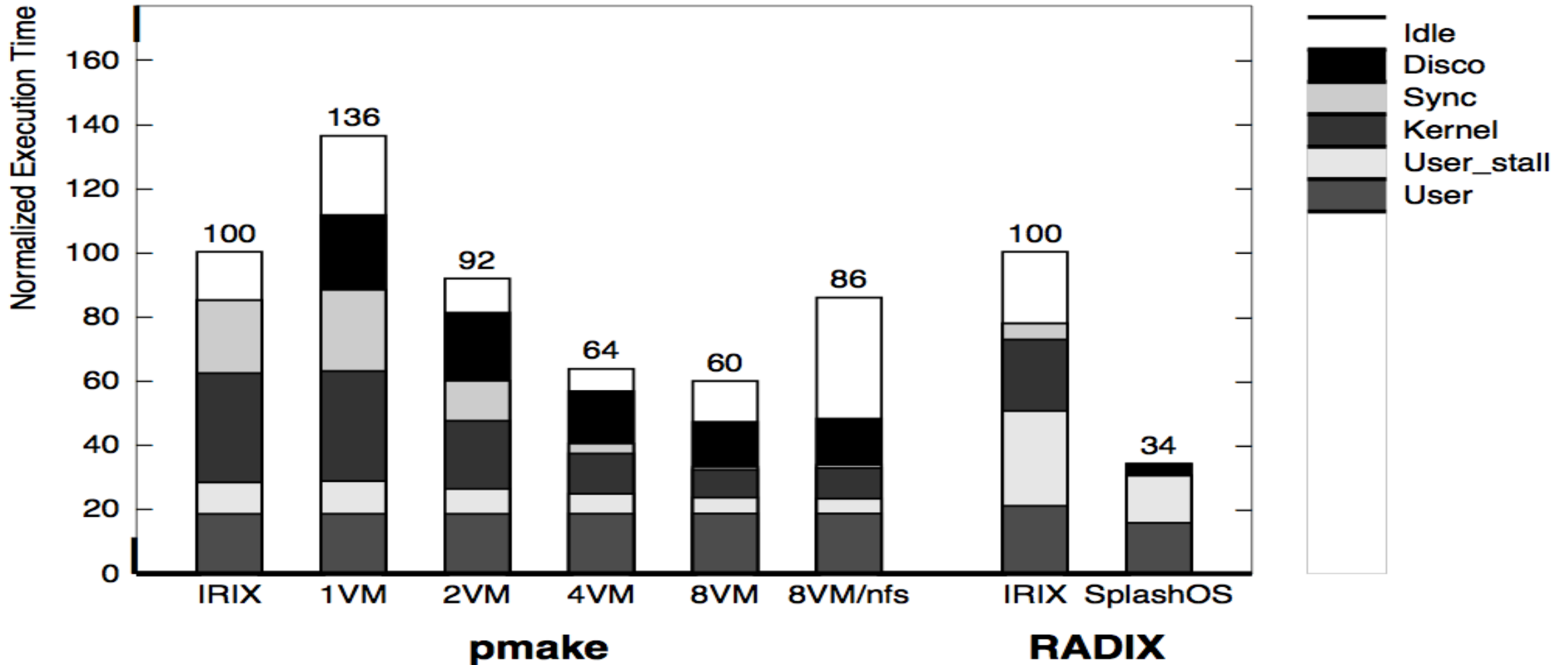
- Scalability is too hard!
- Context:
  - ca. 1995, large ccNUMA multiprocessors appearing
  - Scaling OSES requires extensive modifications
- Idea:
  - Implement a scalable VMM
  - Run multiple OS instances
- VMM has most of the features of a scalable OS:
  - NUMA aware allocator
  - Page replication, remapping, etc.
- VMM substantially simpler/cheaper to implement
- Modern incarnations of this
  - Virtual servers (Amazon, etc.)
  - Research (Cerberus)

# Disco Architecture



[Bugnion et al., 1997]

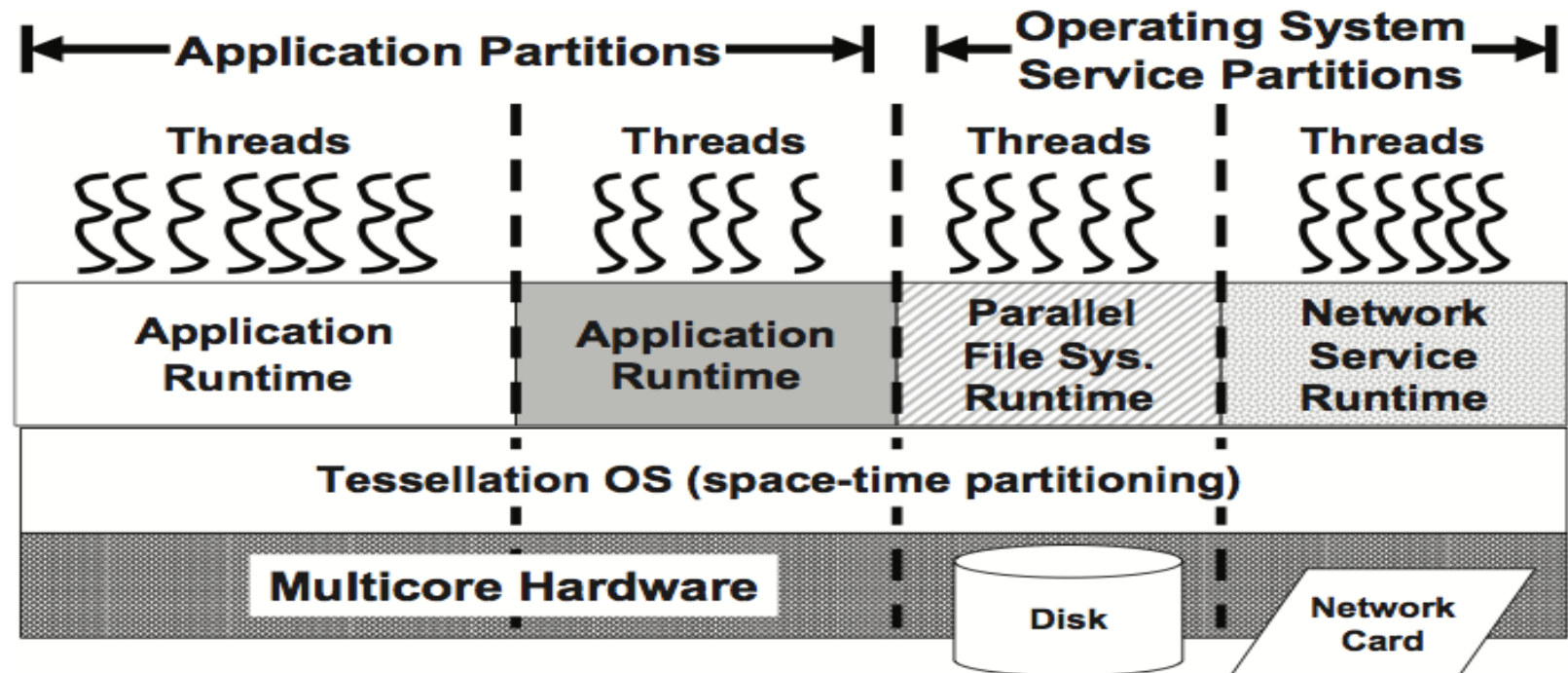
# Disco Performance



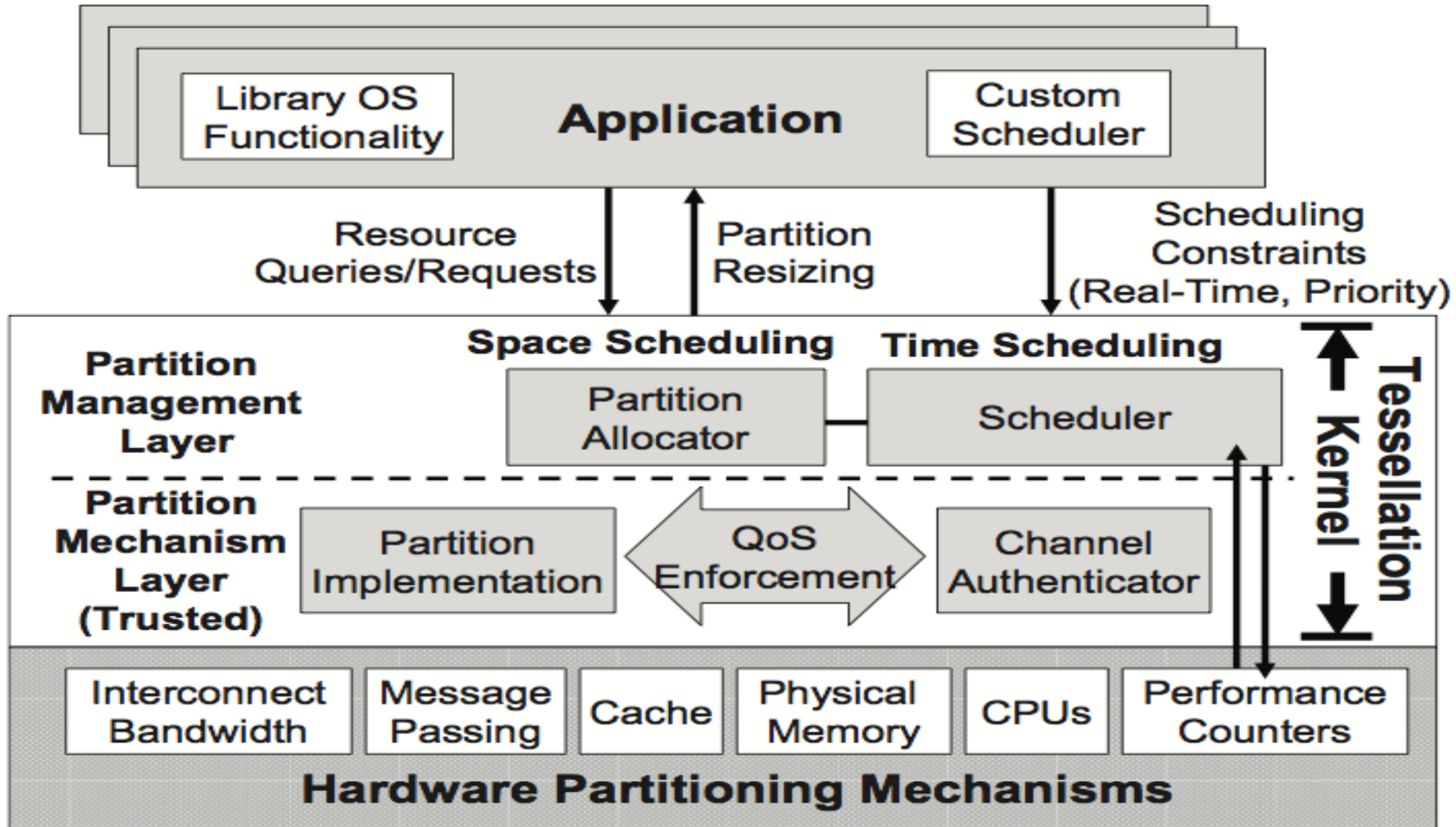
# Space-Time Partitioning

## Tessellation

- Space-Time partitioning
- 2-level scheduling
- Context:
  - 2009-... highly parallel multicore systems
  - Berkeley Par Lab



# Tessellation



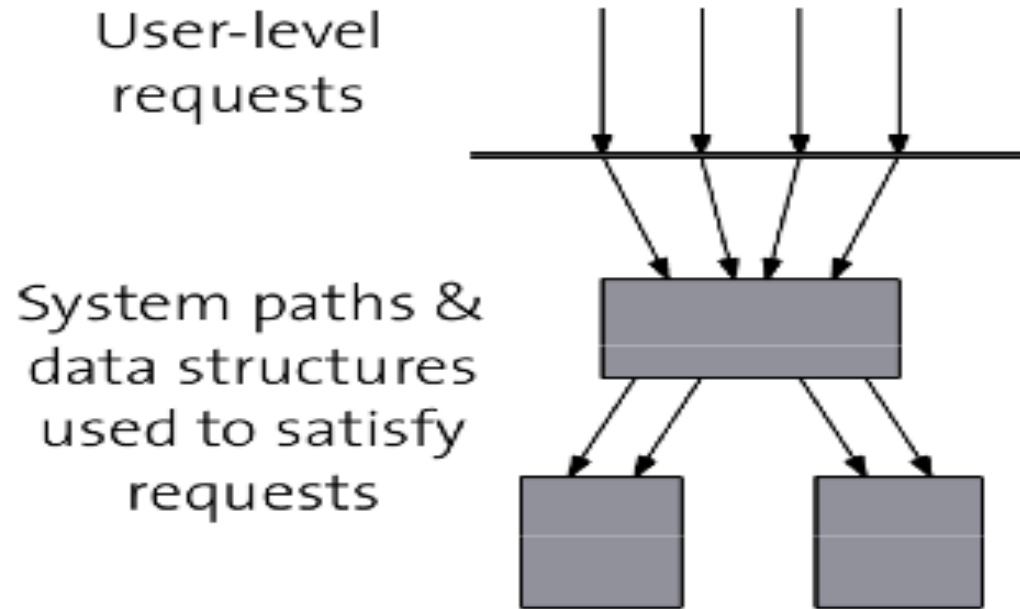
# Reduce Sharing

## 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

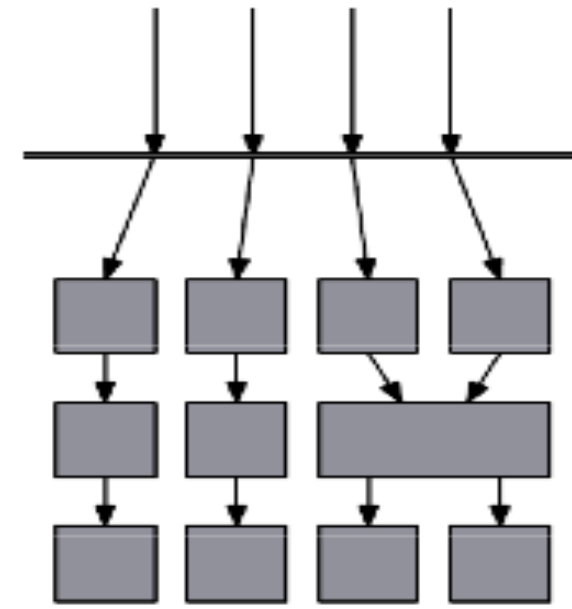
# K42: Fine-grained objects

Traditional System



- much sharing

OO Decomposed System

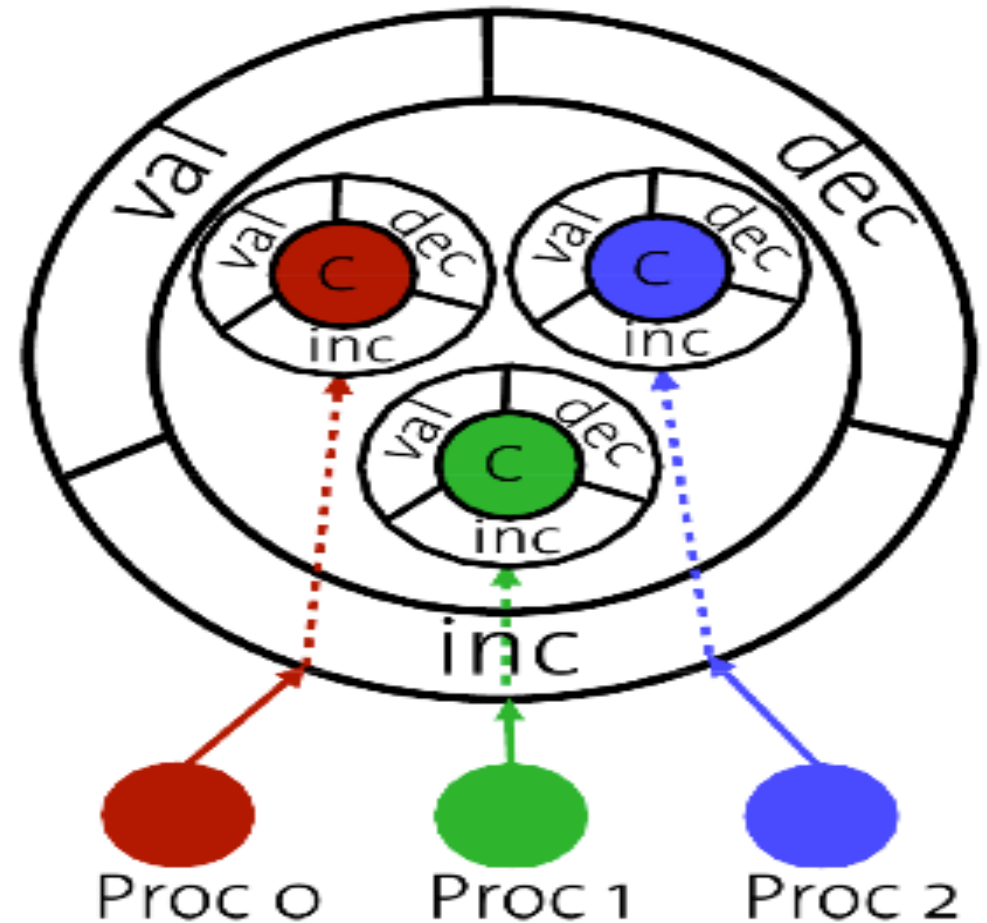


- much **less** sharing
- better performance

[Appavoo, 2005]

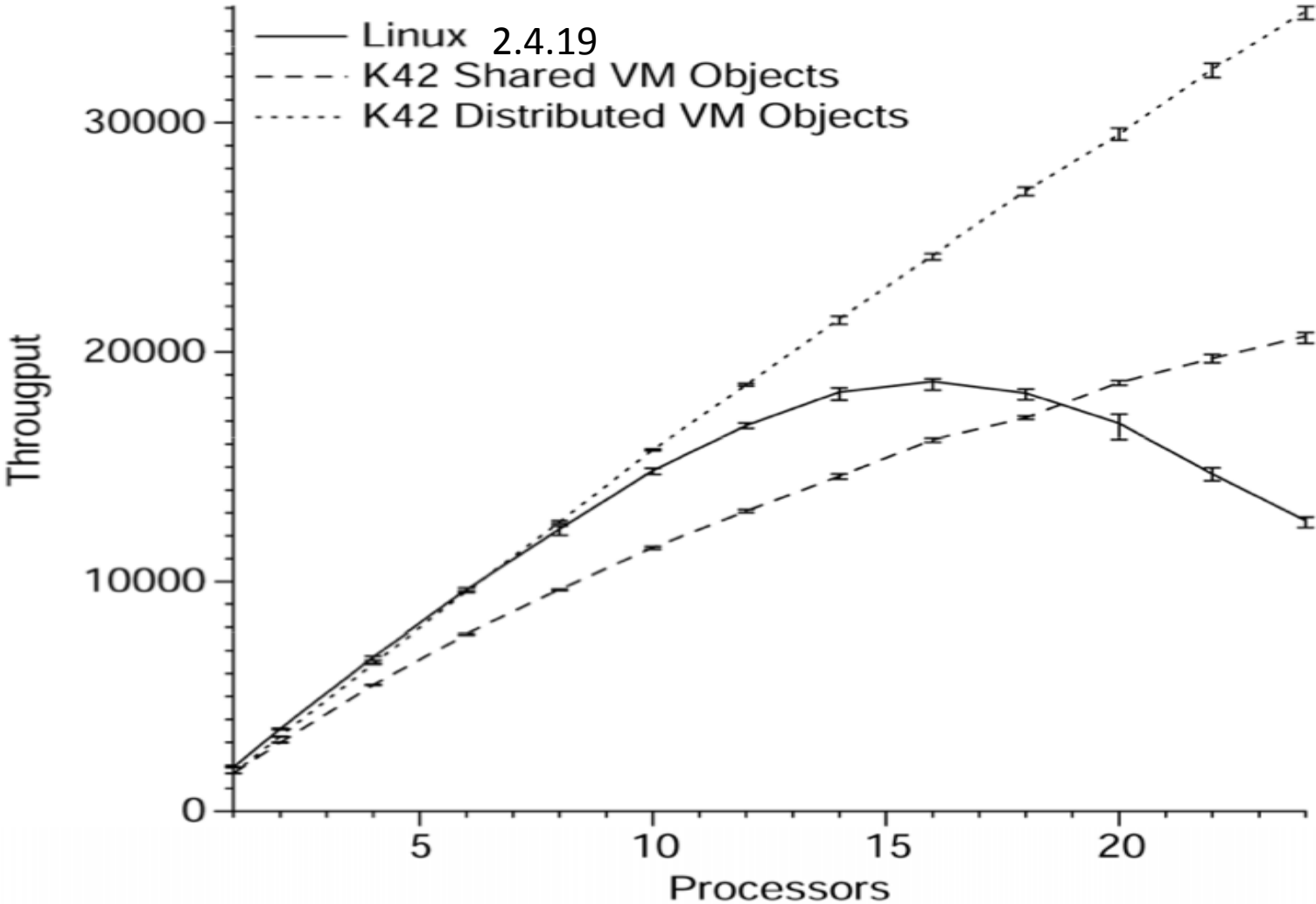
# K42: Clustered objects

- Globally valid object reference
- Resolves to
  - Processor local representative
- Sharing, locking strategy local to each object
- Transparency
  - Eases complexity
  - Controlled introduction of locality
- Shared counter:
  - *inc, dec*: local access
  - *val*: communication
- Fast path:
  - Access mostly local structures





# K42 Performance



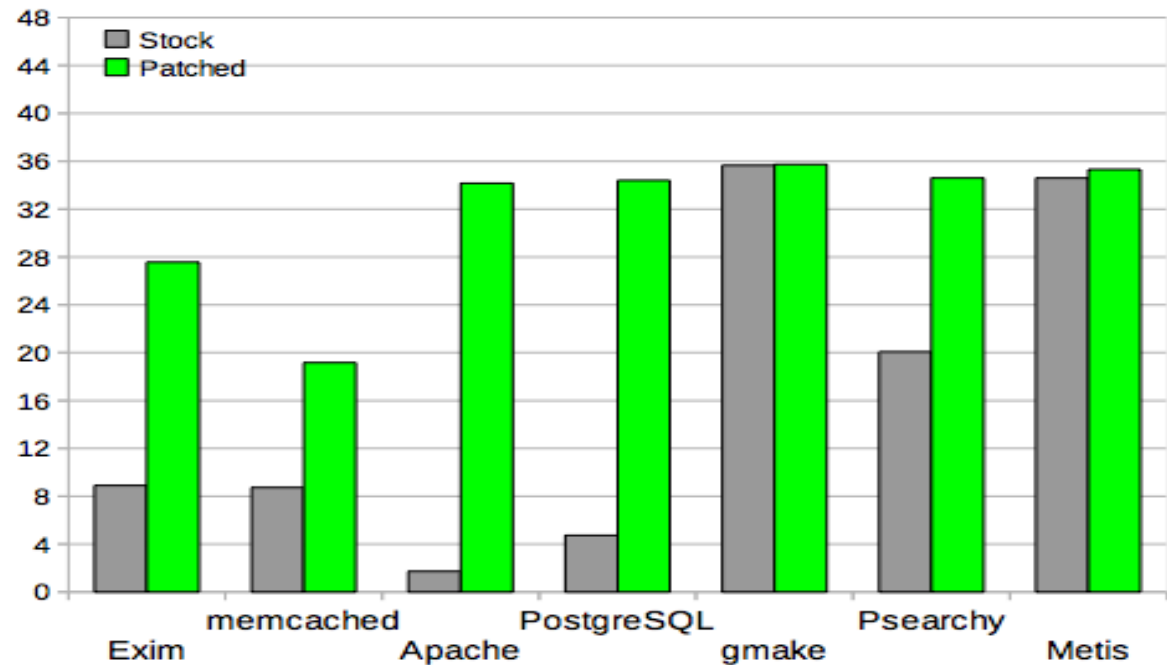
# Corey

- 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.



# 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)

# 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.

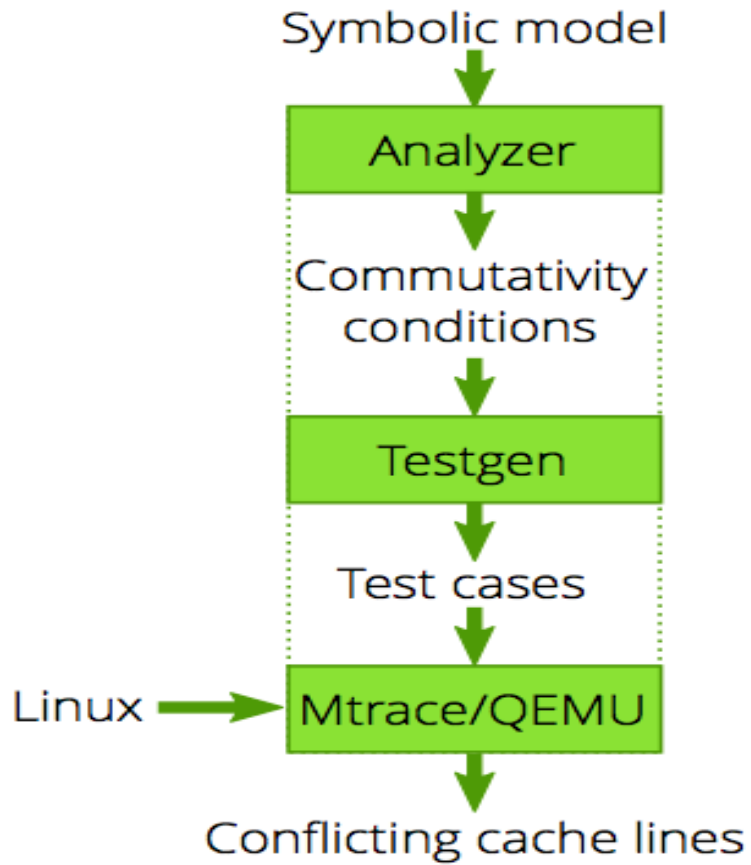
# Scalability of the API

- 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

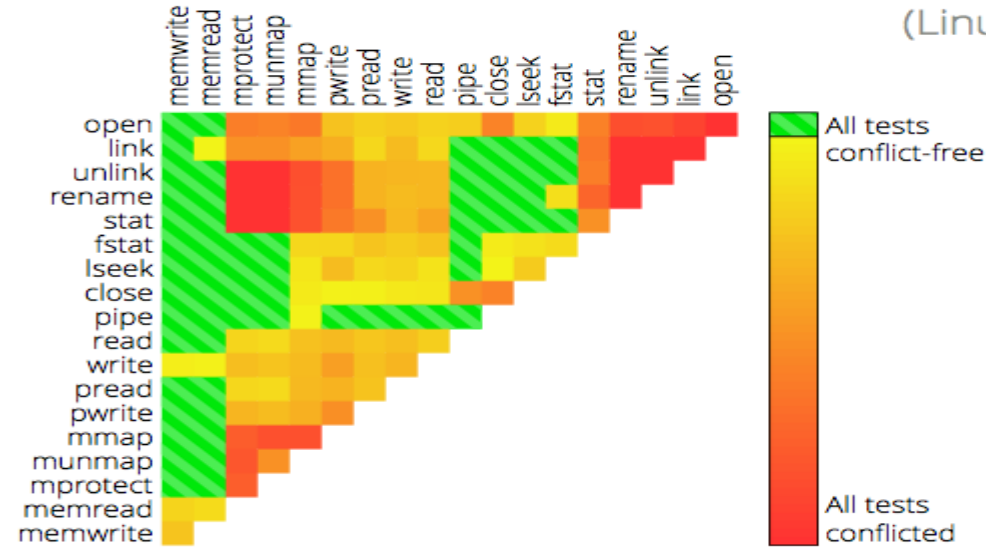
# Scalable Commutativity Rule

- 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  $\Rightarrow$  communication is unnecessary
  - without communication, no conflicts
- Informs software design process
  - Design: design guideline for scalable interfaces
  - Implementation: clear target
  - Test: workload-independent testing

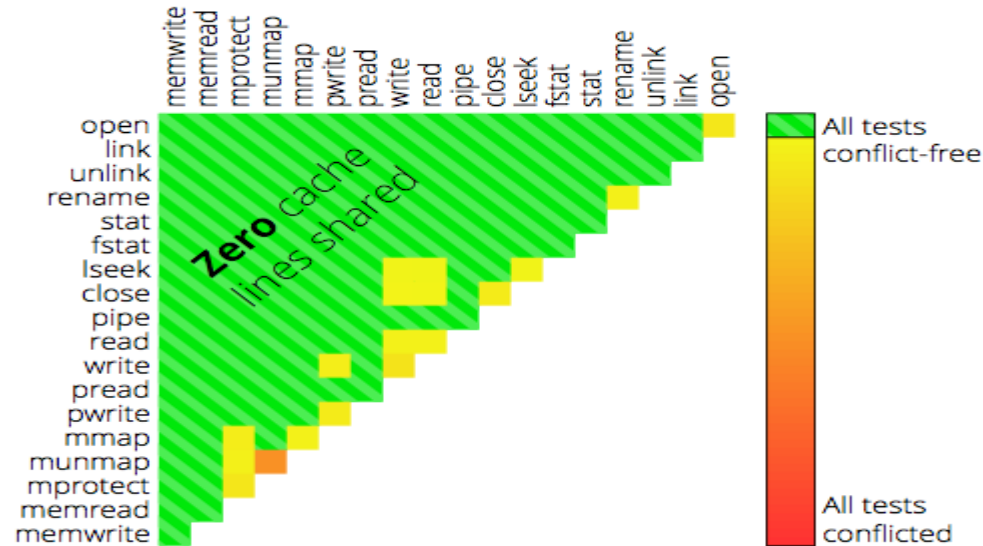
# Commuter: An Automated Scalability Testing Tool



(Linux 3.8, ramfs)

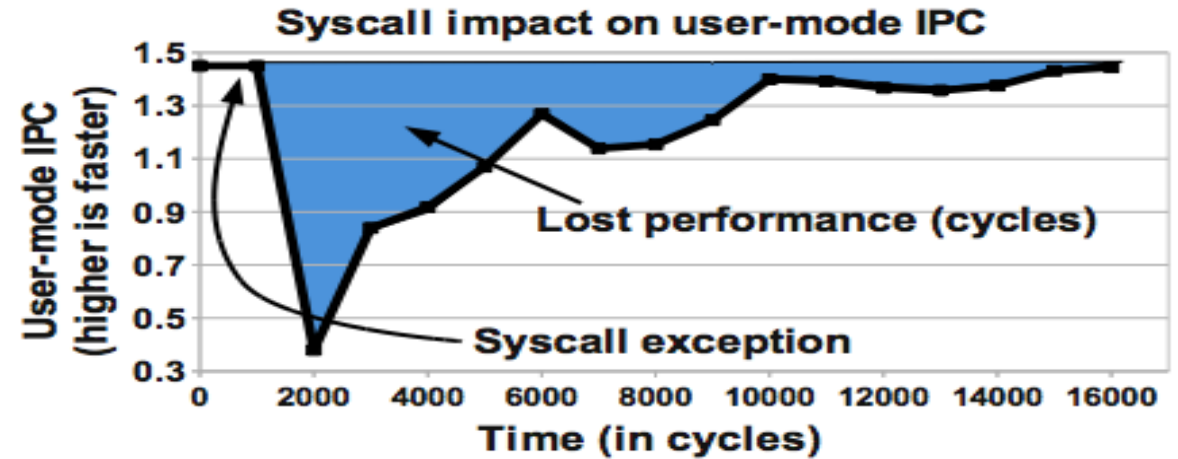


(sv6)



# FlexSC

- Context:
  - 2010, commodity multicores
  - U Toronto
- Goal:
  - Reduce context switch overhead of system calls
- Syscall context switch:
  - Usual mode switch overhead
  - But: cache and TLB pollution!

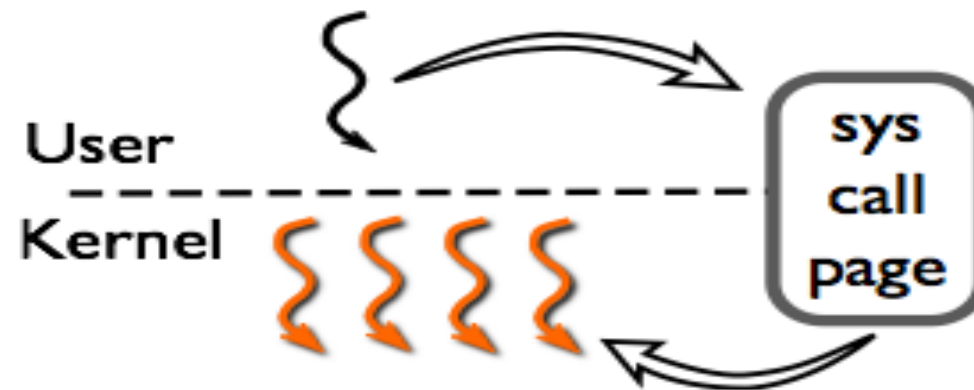


Syscall	Instructions	Cycles	IPC	i-cache	d-cache	L2	L3	d-TLB
stat	4972	13585	0.37	32	186	660	2559	21
pread	3739	12300	0.30	32	294	679	2160	20
pwrite	5689	31285	0.18	50	373	985	3160	44
open+close	6631	19162	0.34	47	240	900	3534	28
mmap+munmap	8977	19079	0.47	41	233	869	3913	7
open+write+close	9921	32815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

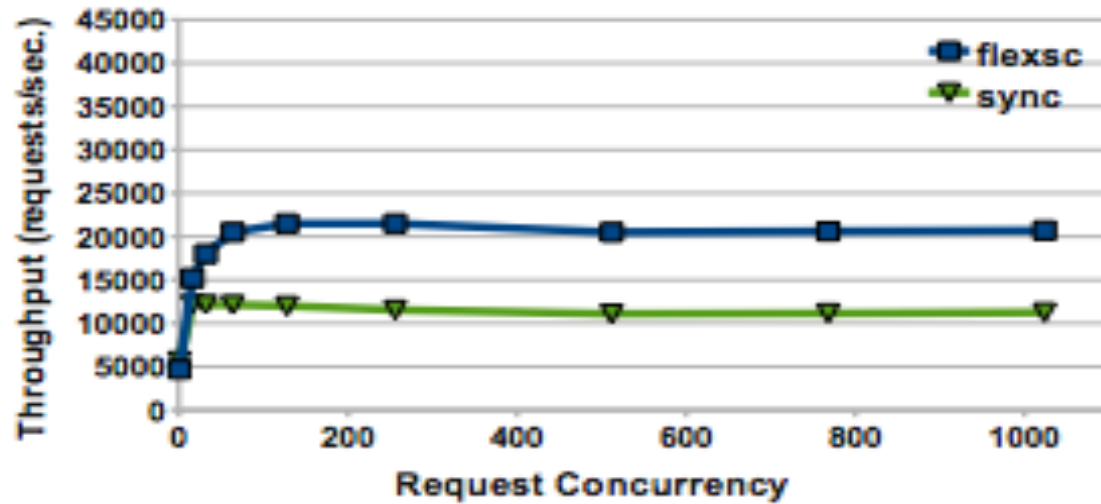


# FlexSC

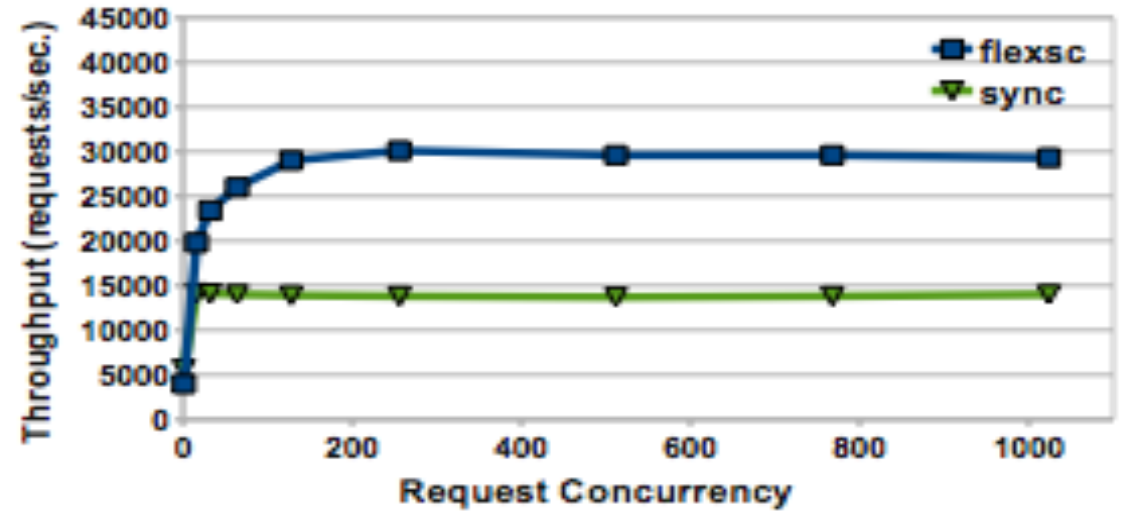
- Asynchronous system calls
  - Batch system calls
  - Run them on dedicated cores
- FlexSC-Threads
  - M on N
  - $M \gg N$



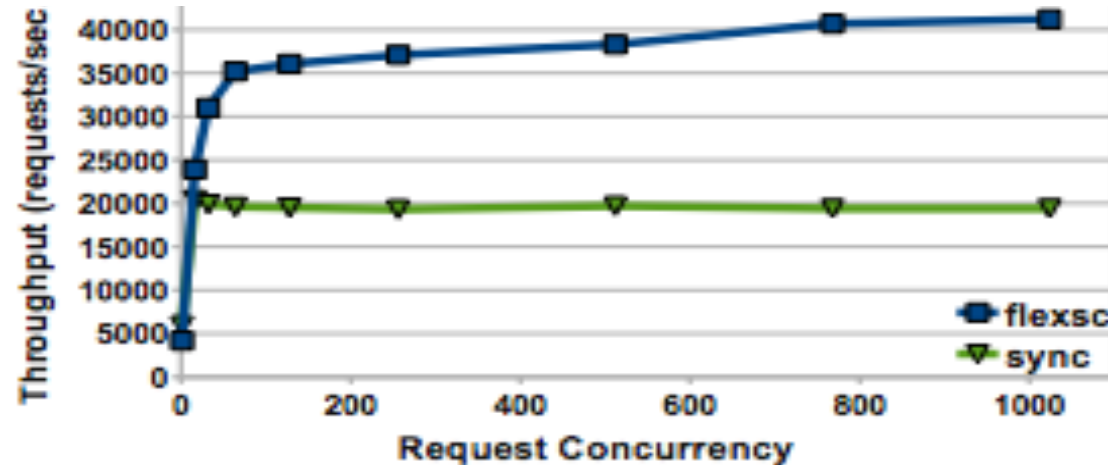
# FlexSC Results



(a) 1 Core



(b) 2 Cores

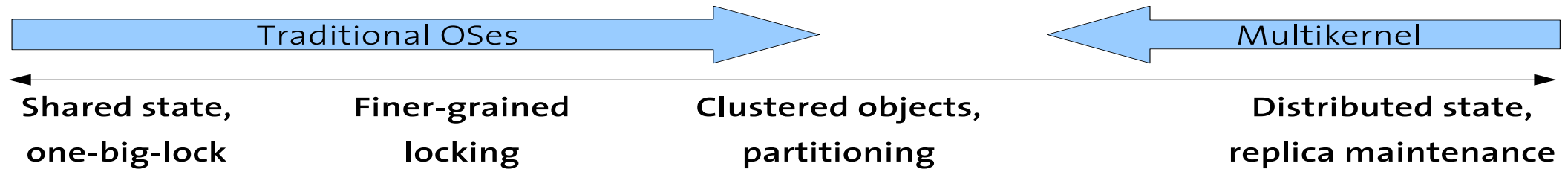


(c) 4 Cores

Apache  
FlexSC: batching,  
sys call core redirect

# No sharing

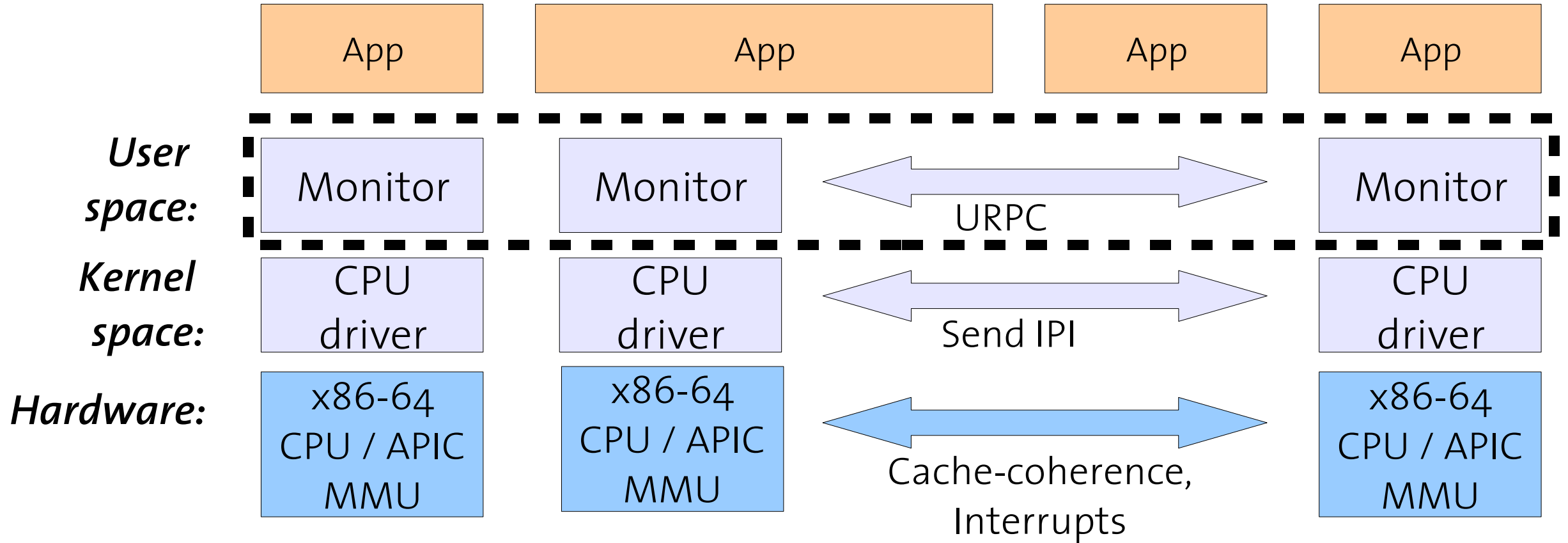
- Multikernel
  - Barrelfish
  - fos: factored operating system



# 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

# Barrelfish

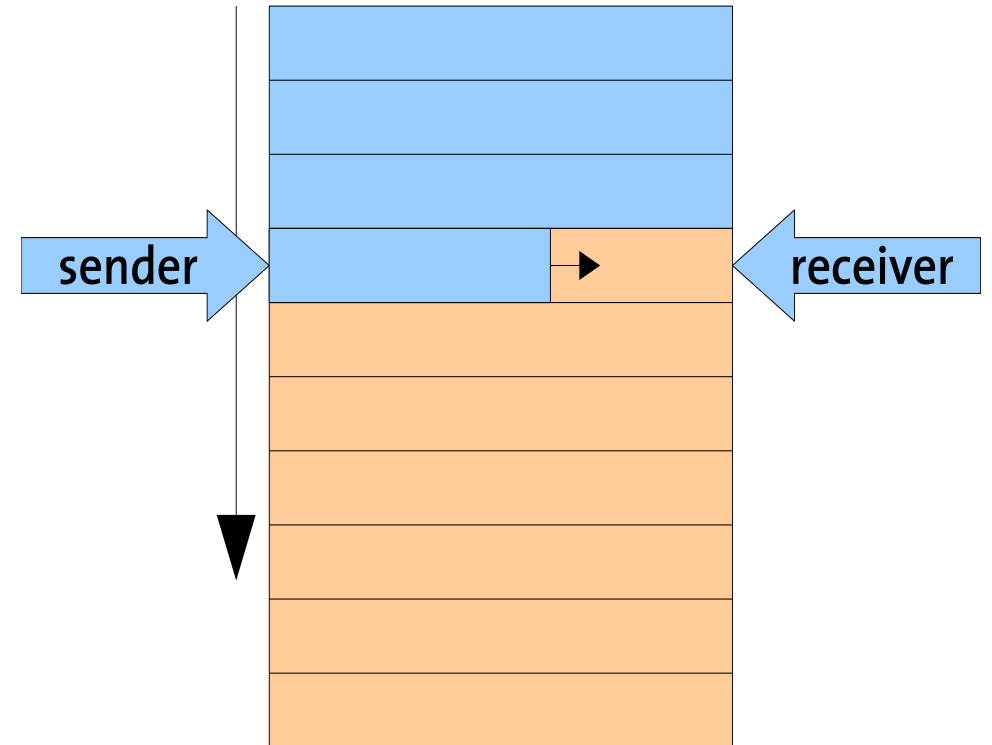


# Barrelfish: Replication

- 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

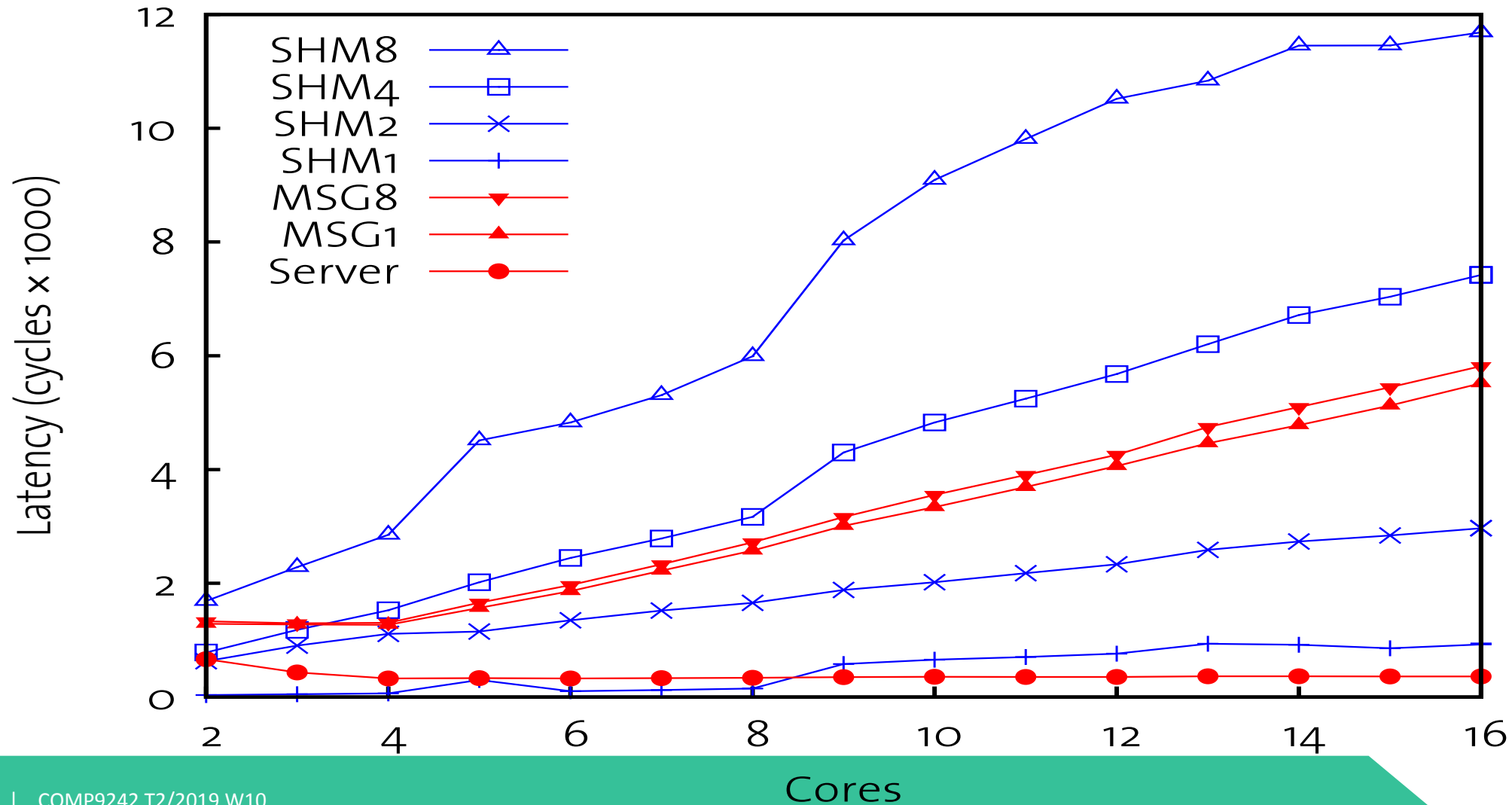
# Barrelfish: Communication

- Different mechanisms:
  - Intra-core
    - Kernel endpoints
  - Inter-core
    - URPC
- URPC
  - Uses cache coherence + polling
  - Shared buffer
    - 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

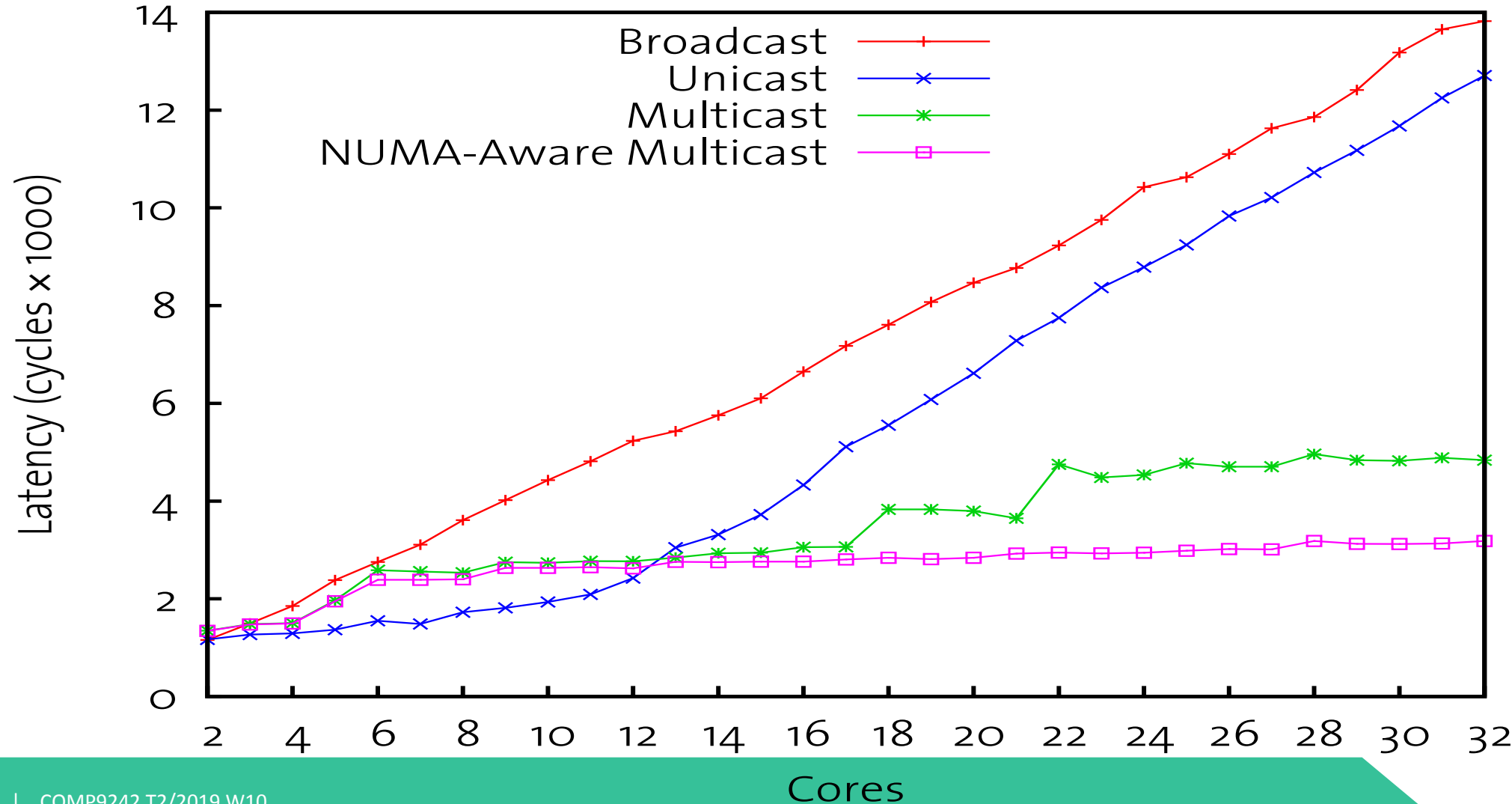
- Message passing vs caching





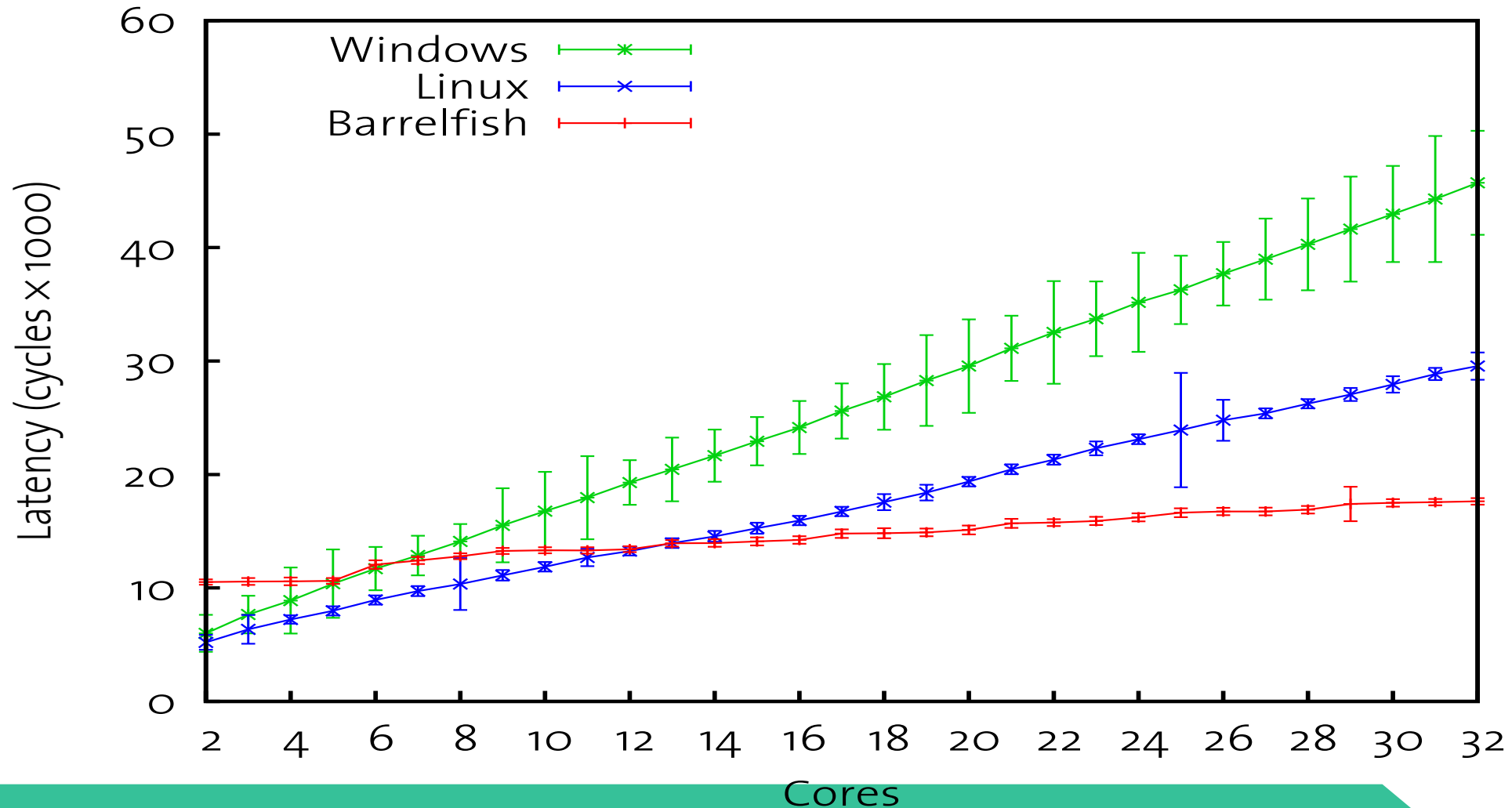
# Barrelfish: Results

- Broadcast vs Multicast



# Barrelfish: Results

- TLB shutdown



# 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)