Systems@ETH Zürich

Week 7: Multiprocessing

(263-3800-00L)

Timothy Roscoe

Herbstsemester 2016

http:// www.systems.ethz.ch/courses/fall2016/aos

Milestone 5: Another core

Systems@ETH zürich

- Assignment:
  - Bring up the second Cortex-A9 core
  - Distribute your OS.
- Today:
  - Multiprocessor operating systems
  - Inter-core messaging in Barrelfish
  - Cache coherency and weak memory

Systems@ETH zürich

THE EVOLUTION OF MULTIPROCESSORS

Multiprocessor OSes

Systems@ETH zürich

- Multiprocessor computers were anticipated by the research community long before they became mainstream
  - Typically restricted to “big iron”
- But relatively few OSes designed *from the outset* for multiprocessor hardware
- A multiprocessor OS:
  - Runs on a tightly-coupled (usually shared-memory) multiprocessor machine
  - Provides system-wide OS abstractions

Multics

Systems@ETH zürich

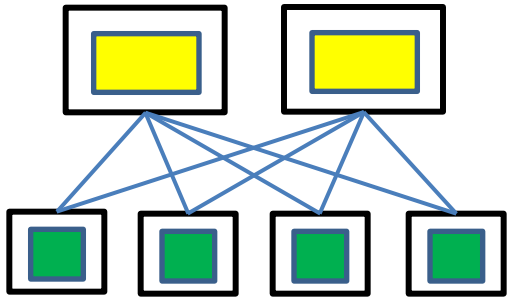
- Time-sharing operating system for a multiprocessor mainframe
- Joint project between MIT, General Electric, and Bell Labs (until 1969)
- 1965 – mid 1980s
  - Last Multics system decommissioned in 2000
- Goals: reliability, dynamic reconfiguration, security, etc.
- Very influential

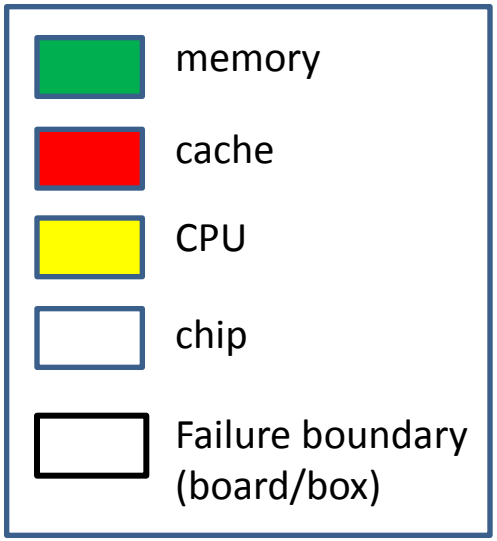
Multics: typical configuration

GE645 computer
Symmetric multiprocessor

Communication was by using "mailboxes" in the memory modules and corresponding interrupts (asynchronous).

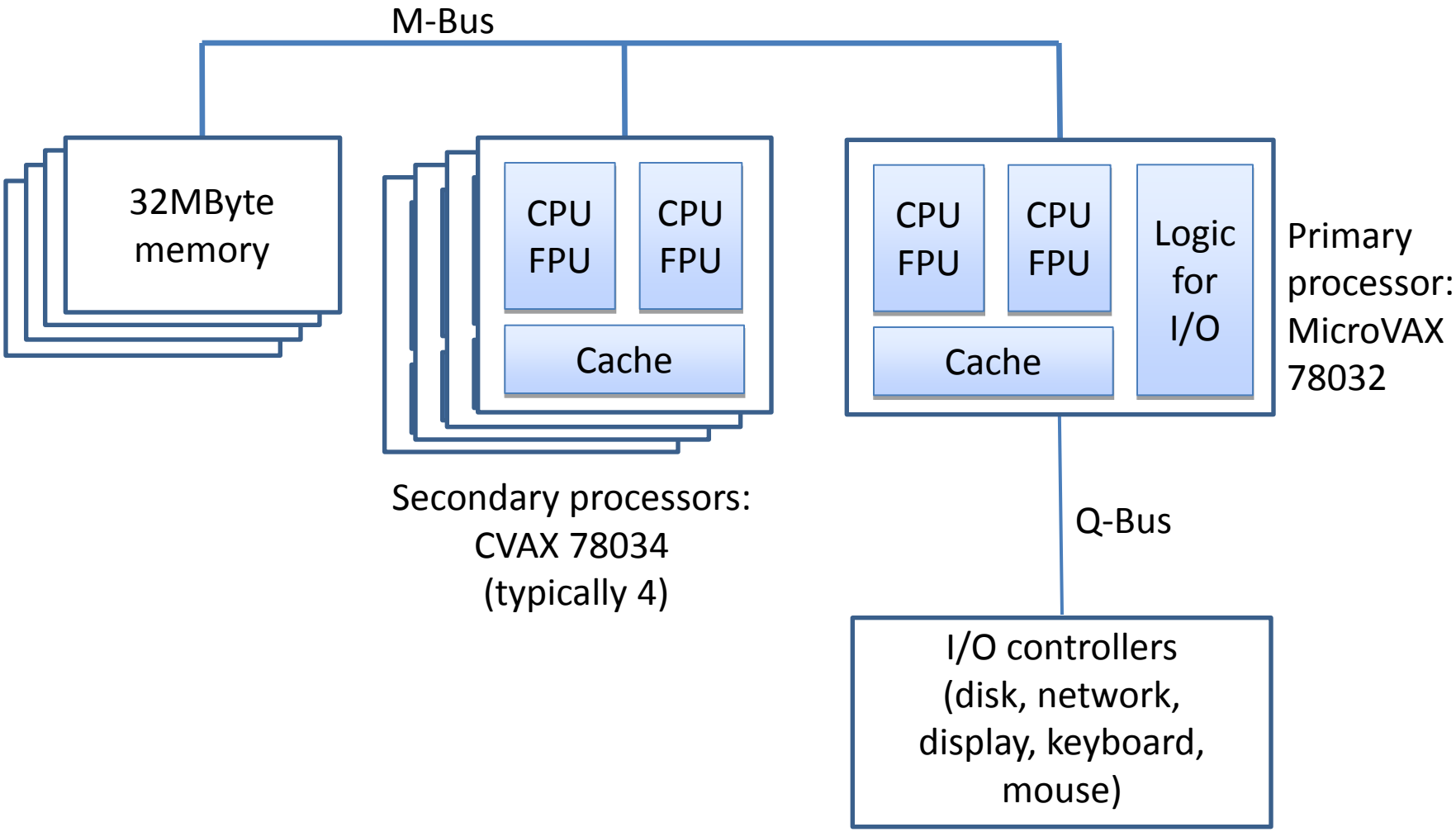
Multics on GE645



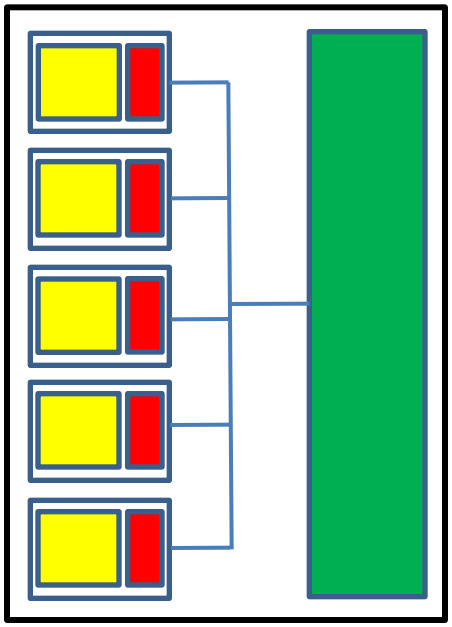


- Reliable interconnect
- No caches
- Single level of shared memory
  - Uniform memory access (UMA)
- Online reconfiguration of the hardware
  - Regularly partitioned into 2 separate systems for testing and development and then recombined
- Slow!

Firefly c. 1991

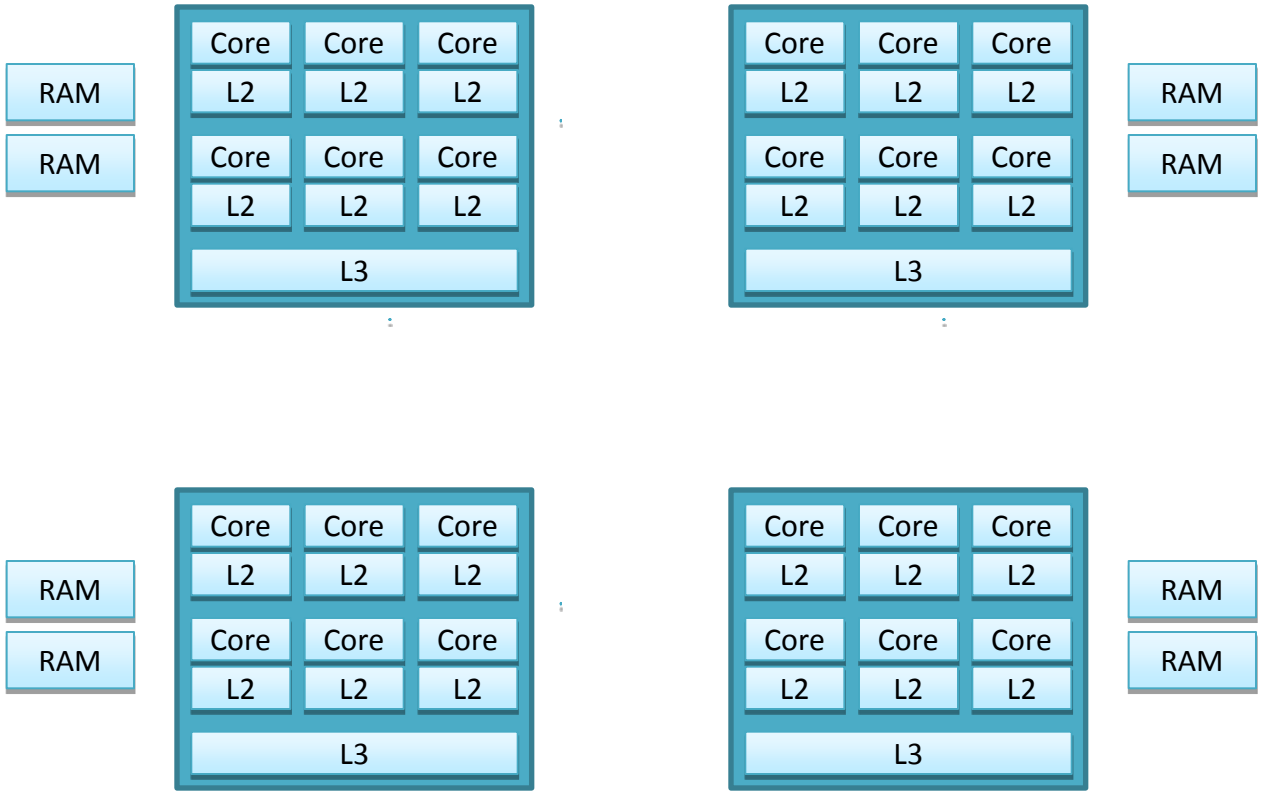


Firefly



- UMA
- Reliable interconnect
- Hardware support for cache coherence
- Bus contention an important issue
  - Analysis using trace-driven simulation and a simple queuing model found that adding processors improved performance up to about 9 processors

Cache-coherent multicore c. 2011

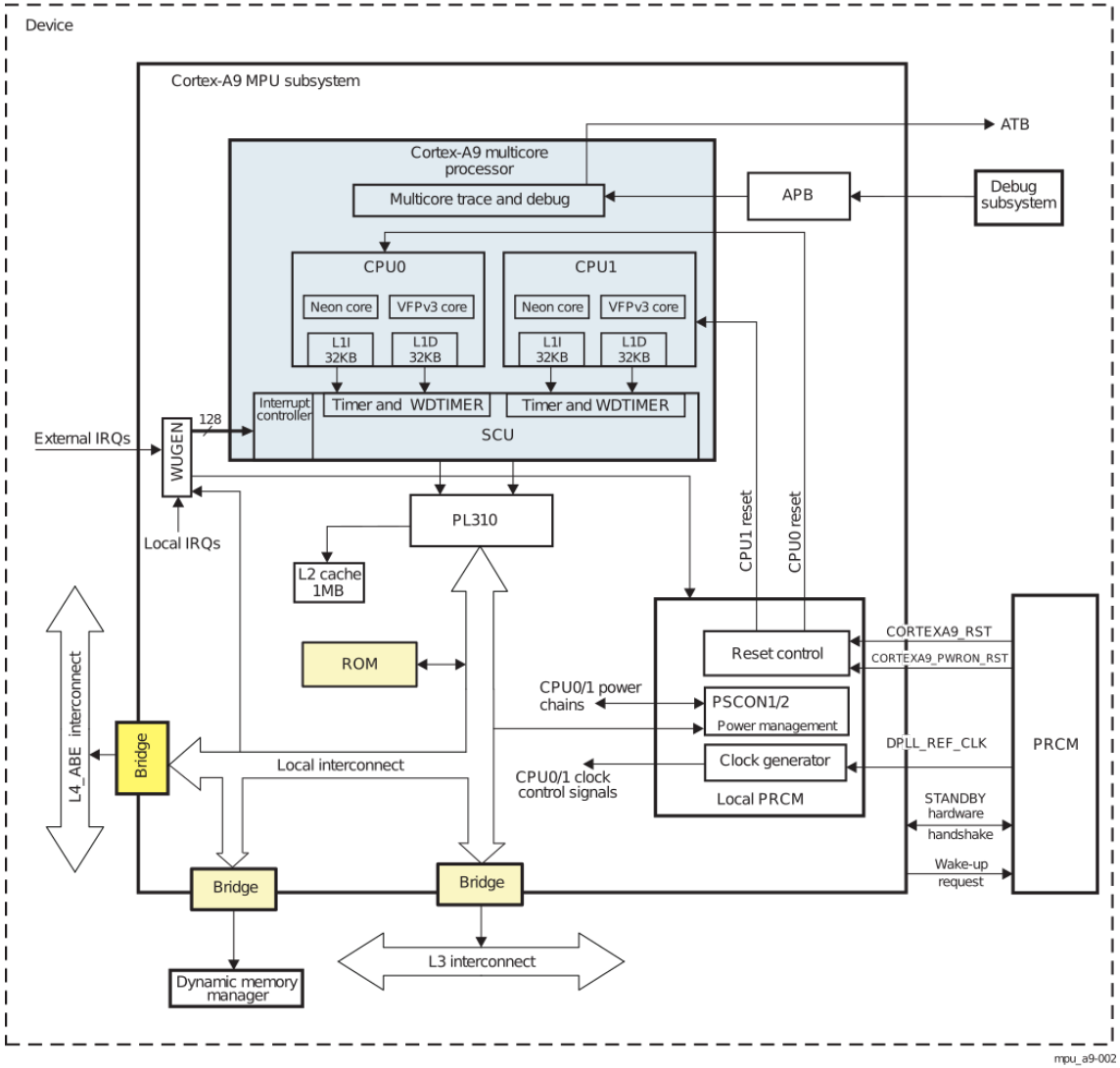


AMD Istanbul: 6 cores, per-core L2, per-package L3

The Cortex A9 MPCore

Systems@ETH zürich

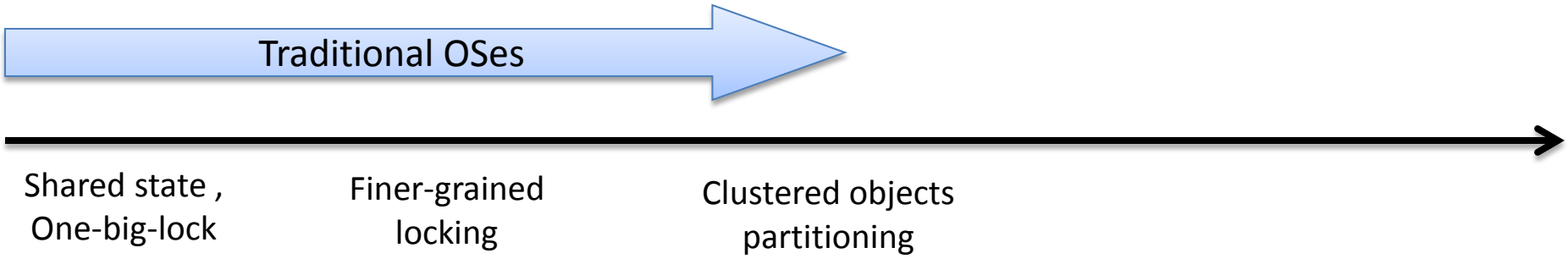
Figure 4-2. Cortex-A9 MPU Subsystem Integration



Systems@ETH zürich

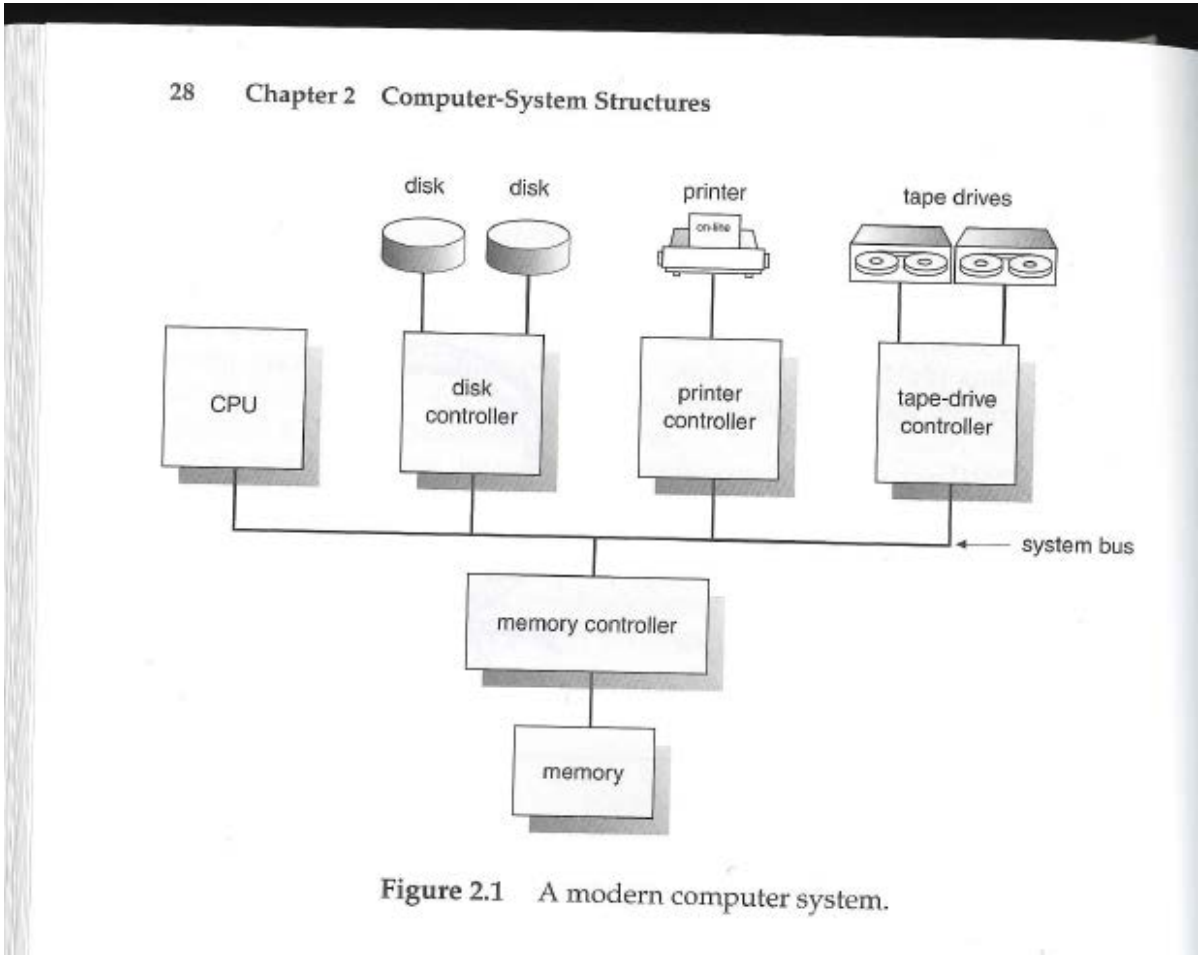
SCALING AN OS

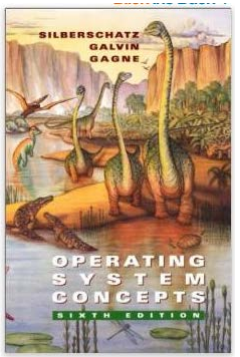
Clear trend….



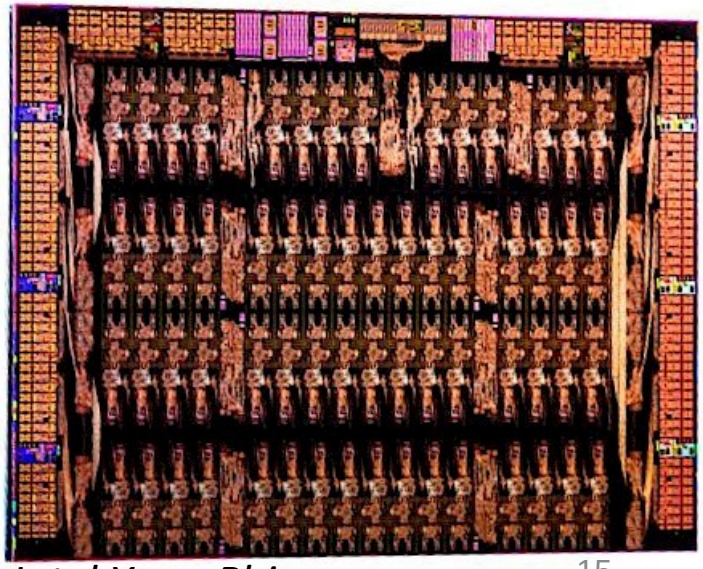
- Finer-grained locking of shared memory
- Replication as an optimization of shared memory

The Myth.

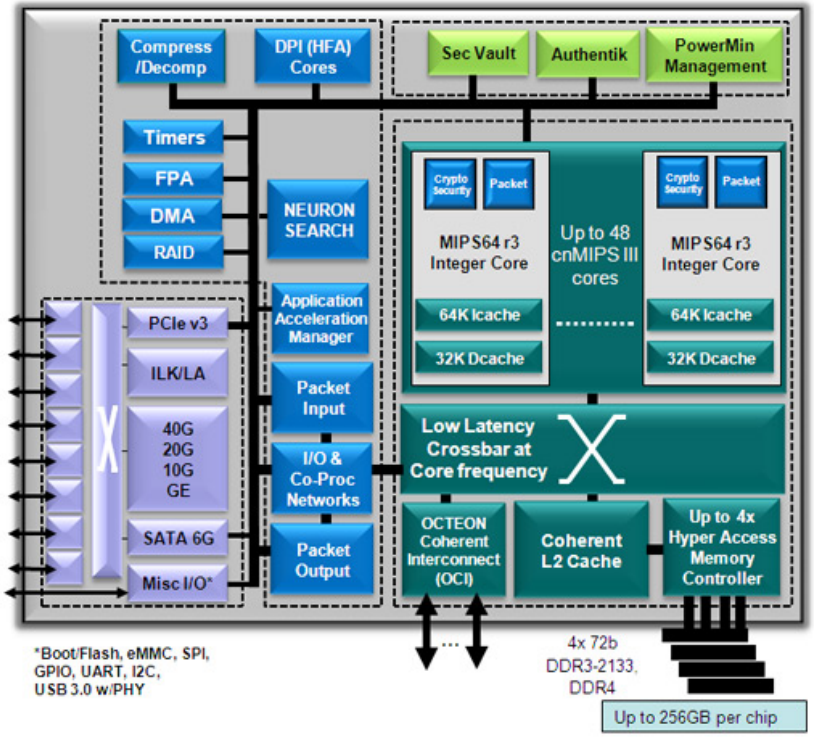

28 Chapter 2 Computer-System Structures
disk
disk
printer
tape drives
on-line
CPU
disk controller
printer controller
tape-drive controller
system bus
memory controller
memory
Figure 2.1 A modern computer system.

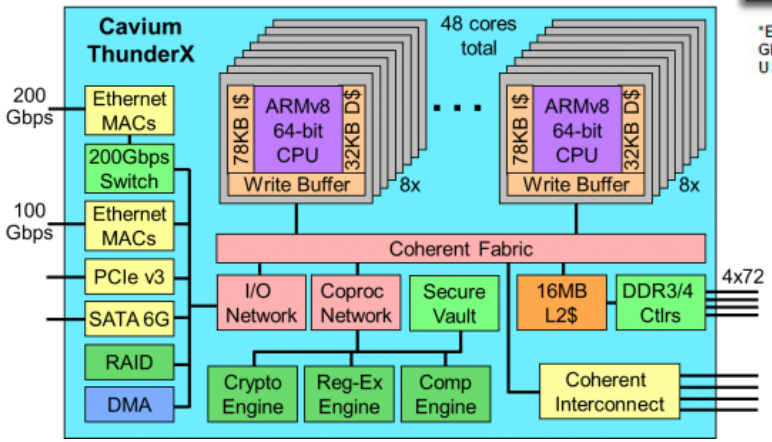

SILBERSCHATZ
GALVIN
GAGNE
OPERATING
SYSTEM
CONCEPTS
SIXTH EDITION

Lots more cores per chip



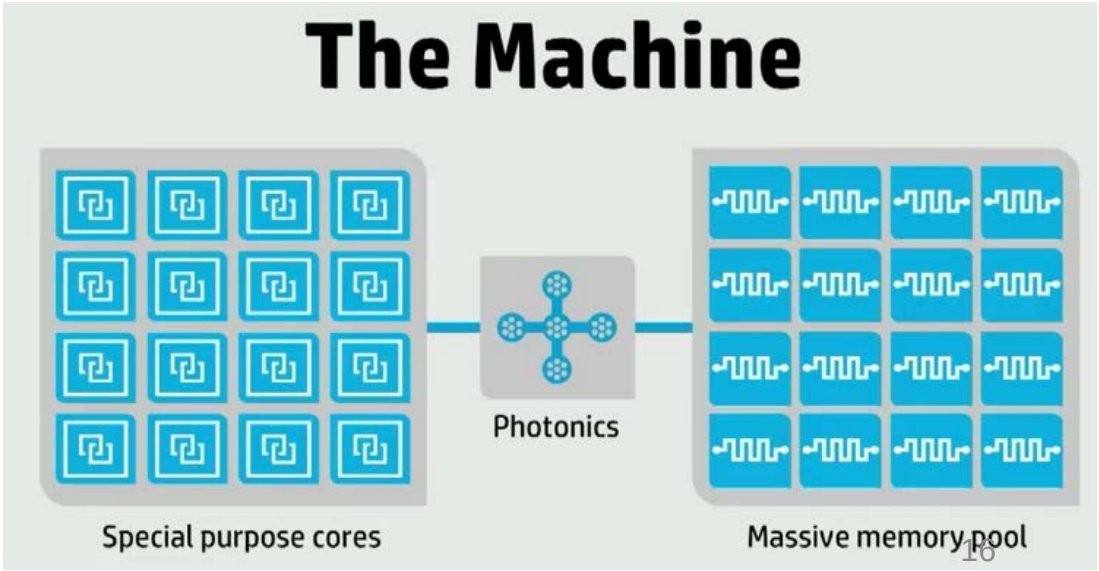
Intel Xeon Phi



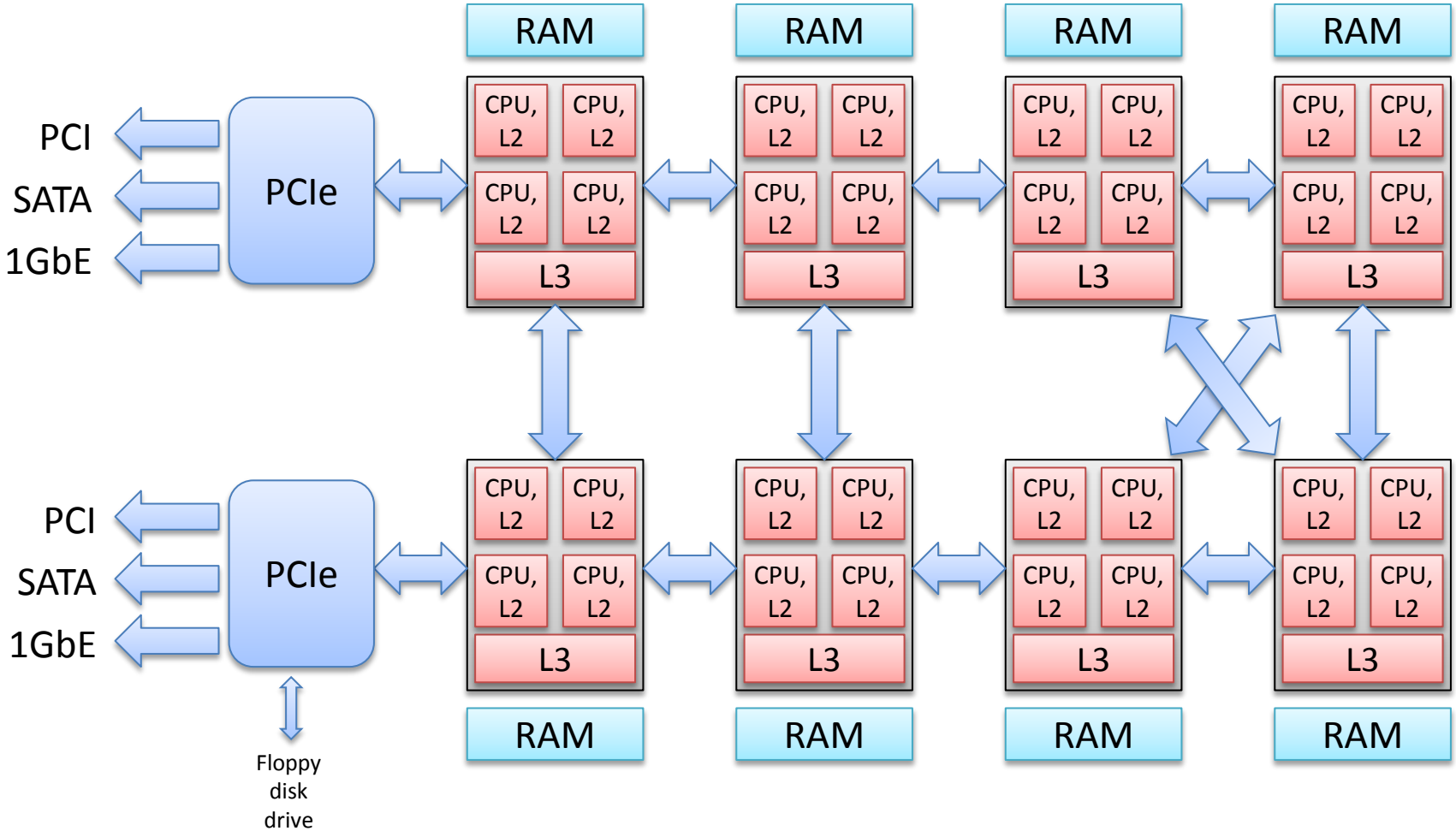


HP's "The Machine"

- Seems likely:
  - Very large persistent main memories
  - 52 physical address bits might not be enough!
    $\Rightarrow$ multiple levels of physical address translation

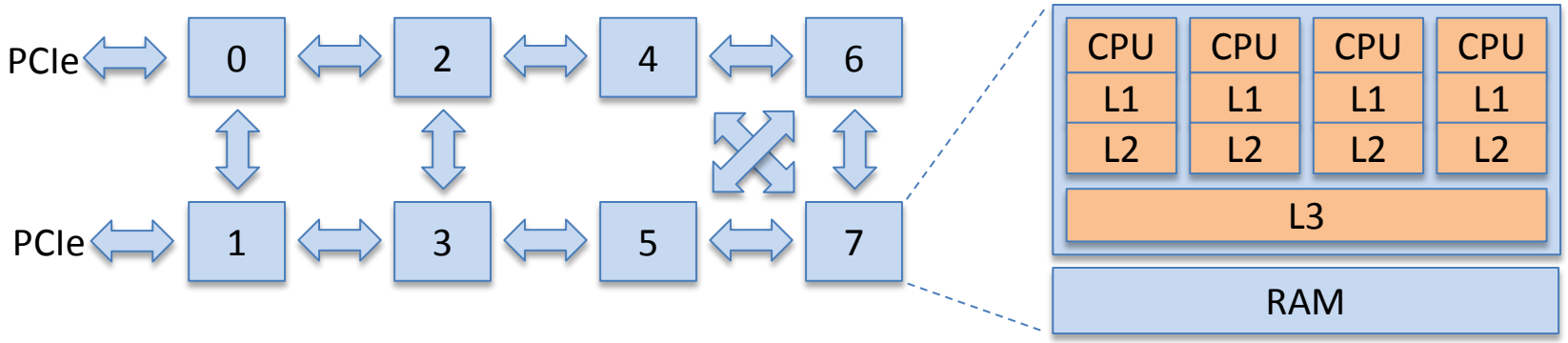


8-socket 32-core AMD Barcelona



Communication latency really matters

Example: 8 * quad-core AMD Opteron



Access	cycles	normalized to L1	per-hop cost
L1 cache	2	1	-
L2 cache	15	7.5	-
L3 cache	75	37.5	-
Other L1/L2	130	65	-
1-hop cache	190	95	60
2-hop cache	260	130	70

What makes scaling hard?

Systems@ETH zürich

- Locking *complexity*
  - Fine-grained locking
  - Strict lock acquisition orders, etc.
- Locking *overhead*
  - Scalable locks (e.g. Mellor-Crummey-Scott)
  - Read-Copy-Update and other techniques

MCS locks

[Mellor-Crummey and Scott, 1991]

Systems@ETH zürich

- **Problem**: in a CAS-based spin lock
  - cache line containing the lock is “hot”
  - constantly invalidated as each core tries to acquire
  - dominates interconnect traffic
- **Solution**: When acquiring
  - core enqueues itself on a list of waiting cores
  - spins on its own private entry in the list
- When releasing, only next core is awakened

MCS lock pseudocode

```
struct qnode {
        struct qnode *next;
        int locked;
};
typedef struct qnode *lock_t;
```

qnode in local memory

```
void acquire( lock_t *lock, struct qnode *local) {
        local->next = NULL;
        struct qnode *prev = XCHG(lock, local);
        if (prev) {  // queue was non-empty
                local->locked = 1;
                prev->next = local;
                while (local->locked) ;  // spin
        }
}
```

lock → last element of a queue of spinning processes

1. Add ourselves to the end of this queue using XCHG
2. If the queue was empty, we have the lock!
3. If not, point the previous tail at us, and spin.

MCS lock pseudocode

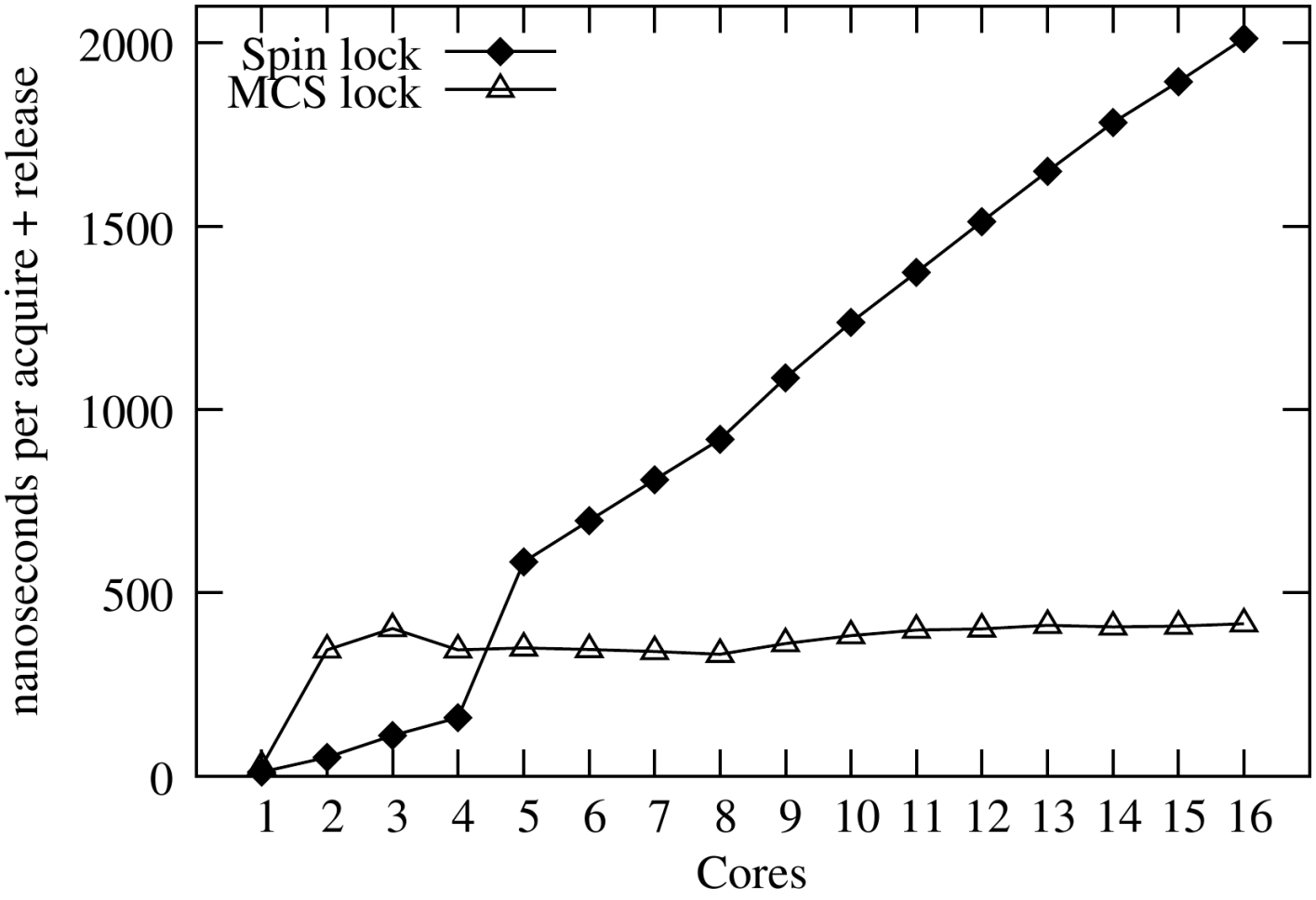
```
struct qnode {
           struct qnode *next;
           int locked;
};
typedef struct qnode *lock_t;
```

```
void release (lock_t *lock, struct qnode *local) {
      if (local->next == NULL) {
            if ( CAS(lock, local, NULL) )
                  return;
             while (local->next == NULL) ; // spin
      local->next->locked = 0;
}
```

1. We have the lock.
   Is someone after us waiting?
2. If yes, tell them, and they will do the rest
   (see acquire() !)
3. If no, set the lock to NULL unless someone appears in the meantime
4. If they do, wait for them to enqueue, and then go to (2)

MCS lock performance

4x4-core AMD Opteron, Linux



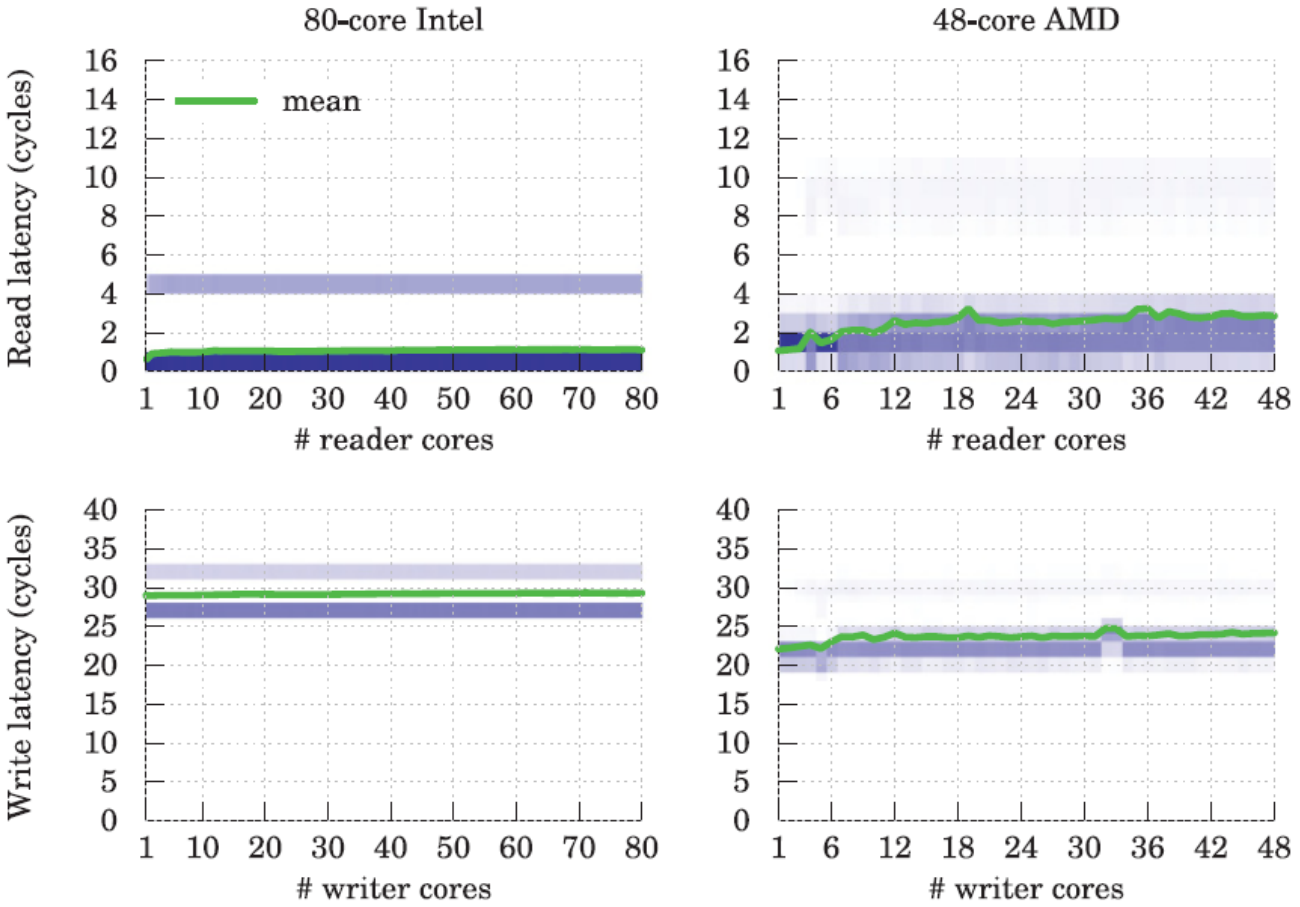
[Boyd-Wickizer et al., 2008]

What makes scaling hard?

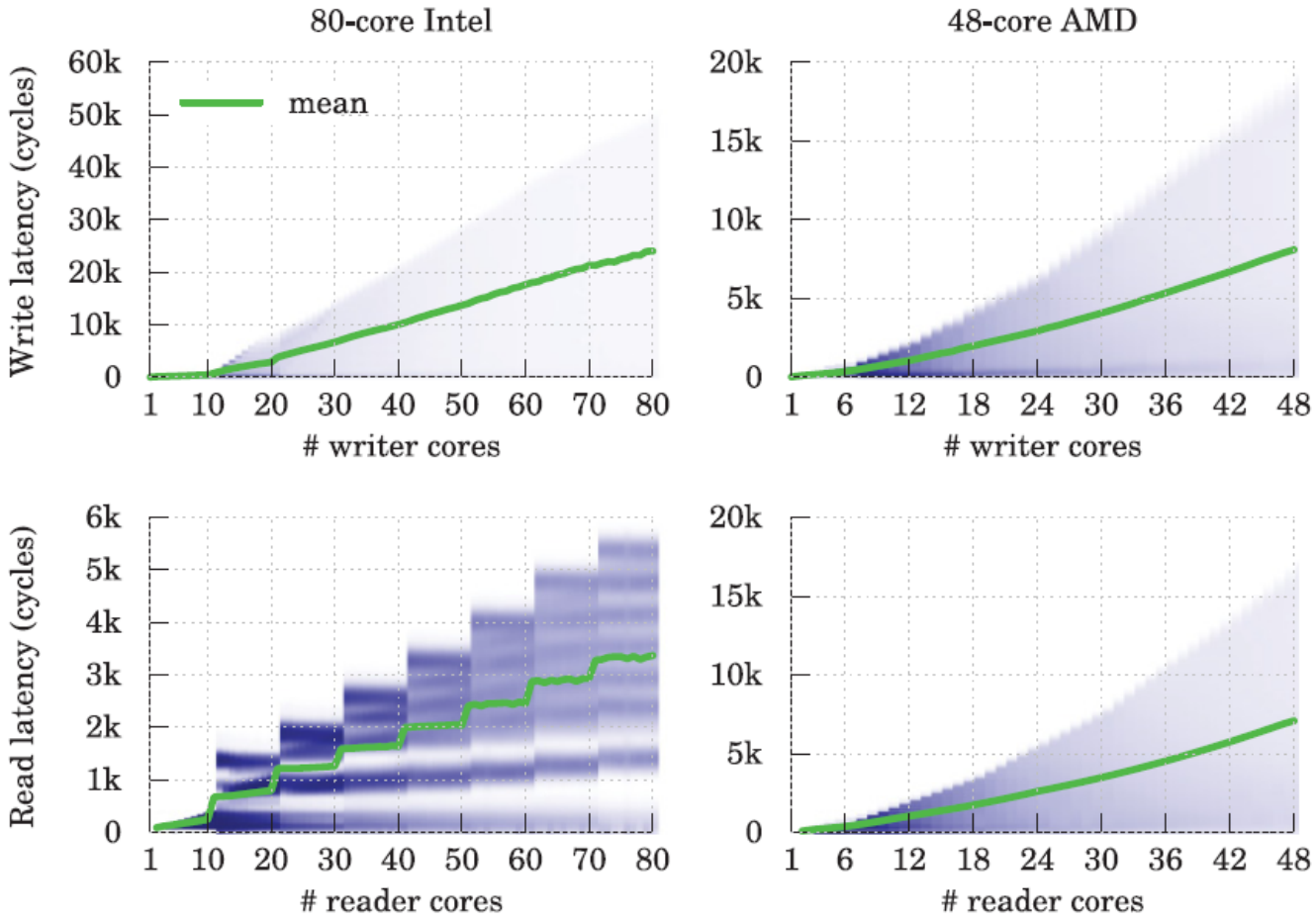
Systems@ETH zürich

- Locking *complexity*
  - Fine-grained locking
  - Strict lock acquisition orders, etc.
- Locking *overhead*
  - Scalable locks (e.g. Mellor-Crummey-Scott)
  - Read-Copy-Update and other techniques
- Sharing *data*
  - Moving cache lines between cores

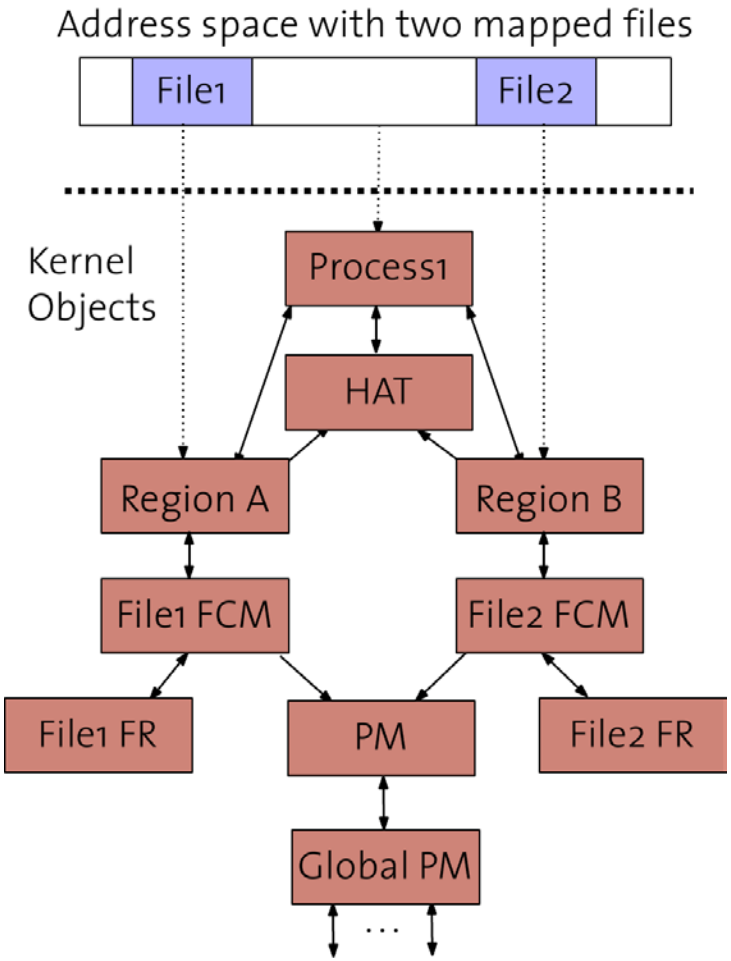
Conflict-free reads and writes



Conflicting reads and writes



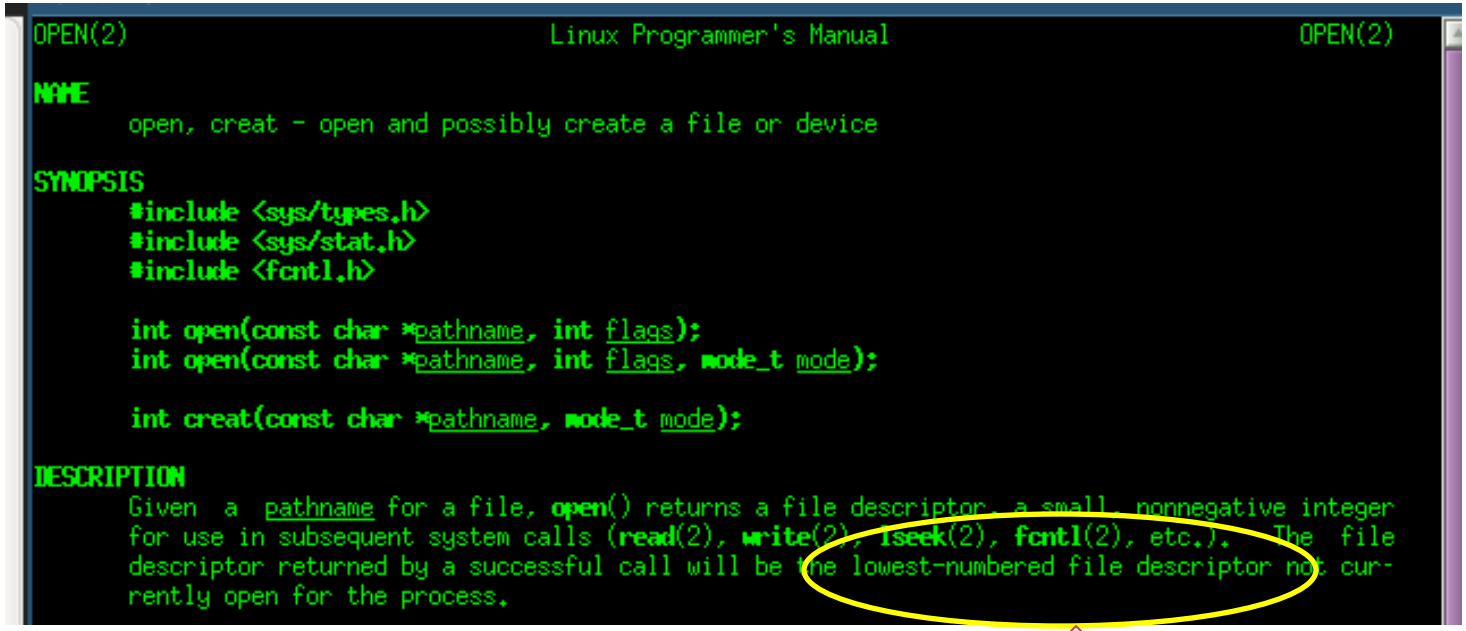
Example: VM objects in K42



- Why this decomposition?
- Which ones are possible?

- File system and VM API constrains the implementation!

API scaling example



```
OPEN(2)                          Linux Programmer's Manual                          OPEN(2)

NAME
       open, creat - open and possibly create a file or device

SYNOPSIS
       #include <sys/types.h>
       #include <sys/stat.h>
       #include <fcntl.h>

       int open(const char *pathname, int flags);
       int open(const char *pathname, int flags, mode_t mode);

       int creat(const char *pathname, mode_t mode);

DESCRIPTION
       Given  a  pathname for a file, open() returns a file descriptor, a small, nonnegative integer
       for use in subsequent system calls (read(2), write(2), lseek(2), fcntl(2), etc.).   The  file
       descriptor returned by a successful call will be the lowest-numbered file descriptor not cur-
       rently open for the process.
```

Can we generalize this notion?

Requires coordination across all threads!

The Scalable Commutativity Rule

Systems@ETH zürich

Where several API operations commute,
$\exists$ an implementation which is conflict-free
$\Rightarrow$ this implementation will scale.

Commute $\equiv$ their execution order cannot be distinguished from the interface

Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. 2015. The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors. ACM Trans. Comput. Syst. 32, 4, Article 10 (January 2015),

Another Unix issue

Systems@ETH zürich

- Local Unix domain sockets order all messages
  $\Rightarrow$ `send`/`recv` calls on do not commute on a socket.

- Relaxing this would allow scalable Unix domain datagram socket communication

Reducing sharing

Systems@ETH zürich

- Per-core data structures
  - c.f. K42 “clustered objects”
  - separate scheduling queues
  - distributed memory allocators
  - etc.
- But how to decompose the system into objects?

WHAT ELSE IS NEW?

What else is new?

Systems@ETH zürich

- Hardware is changing faster than system software
  - Engineering effort to fix scaling problems is becoming overwhelming
- Hardware is already diverse
  - Can't tune OS design to any one machine architecture
- Cores will not all be the same
  - Different performance characteristics
  - Different instruction set variants
  - Different architectures (GPUs, NICs, etc.)

The Gap.

Systems@ETH zürich

For many commercially relevant workloads, cores spend much of their time in the OS.

BUT:

- Processor architects ignore OS designers
  - Don't understand the OS problem
  - Cores never evaluated with >1 app running anyway
- HPC people try to remove the OS
  - And then blow the rest of their s/w development budget putting it back in a user library.

The Gap.

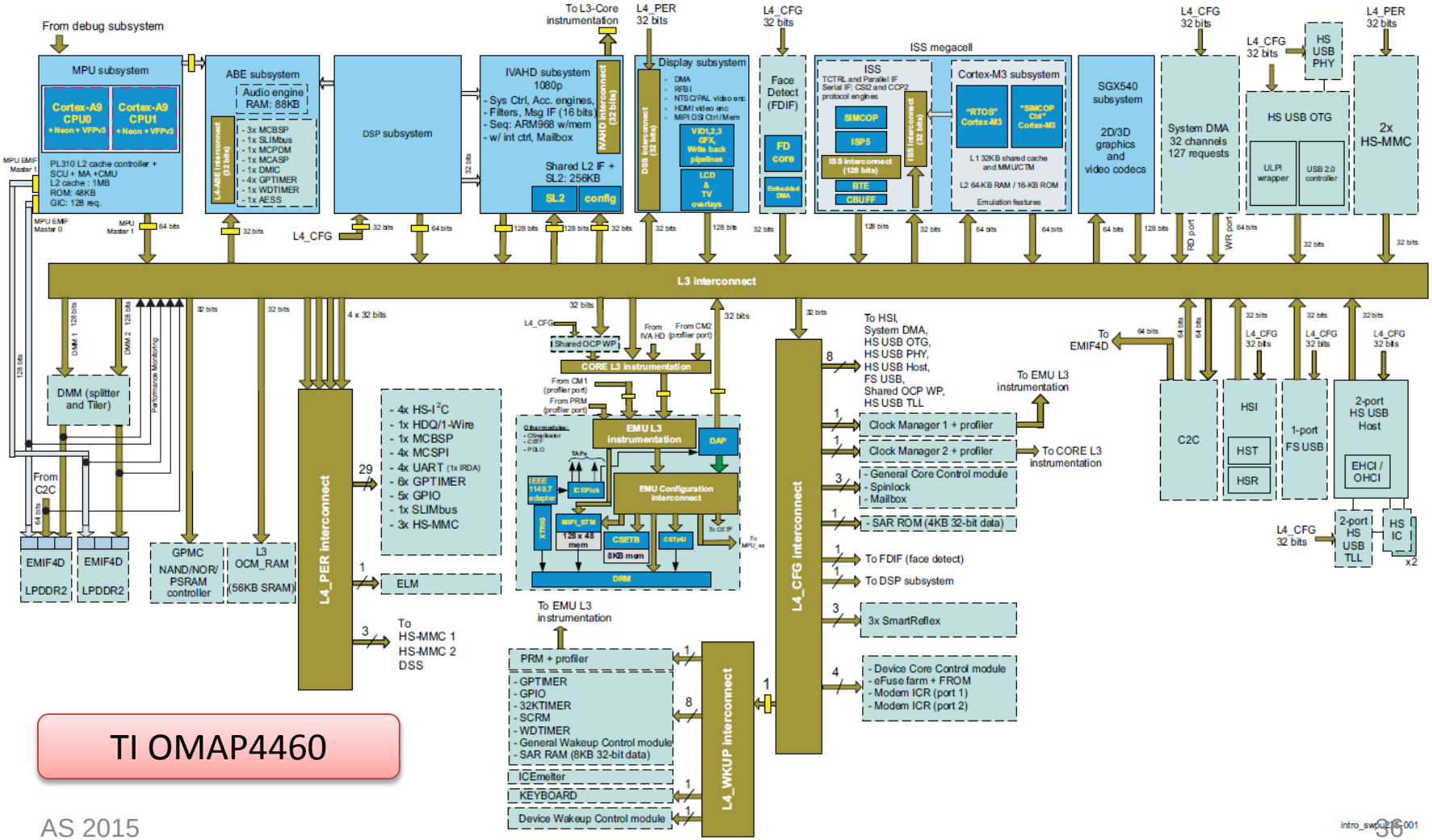
Systems@ETH zürich

For many commercially relevant workloads, cores spend much of their time in the OS.

BUT:

- Processor architects ignore OS designers
  - Don’t understand the OS problem
  - Cores never evaluated with >1 app running anyway
- HPC people try to remove the OS
  - And then blow the rest of their s/w development budget putting it back in a user library.
- and OS design people?
  - Complain among themselves and try and deal with it
  - Don't even try to influence hardware

Every new SoC is different

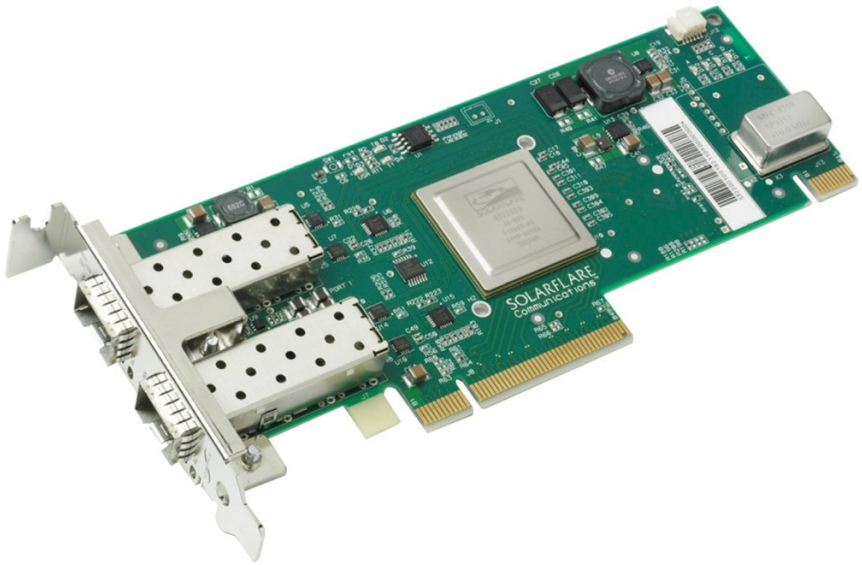


TI OMAP4460

Lots of fixed-function hardware

- Checksum offload
- Segmentation offload
- Multicast filtering
- TCP offload engines
- RDMA hardware support
- Receive-side scaling
- Traffic shaping
- Single-Root I/O virtualization
- Lots of send/receive queues
- Direct Cache Access

etc., etc.



Memory isn't fully cache-coherent, or even shared

Hardware comes and goes

Systems@ETH zürich

- Hotplug everything
  - CPUs, memory, I/O subsystems
- Partial or transient hardware failures will become the norm
  - Potentially of cores and interconnects as well as memory and devices
- Power management is already pervasive
  - Complex dependencies among devices
  - Dark silicon

What the OS needs to know

Systems@ETH zürich

- TI OMAP 5430 software manual is 6107 pages.
  - The first 277 are the table of contents
- For each SoC, the OS needs to know:
  - Locations of all hardware devices
    - … as seen by each core
    - … some of which are on nested buses
      - Memory bus -> I2C -> GPIO pins
  - Interrupt routing
  - Power well allocation and dependencies

Implications

Systems@ETH zürich

- Computers today and in the future are systems of cores and other devices which:
  - Are connected by highly **complex** interconnects
  - Entail significant communication **latency** between nodes
  - Consist of **heterogeneous** cores
  - Show unpredictable **diversity** of system configurations
  - Have a variety of **complex fixed-functions** for the same job
  - Have **dynamic** core set membership
  - Provide only **limited shared** memory or cache **coherence**

Today's operating systems
(and hypervisors)

Systems@ETH zürich

- Single large kernel executed by every core
  - Data structures in coherent shared memory
- Anything else is a device
  - Offload processors: GPUs, accelerators, FPGAs, …
  - Large opaque firmare blobs
  - Narrow driver interface
- OS policies for hardware management written in C

Implications

Systems@ETH zürich

- Computers today and in the future are systems of cores and other devices which:
  - Are connected by highly **complex** interconnects
  - Entail significant communication **latency** between nodes
  - Consist of **heterogeneous** cores
  - Show unpredictable **diversity** of system configurations
  - Have a variety of **complex fixed-functions** for the same job
  - Have **dynamic** core set membership
  - Provide only **limited shared** memory or cache **coherence**

Implications

- Computers today and in the future are systems of cores and other devices which:

The OS model of cooperating processes over a shared-memory multithreaded kernel is dead.

  - Consist of **heterogeneous** cores
  - Show unpredictable **diversity** of system configurations
  - Have **dynamic** core set membership
  - Provide only **limited shared** memory or cache **coherence**

1970s abstractions for managing and programming hardware aren't going to cut it.

What's the future of this problem?

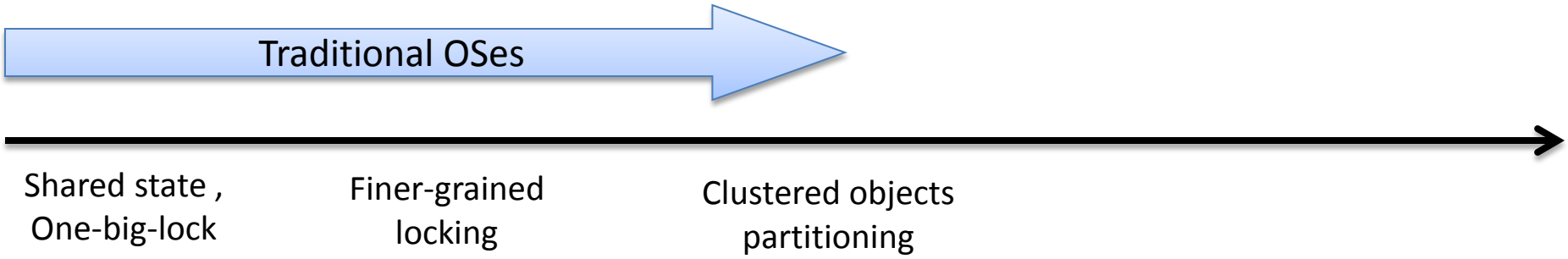
Systems@ETH zürich

- Hardware proliferates and diversifies.
- Hardware becomes more complex.
- Efficient use of hardware gets more critical.

Systems@ETH zürich

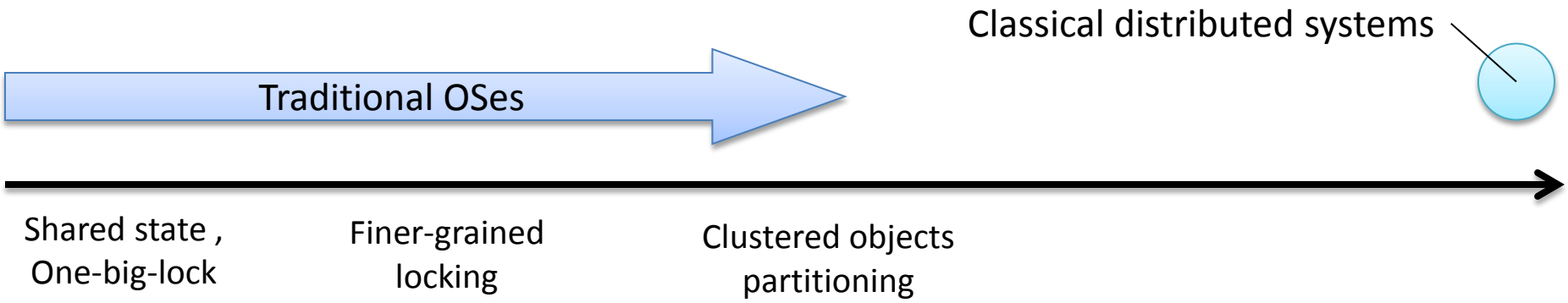
MULTIKERNELS

Clear trend….



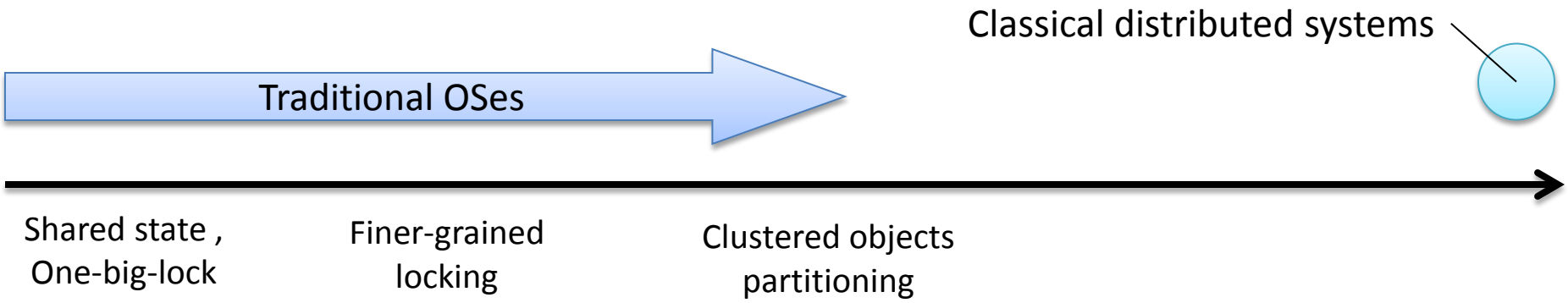
- Finer-grained locking of shared memory
- Replication as an optimization of shared memory

Clear trend….



- Finer-grained locking of shared memory
- Replication as an optimization of shared memory

Clear trend….



- Finer-grained locking of shared memory
- Replication as an optimization of shared memory

What if we build an OS as a distributed system?

The Multikernel Architecture

Systems@ETH zürich

- Computers today and in the future are systems of cores and other devices which:
  - Are connected by highly **complex** interconnects
  - Entail significant communication **latency** between nodes
  - Consist of **heterogeneous** cores
  - Show unpredictable **diversity** of system configurations
  - Have a variety of **complex fixed-functions** for the same job
  - Have **dynamic** core set membership
  - Provide only **limited shared** memory or cache **coherence**

$\Rightarrow$ Forget about shared memory.

The OS is a distributed system based on **message passing**

Multikernel principles

Systems@ETH zürich

- Share ***no*** data between cores
  - All inter-core communication is via explicit messages
  - Each core can have its own implementation
- OS state partitioned if possible, replicated if not
  - State is accessed ***as if*** it were a local replica
- Invariants enforced by distributed algorithms, not locks
  - Many operations become split-phase and asynchronous

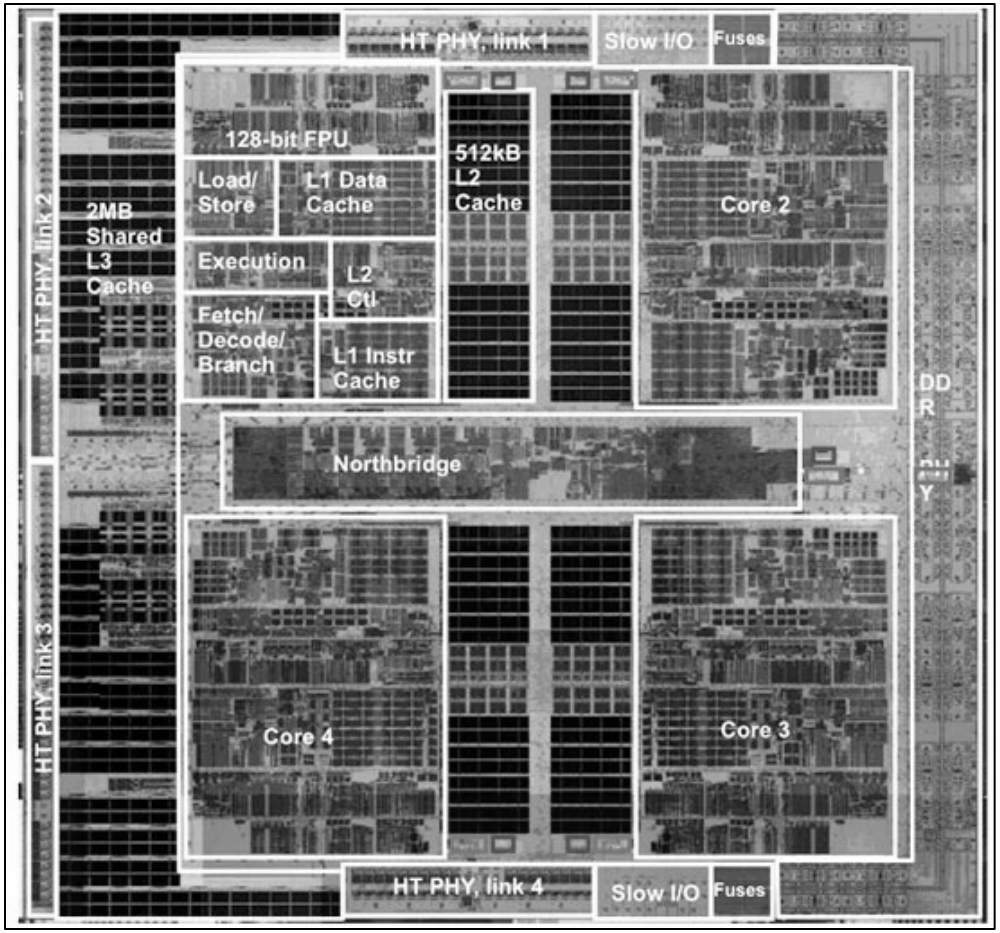
Message passing vs. shared memory

Systems@ETH zürich

- Structures are duals (Lauer & Needham, 1978)
  - Choice depends on machine architecture
- Shared memory has been favoured until now
- What are the trade-offs?
  - Depends on data size and amount of contention

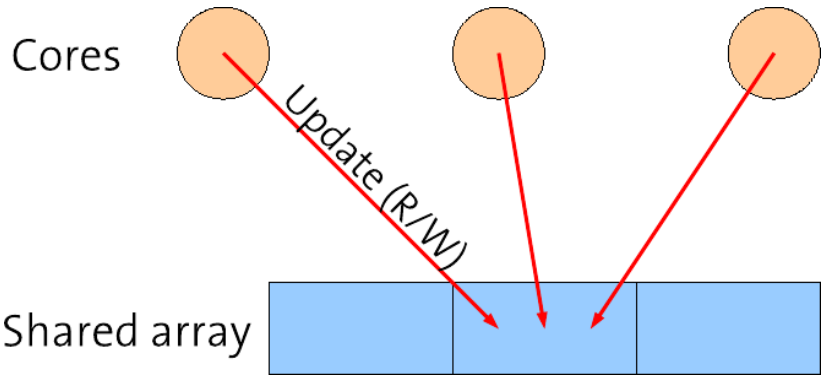
Experiment: shared memory vs message-passing

- Measure costs (latency per operation) of updating a shared data structure
- Hardware: 4*quad-core AMD Opteron



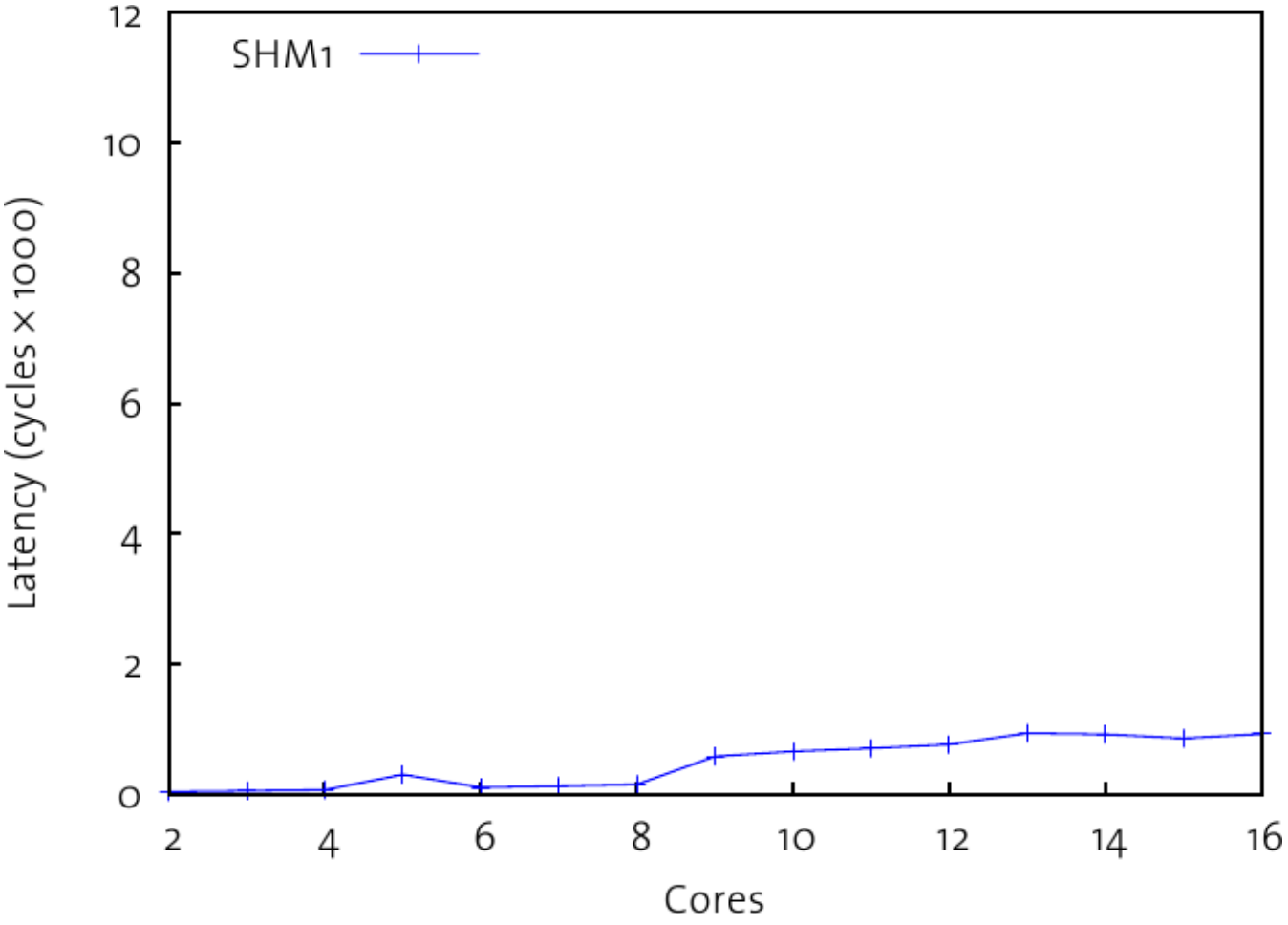
Message passing vs. shared memory: experiment

- Shared memory (move the data to the operation):
- Each core updates the same memory locations (no locking)
- Cache-coherence protocol migrates modified cache lines
  - Processor stalled while line is fetched or invalidated
  - Limited by latency of interconnect round-trips
  - Performance depends on data size (cache lines) and contention (number of cores)



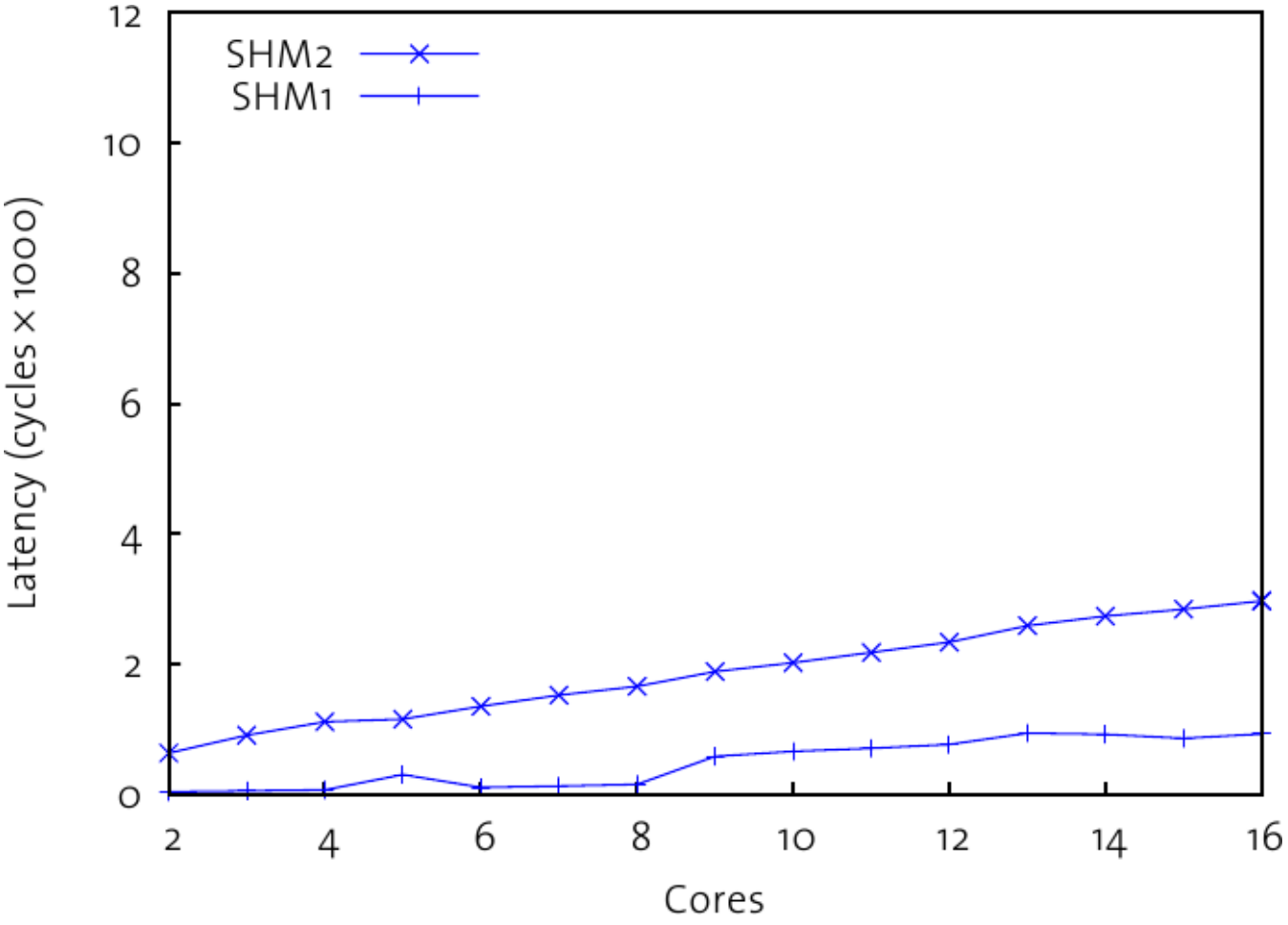
Shared memory results

4x4-core AMD system



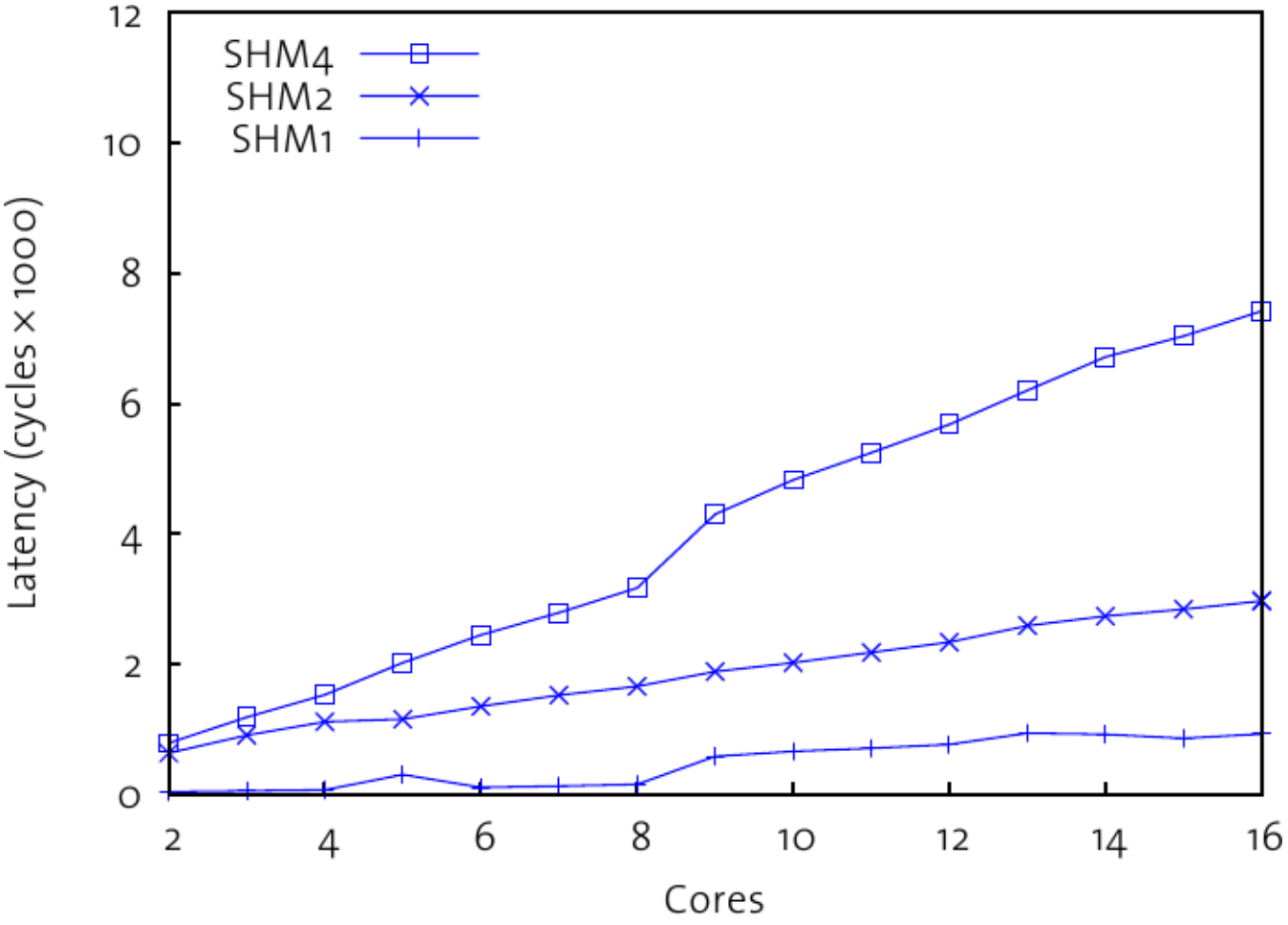
Shared memory results

4x4-core AMD system



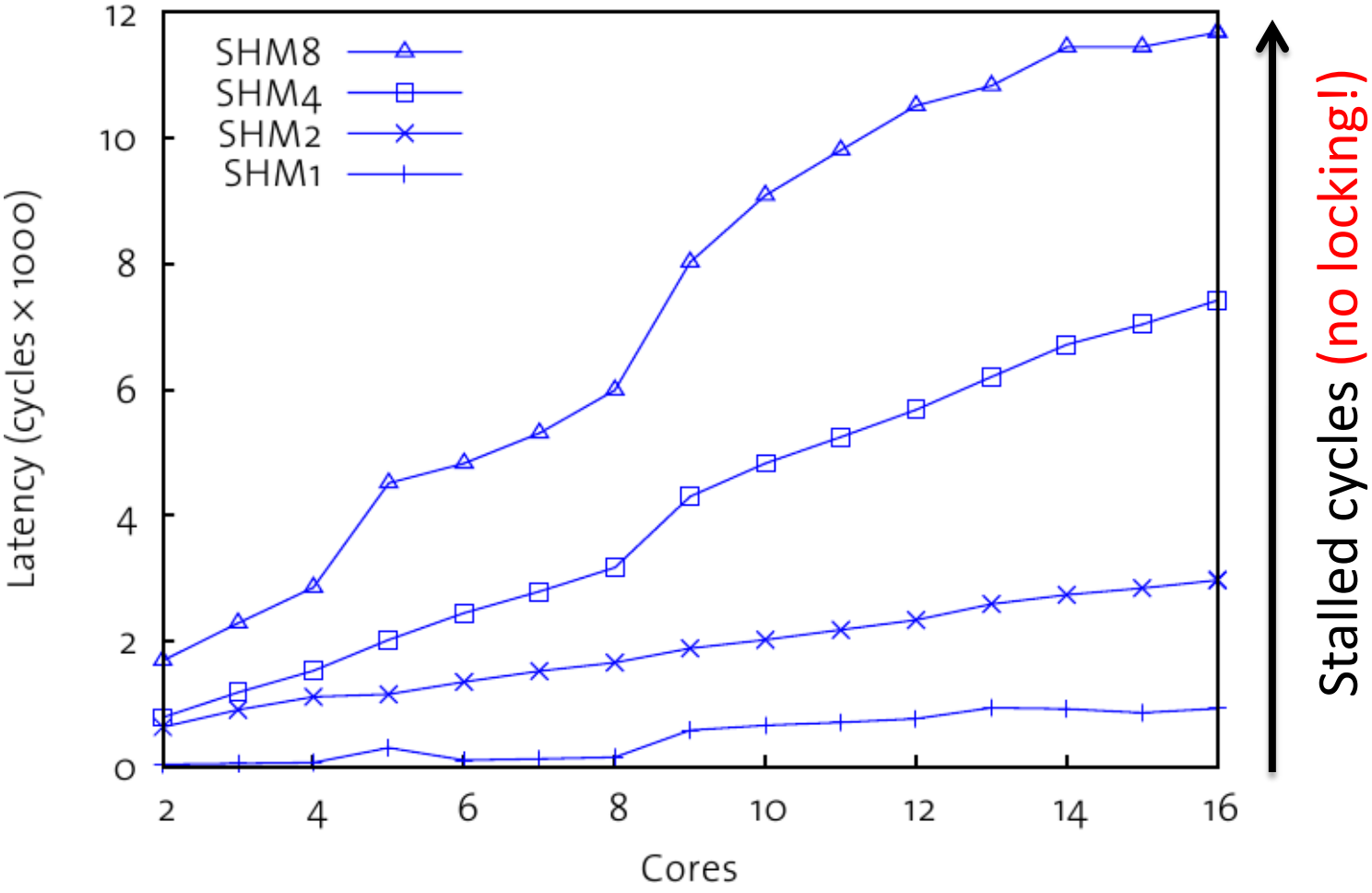
Shared memory results

4x4-core AMD system



Shared memory results

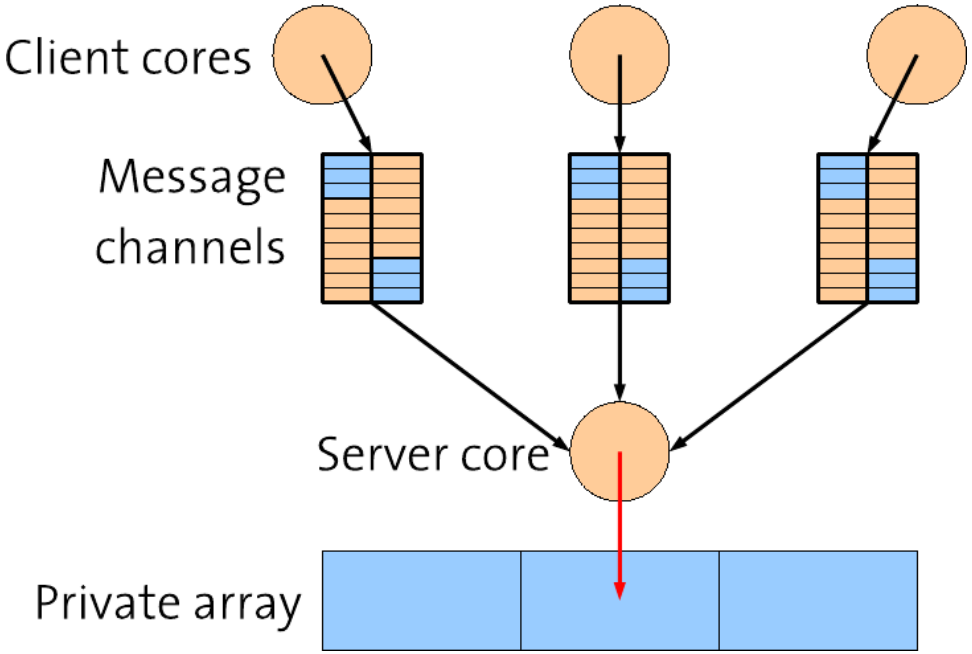
4x4-core AMD system



Message passing vs. shared memory: experiment

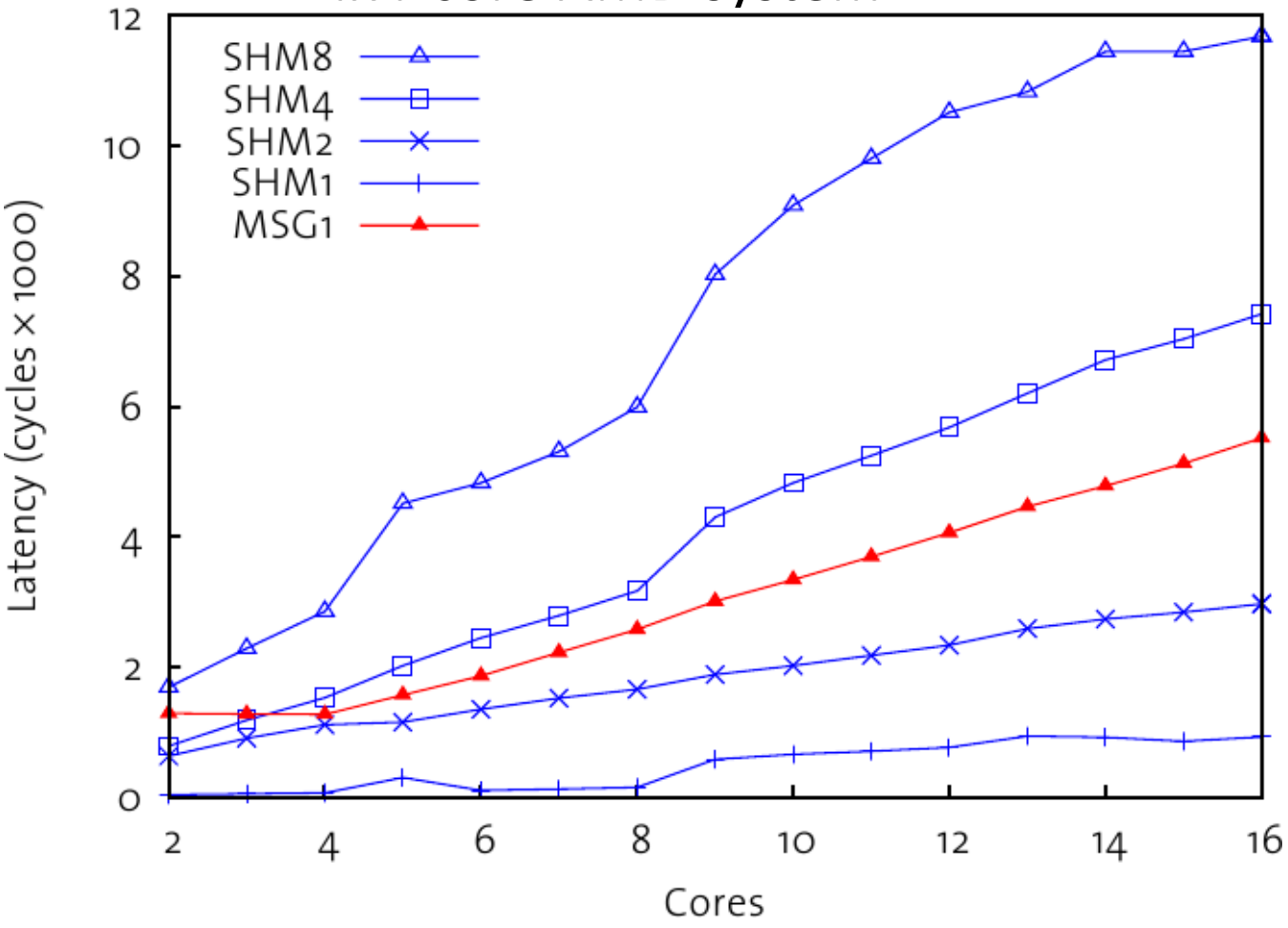
Message passing (move the operation to the data):

- A single server core updates the memory locations
- Each client core sends RPCs to the server
  - Operation and results described in a single cache line
  - Block while waiting for a response (in this experiment)

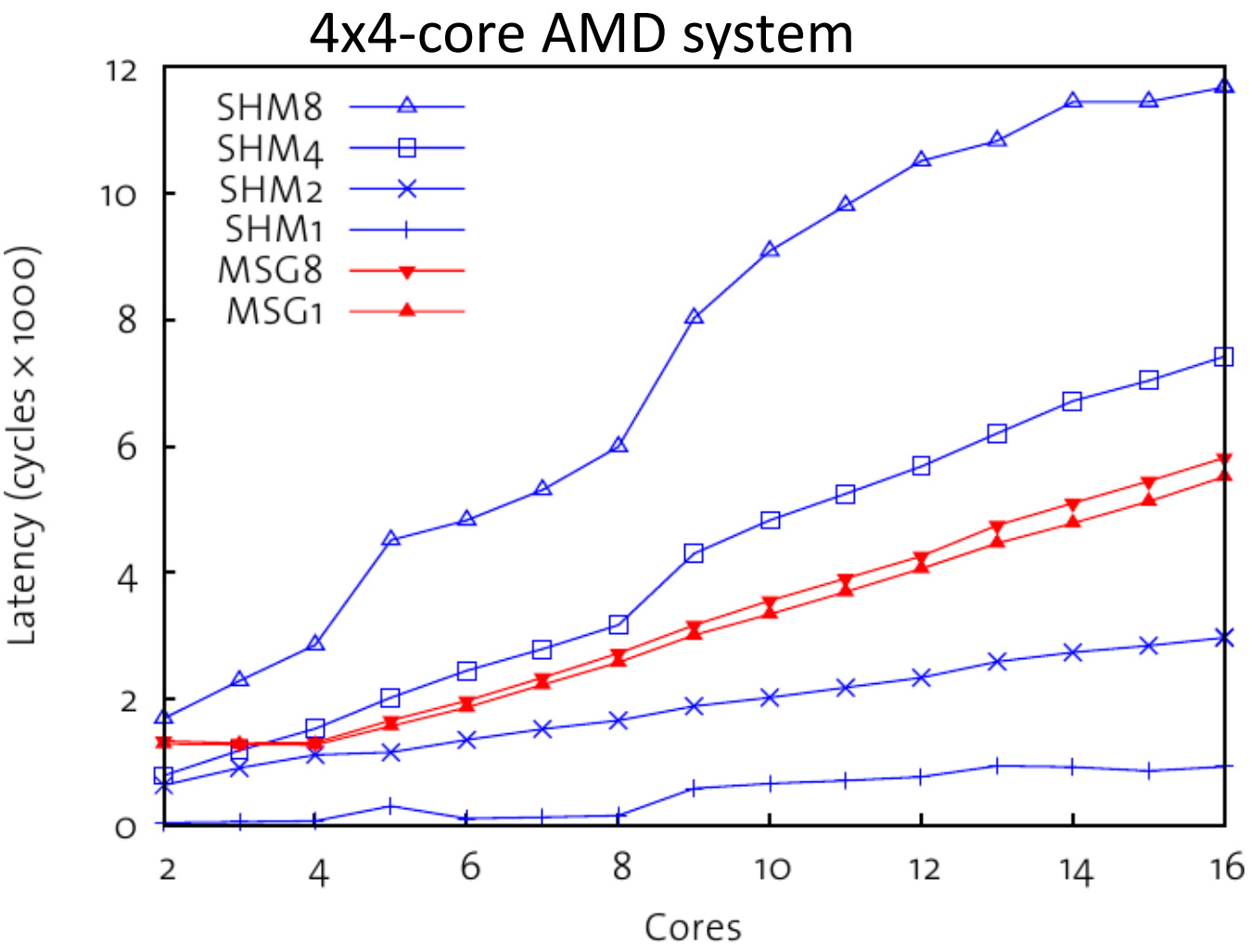


Message passing vs. shared memory: tradeoff

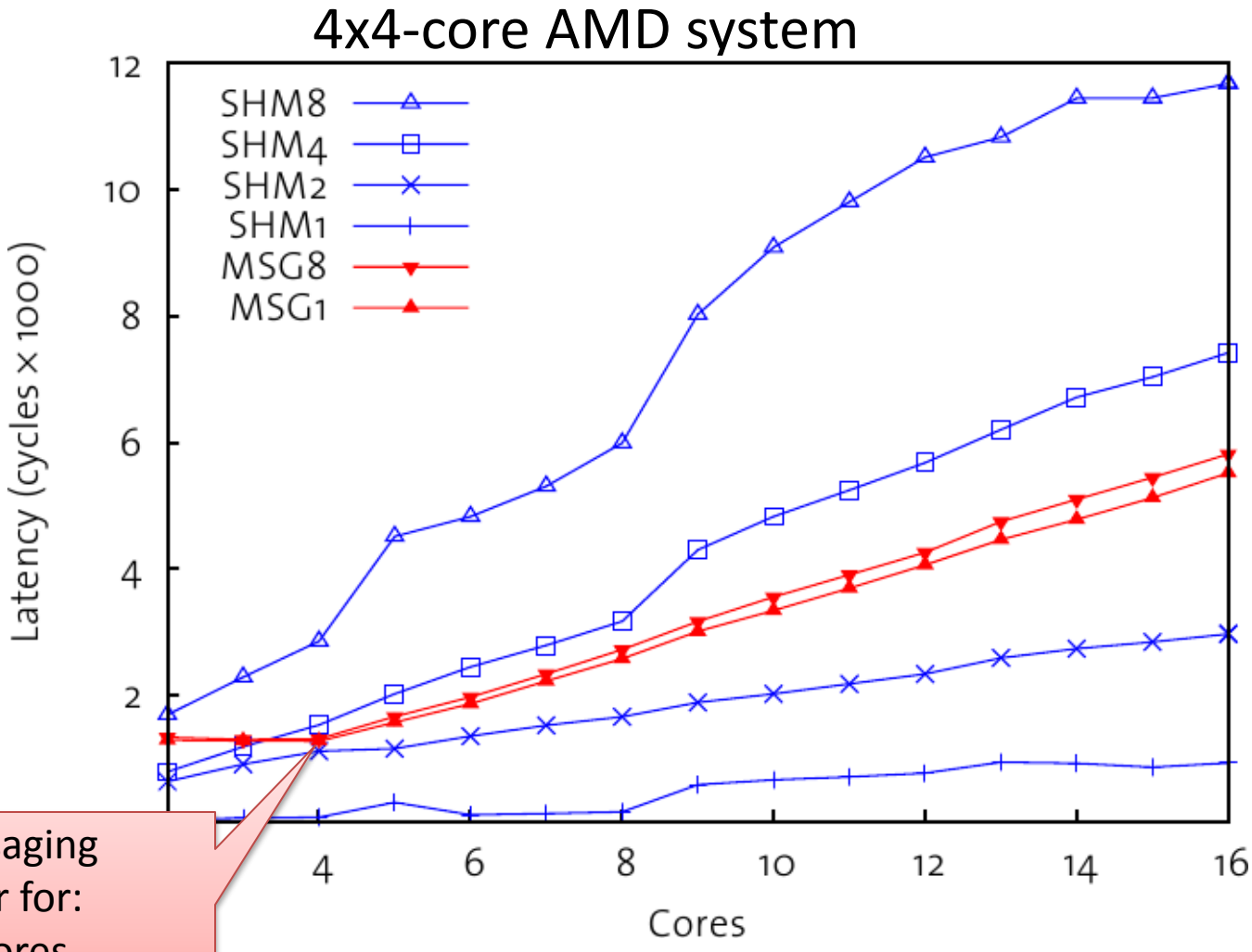
4x4-core AMD system



Message passing vs. shared memory: tradeoff



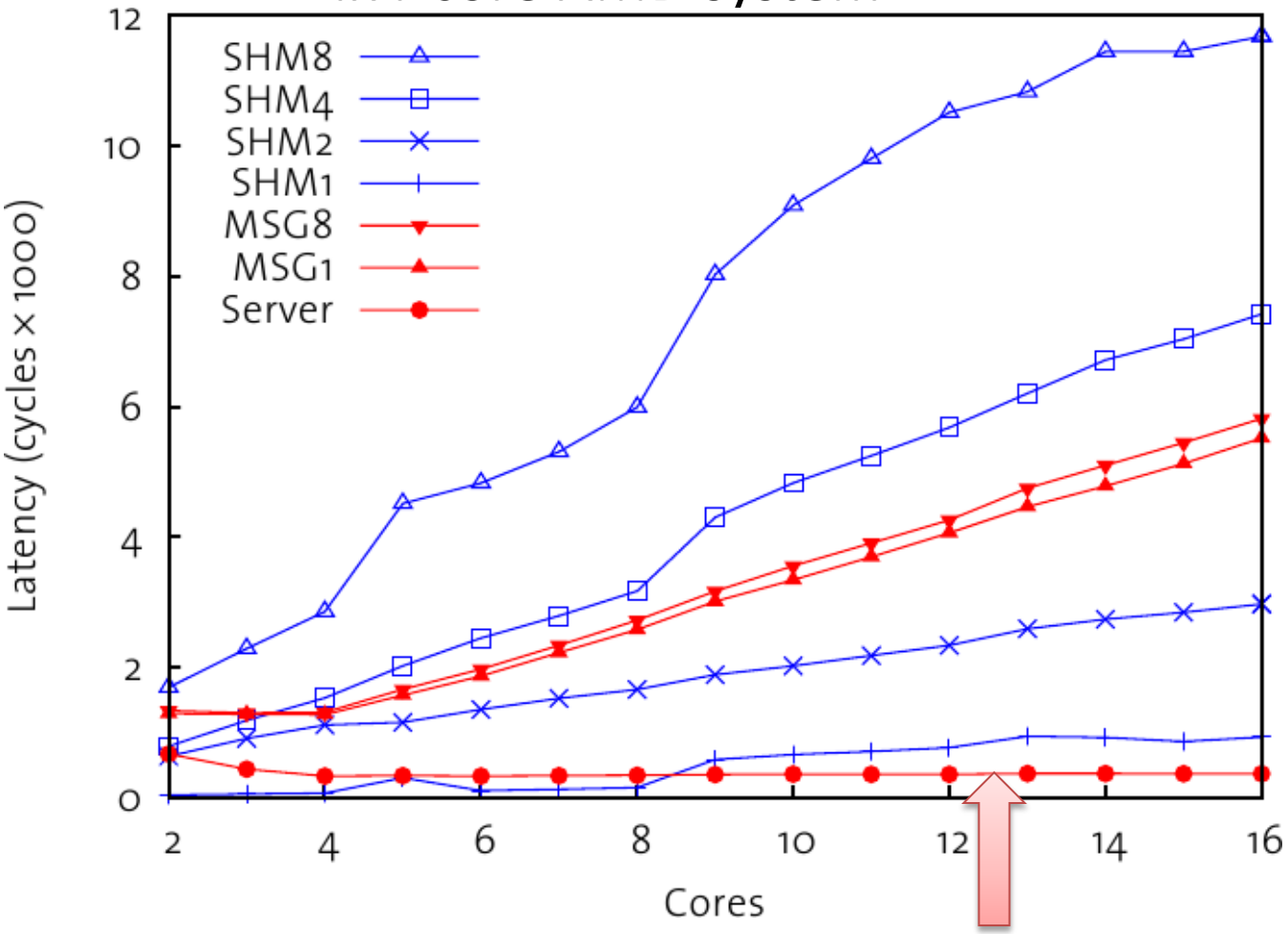
Message passing vs. shared memory: tradeoff



Messaging faster for:
≥ 4 cores
≥ 4 cache lines

Message passing vs. shared memory: tradeoff

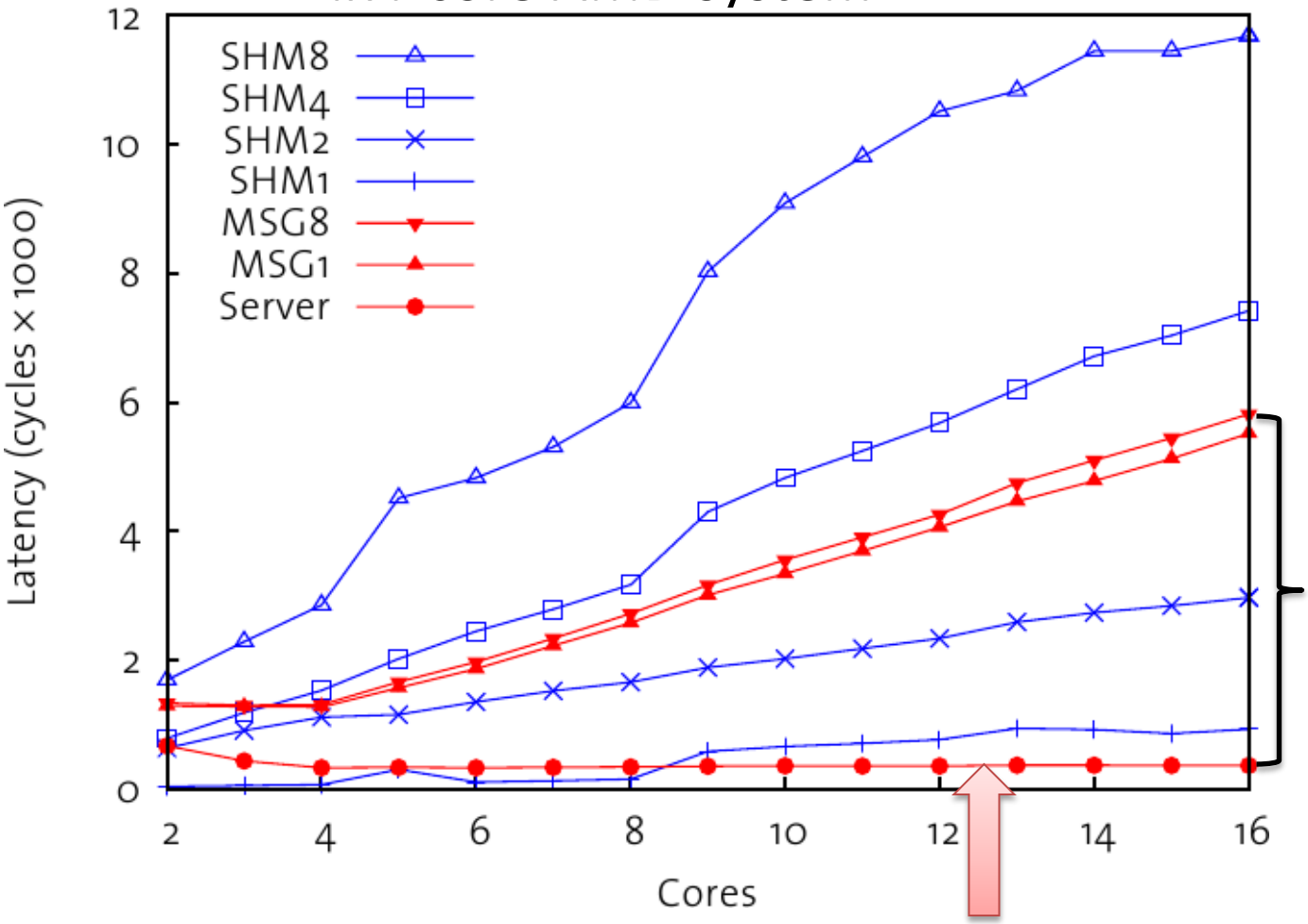
4x4-core AMD system



Actual cost of update at server

Message passing vs. shared memory: tradeoff

4x4-core AMD system



Systems@ETH zürich

BARRELFISH

The Barrelfish multikernel

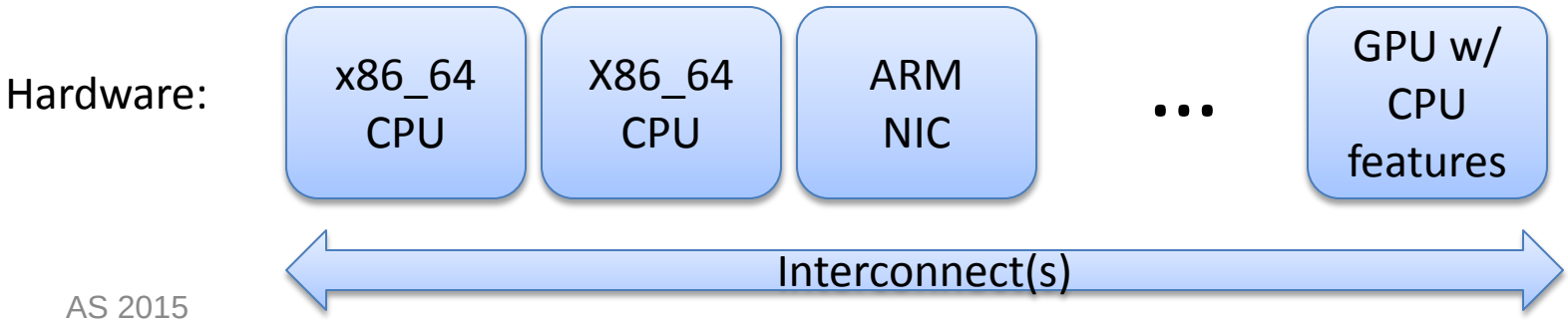
Systems@ETH zürich

- Mix of untrusted applications
- Soft real-time requirements for some
- Variety of hardware platforms
- Primarily a vehicle for research

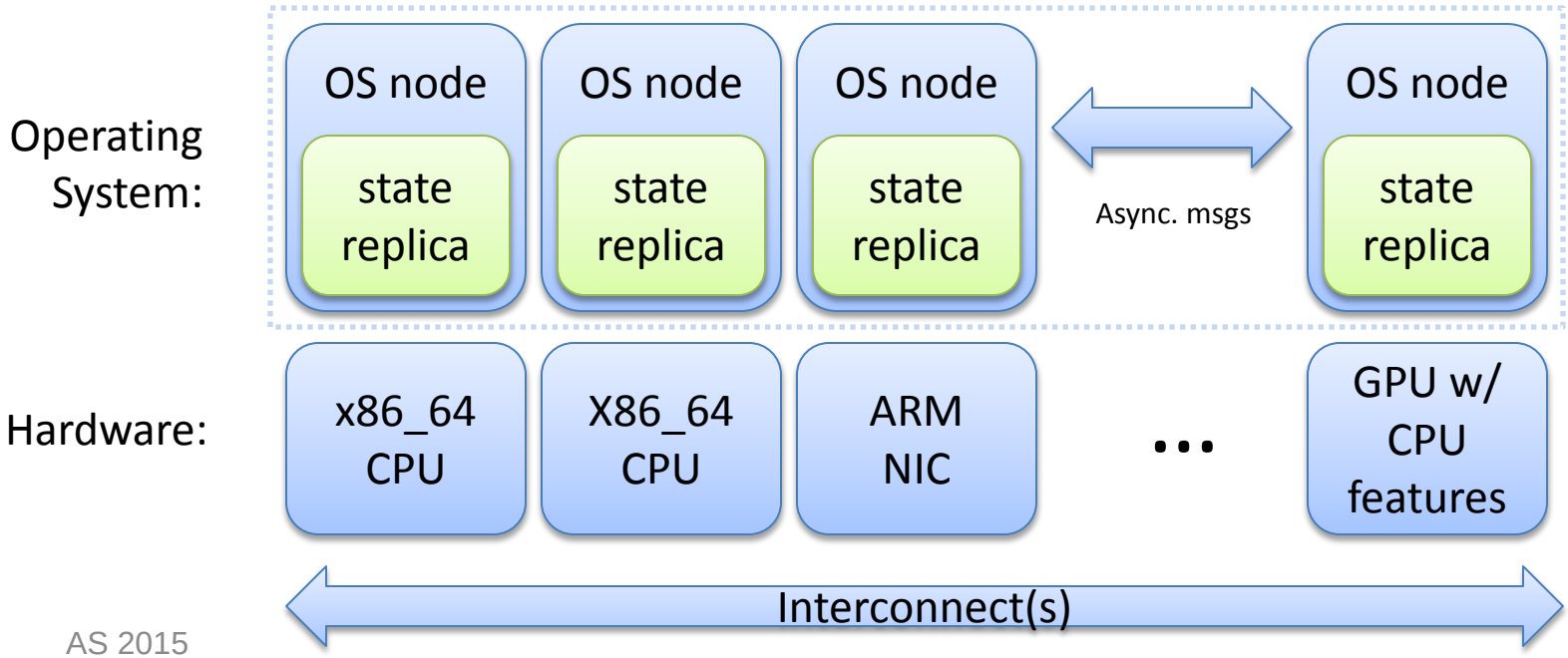
≥

≥

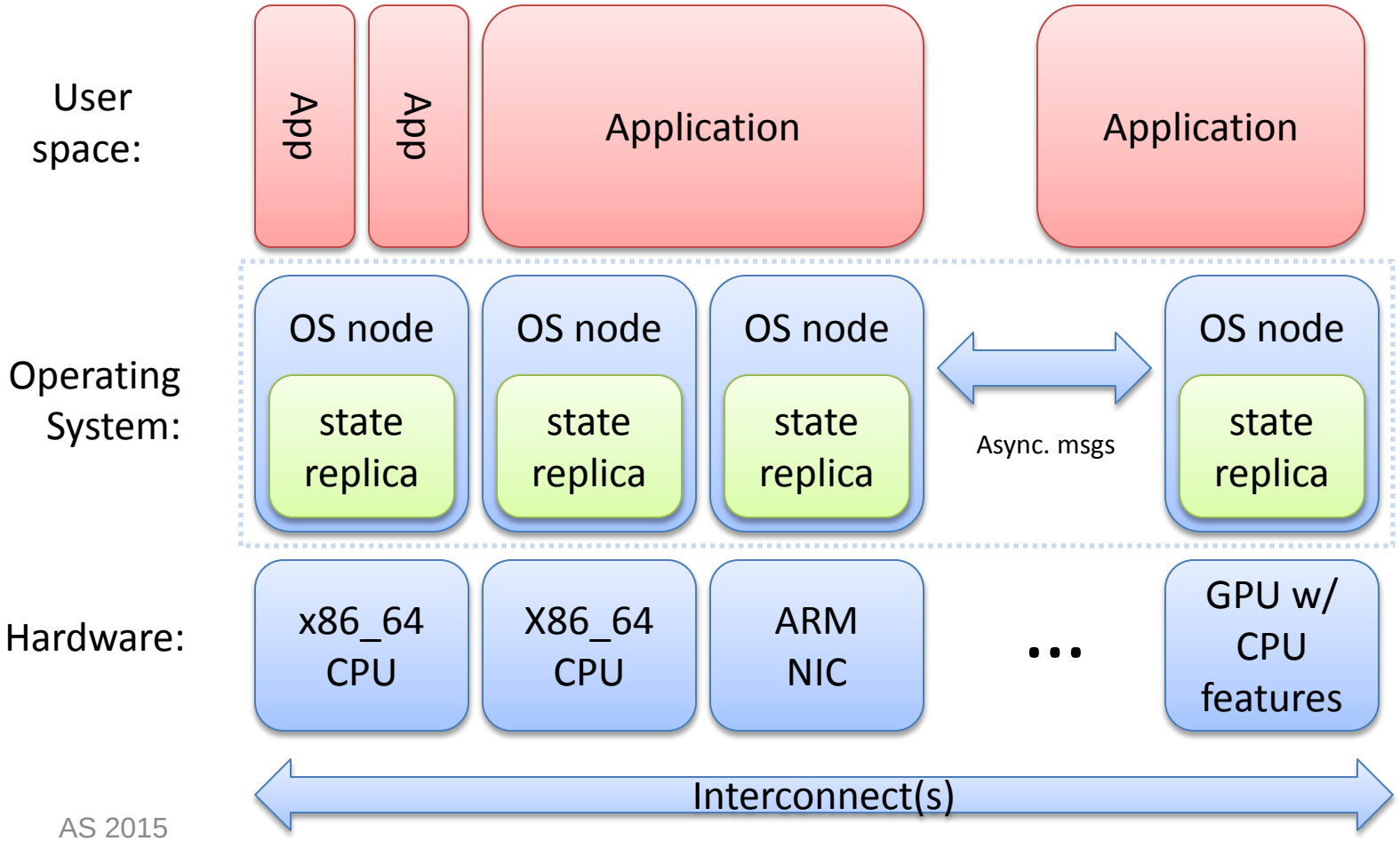
Multikernel: rethinking the OS



Multikernel: rethinking the OS



Multikernel: rethinking the OS



What does this buy us?

Systems@ETH zürich

- Variable cache-coherence or shared memory

  → non-problem
- Scalability & latency tolerance
- Dynamic core membership

  → distributed algorithms problems
- Heterogeneity of cores

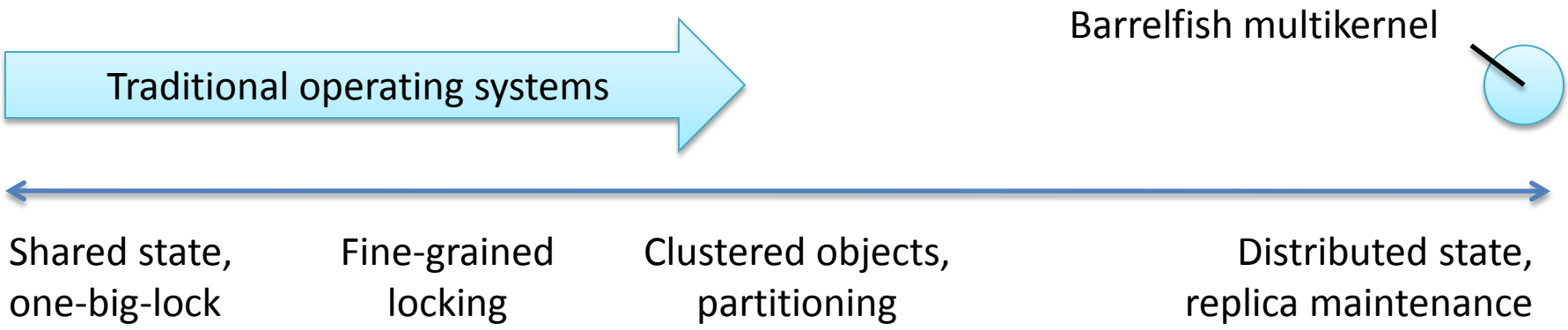
  → build-system problem

What it showed

- Performance comparable to Linux on NUMA MPs
- Can build a real general-purpose OS this way
  - Webserver: **`http://www.barrelfish.org/`**
  - Databases: SQLite, PostgreSQL, other…
  - OpenSSH server, TTY subsystem, etc.
  - Full VMM
- Support weird and wonderful platforms
  - Combined 64/32bit x86
  - Intel SCC, Xeon Phi
  - ARM Cortex A/M mixtures

!

Messaging vs shared data as default



- Fundamental model is message based
- Shared-memory is a local optimisation
  - Only when faster on a given platform
  - E.g., shared objects accessed with locking or TM

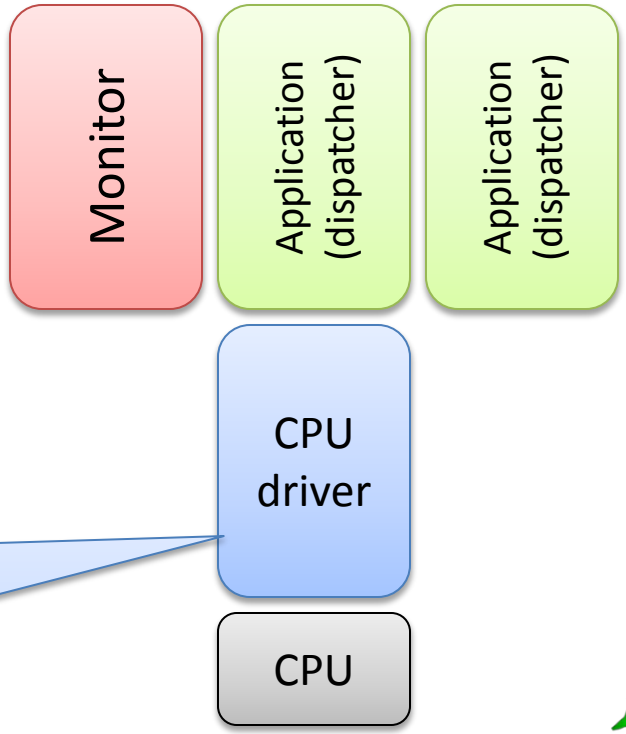
Multikernels are inevitable?

Systems@ETH zürich

- Almost every proposal looks like this
  - Corey, fos, Tesselation, akaros, Helios, etc.
- Sounds so easy: apply standard distributed system design techniques to the OS
  - Simplifies OS on a single core
- It's harder than you think:
  - Fundamental problem is qualitatively the same
  - Numbers and tradeoffs are very different
  - Shared memory throws a spanner in the works

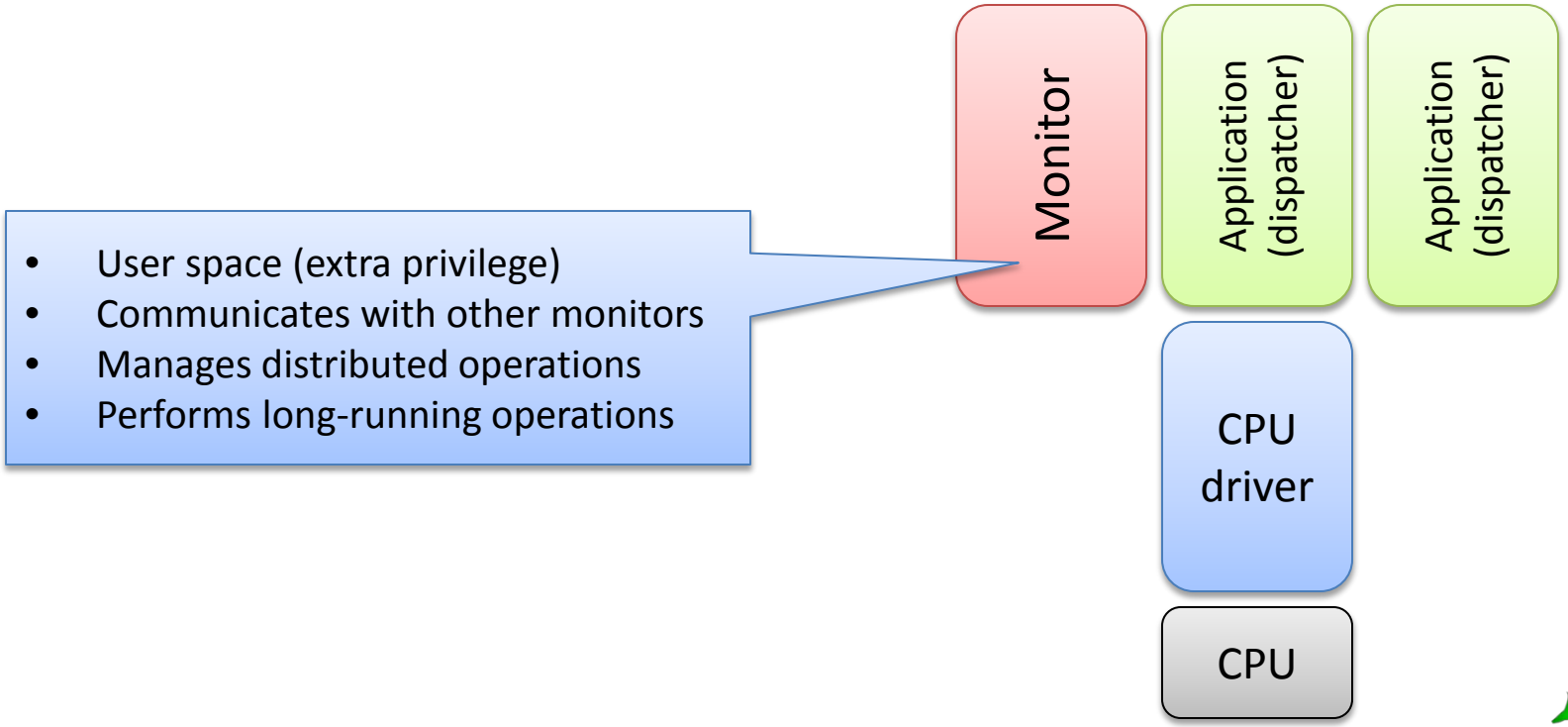
INTER-CORE MESSAGING

Barrelfish per-core architecture



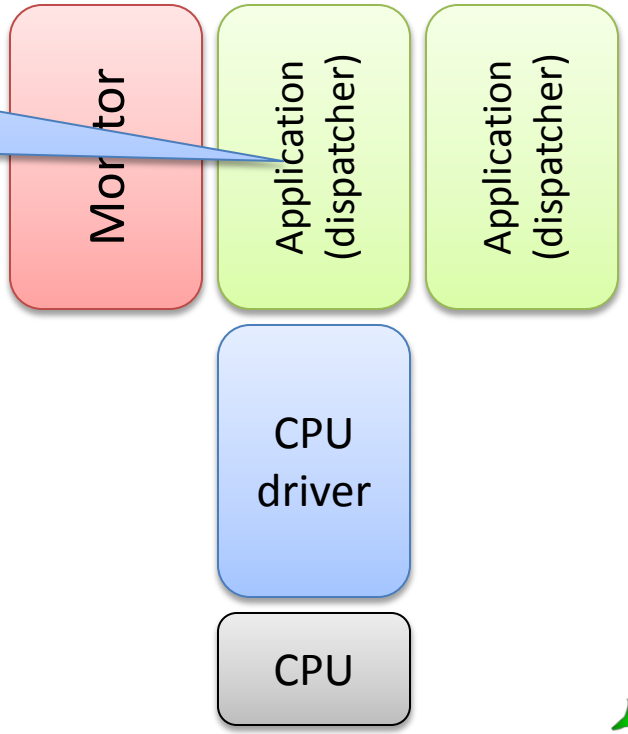
- Kernel space
- Manages CPU, MMU, APIC
- Multiplexes core btw. *dispatchers*
- Handles interrupts
- Implements protection via *capability* validation
- Single-threaded, non-preemptable

Barrelfish per-core architecture

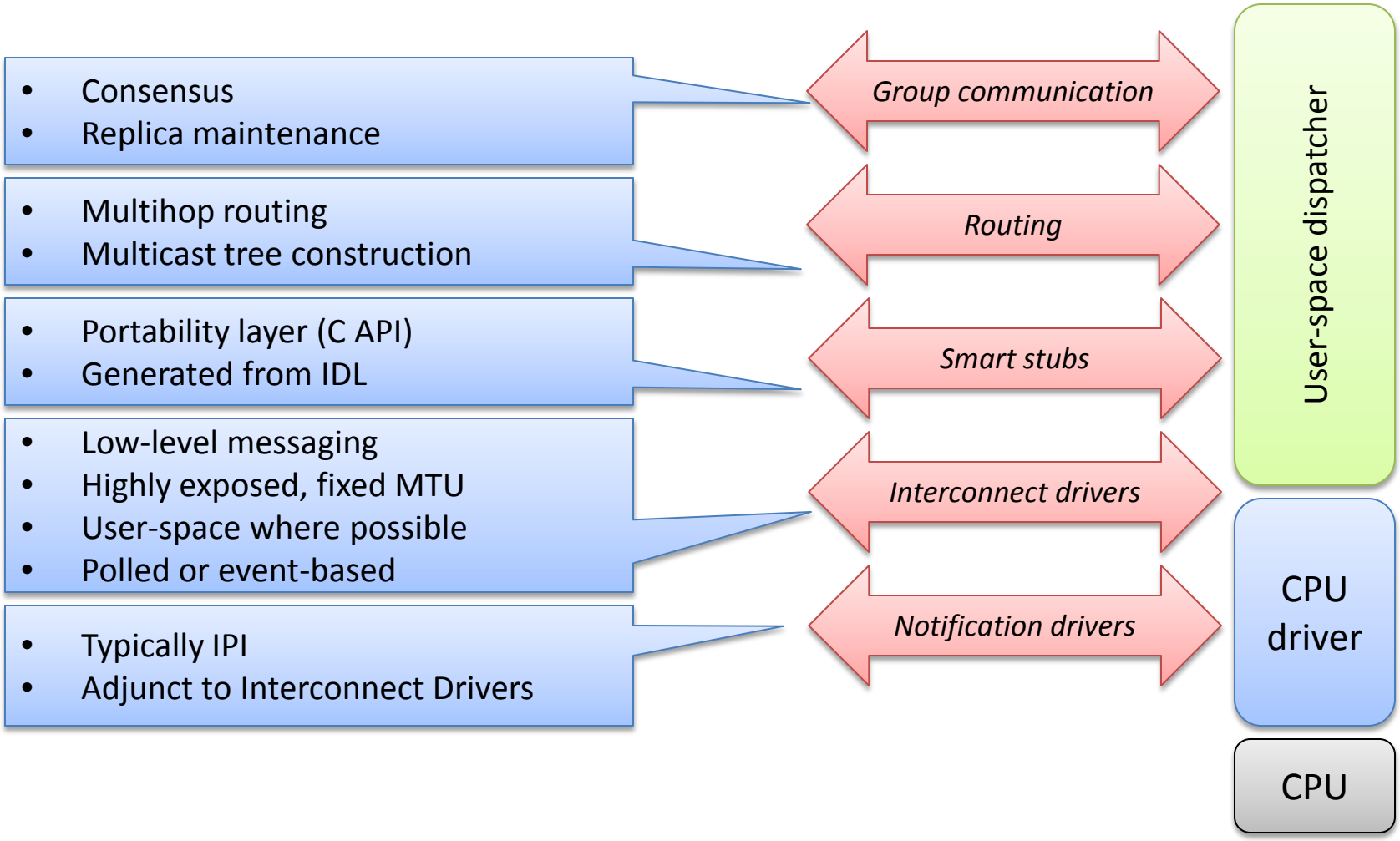


Barrelfish per-core architecture

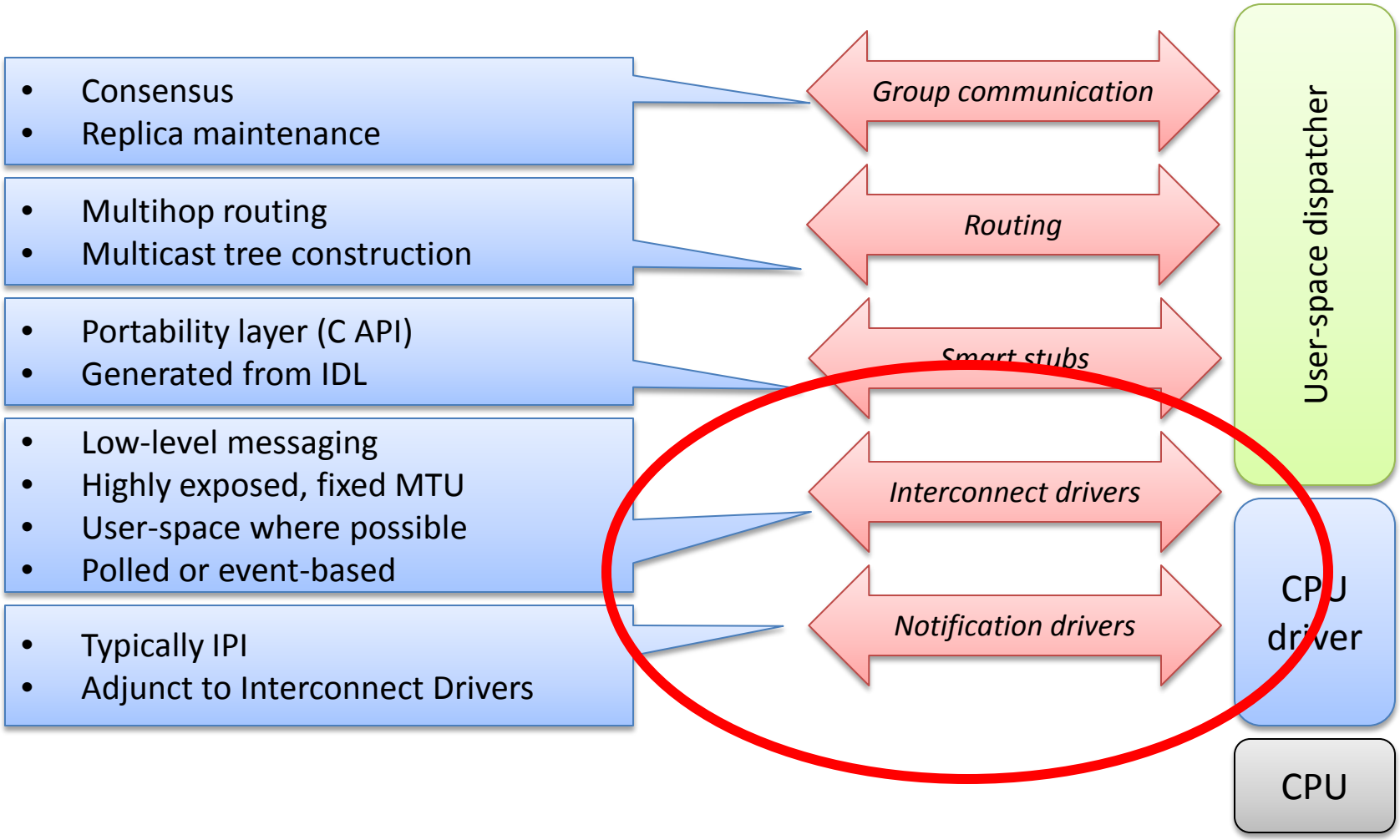
- Representative of app. on each core (including drivers and services)
- Upcalled from CPU driver
- Local thread scheduling
- Communicates with peer dispatchers



Communication stack



Barrelfish communication stack



It's a network, but not as we know it…

Systems@ETH zürich

- Lots of ways to get a message from core to core
  - Typically only one works well for a given pair of cores
- Mechanisms vary considerably
  - Multiplexed or not
  - Polled or event-based
  - MTU size
  - Privilege required (or available)

It's a network, but not as we know it…

Systems@ETH zürich

- Inter-core message latency *extremely* low
  - Ethernet in a rack: about 20μs
  - Intel SCC: about 10ns (yes, nanoseconds)
  - Shared-memory UMP: about 100ns
- Software costs *completely* dominate
  - Stub performance
  - Polling
  - Interrupts are expensive in comparison (~1μs)
  - Cache miss cost is about the same as send time

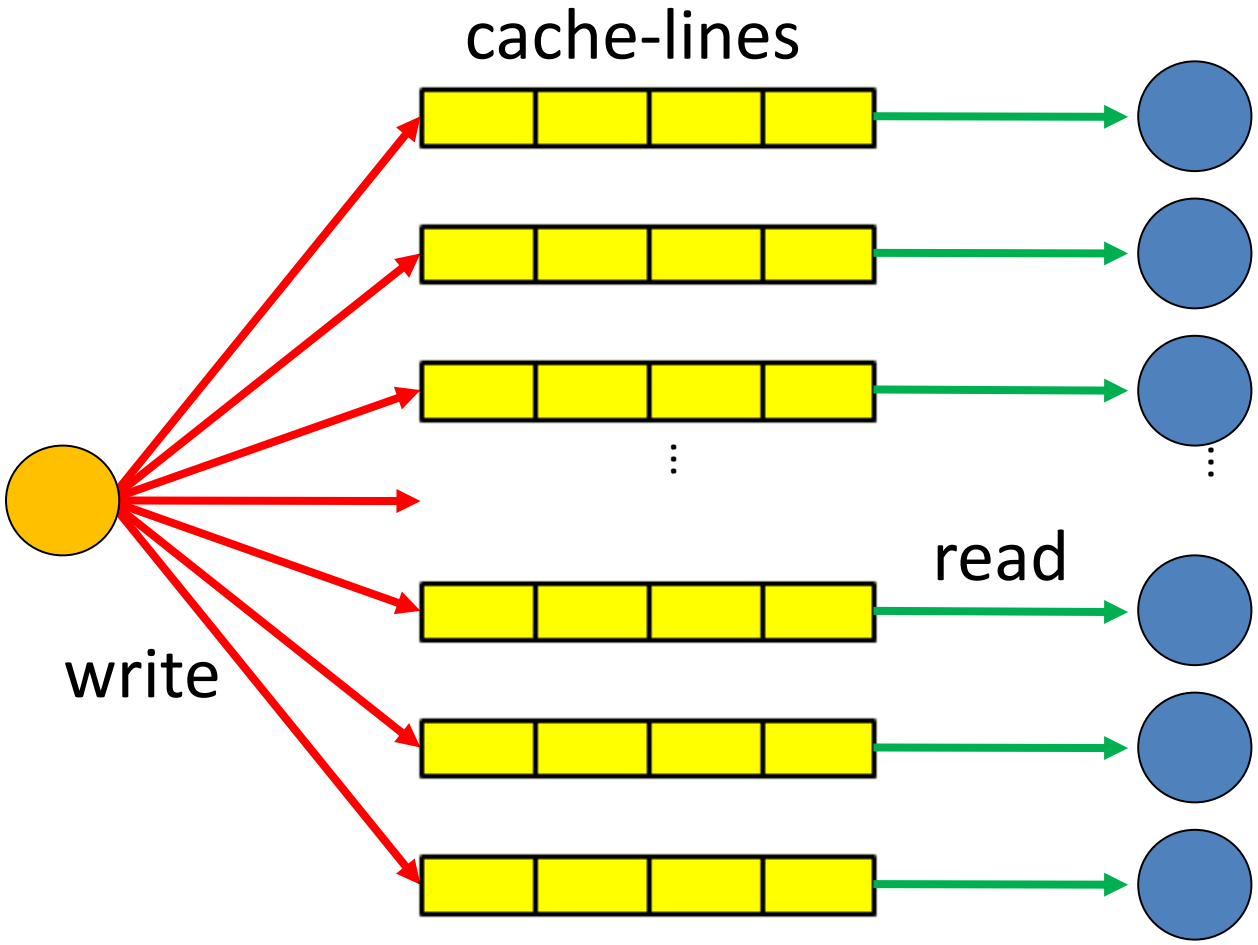
Interconnect Drivers

Systems@ETH zürich

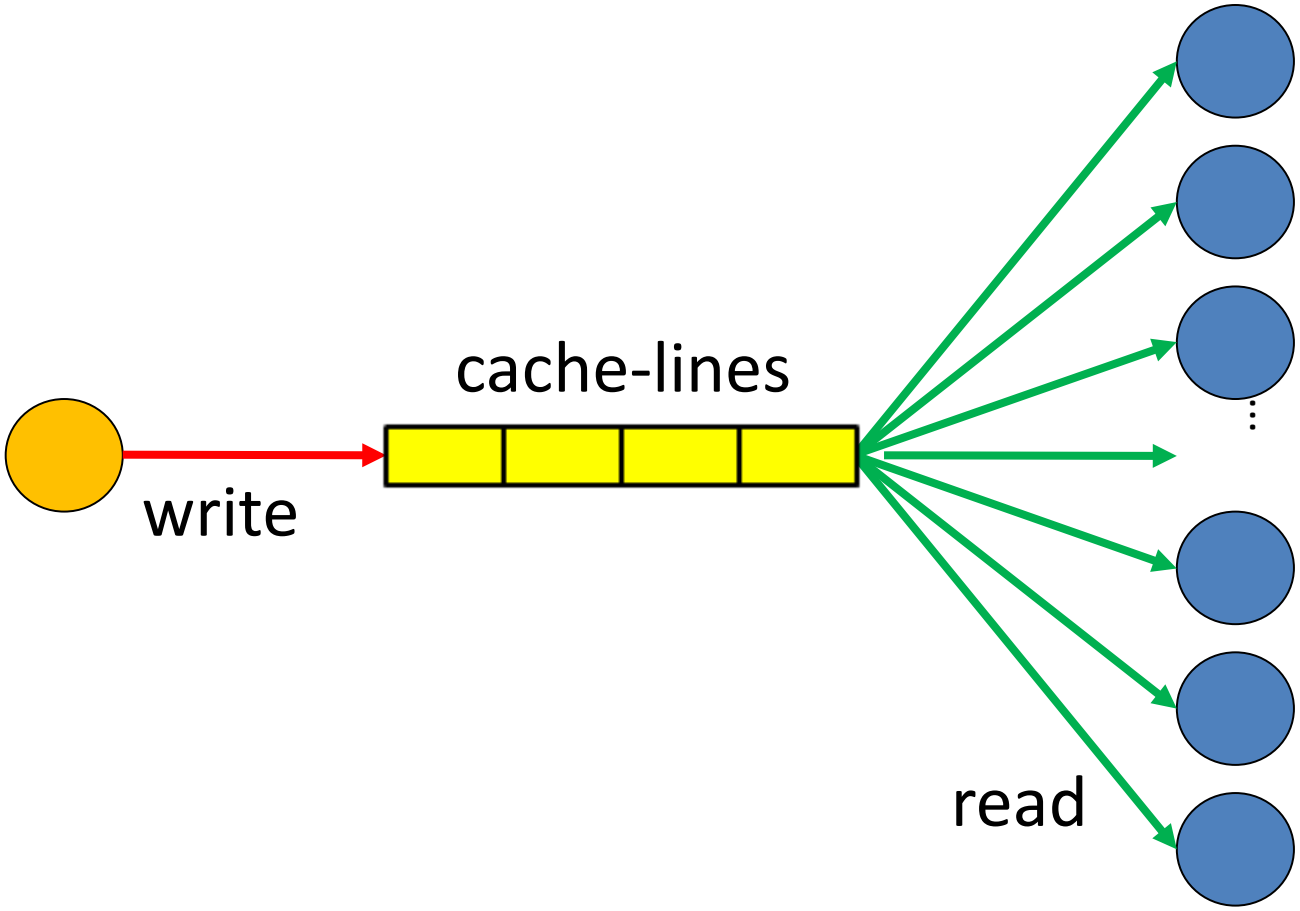
- *Barely* abstract an inter-core message channel
  - Expose message size (e.g. 64 bytes)
  - Don't implement flow control, etc. unless it's there
  - Don't require privilege, unless you have to
    - May not be able to send capabilities over this channel!
  - Separate out notification, unless it's coupled
  - Interface can be either polled or event-driven
- General philosophy:
  - Don't cover up anything
  - Expose all functionality
  - Abstract at a higher layer: in the stubs

EXAMPLES

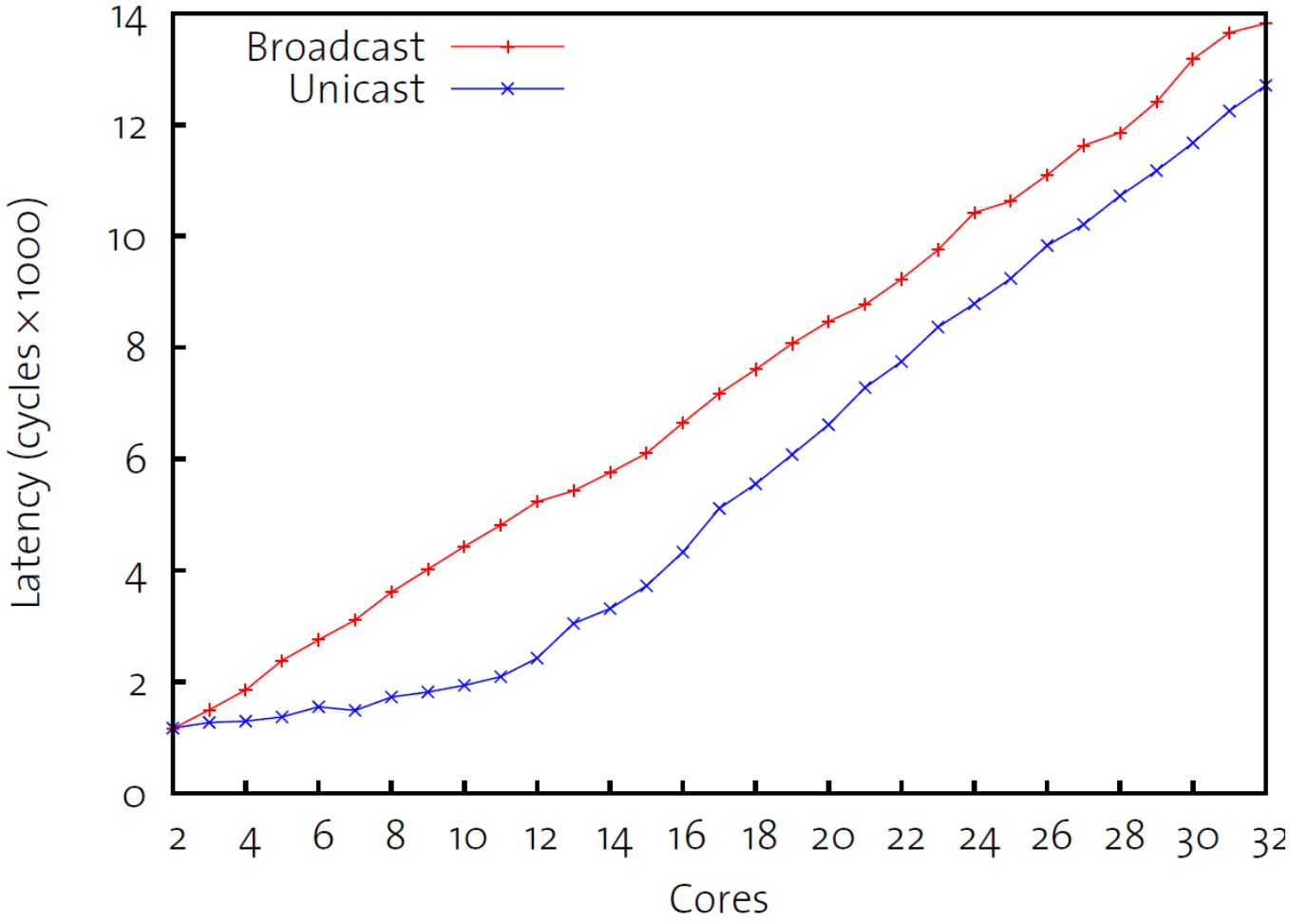
1P commit: n*unicast



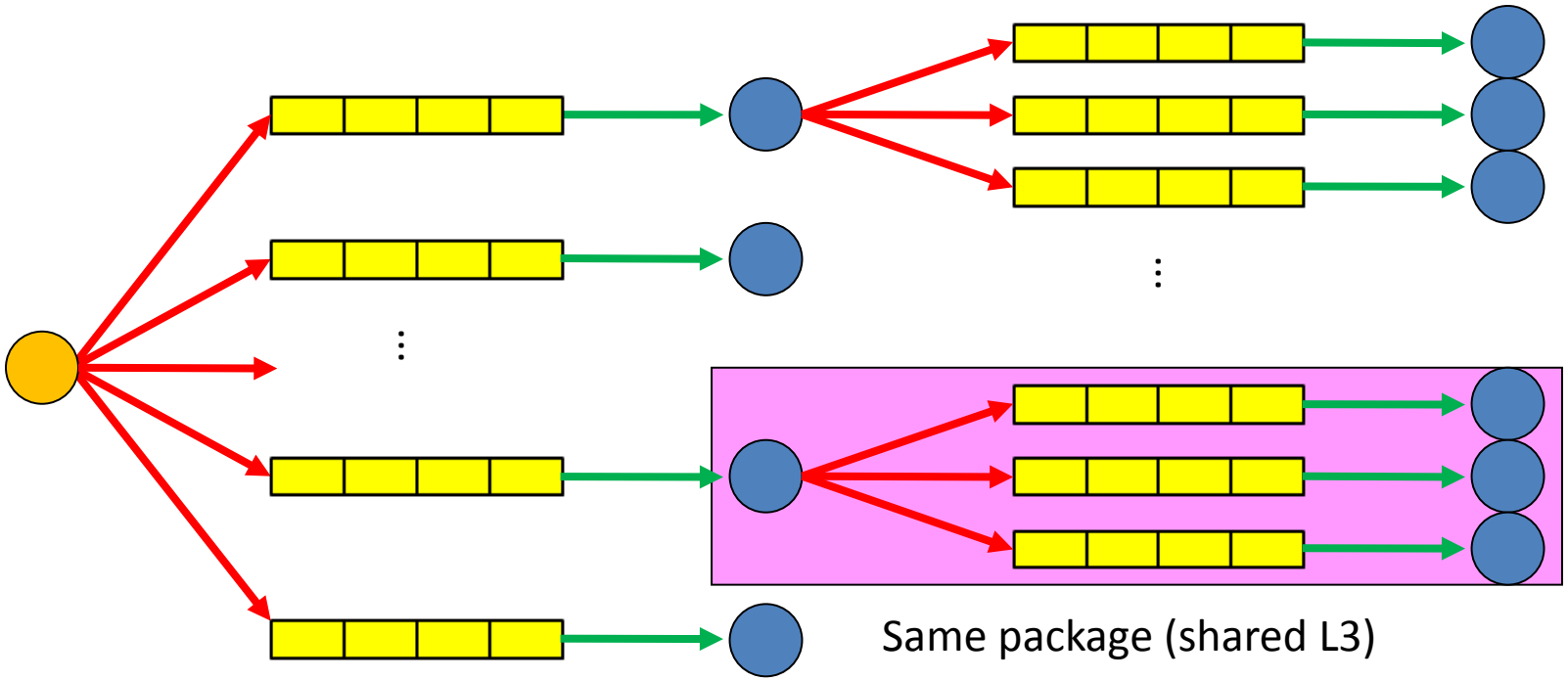
1P commit: 1*broadcast



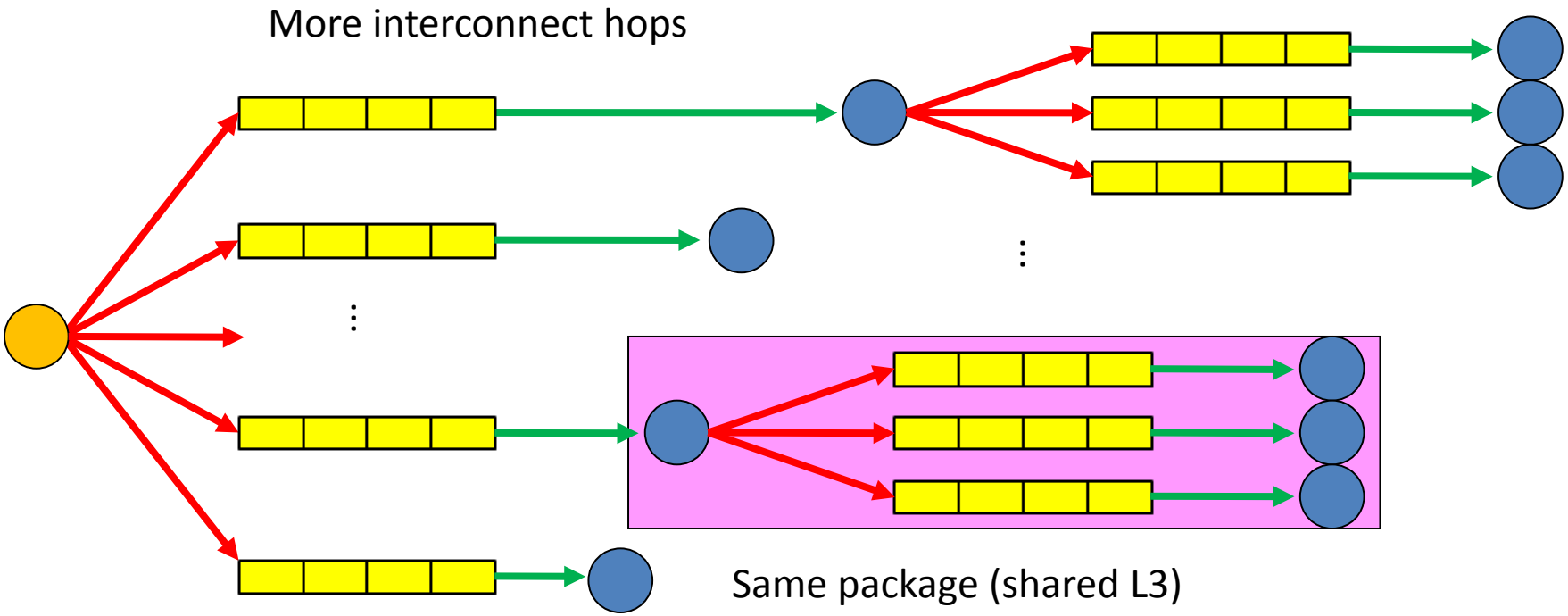
Messaging costs



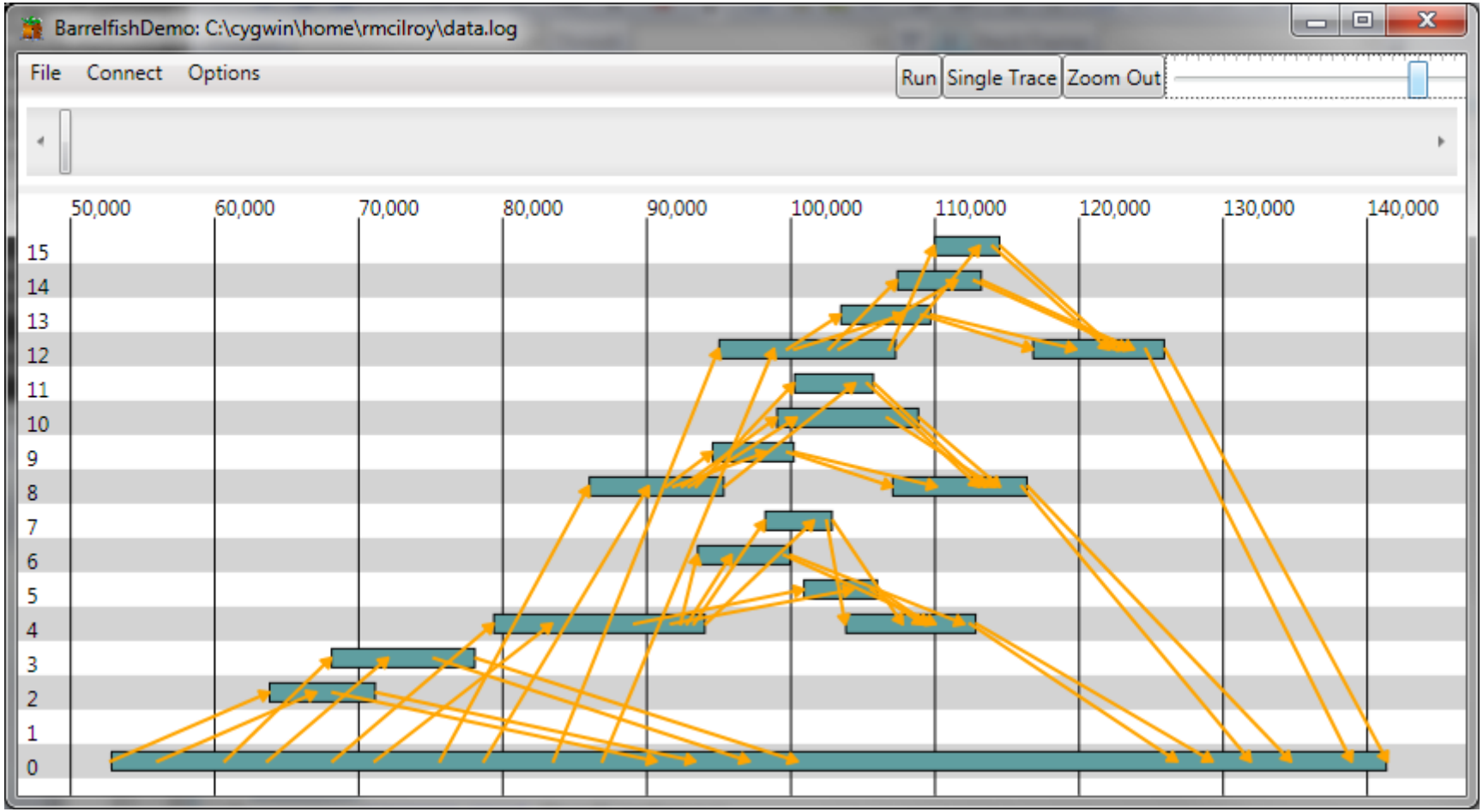
1P commit: multicast



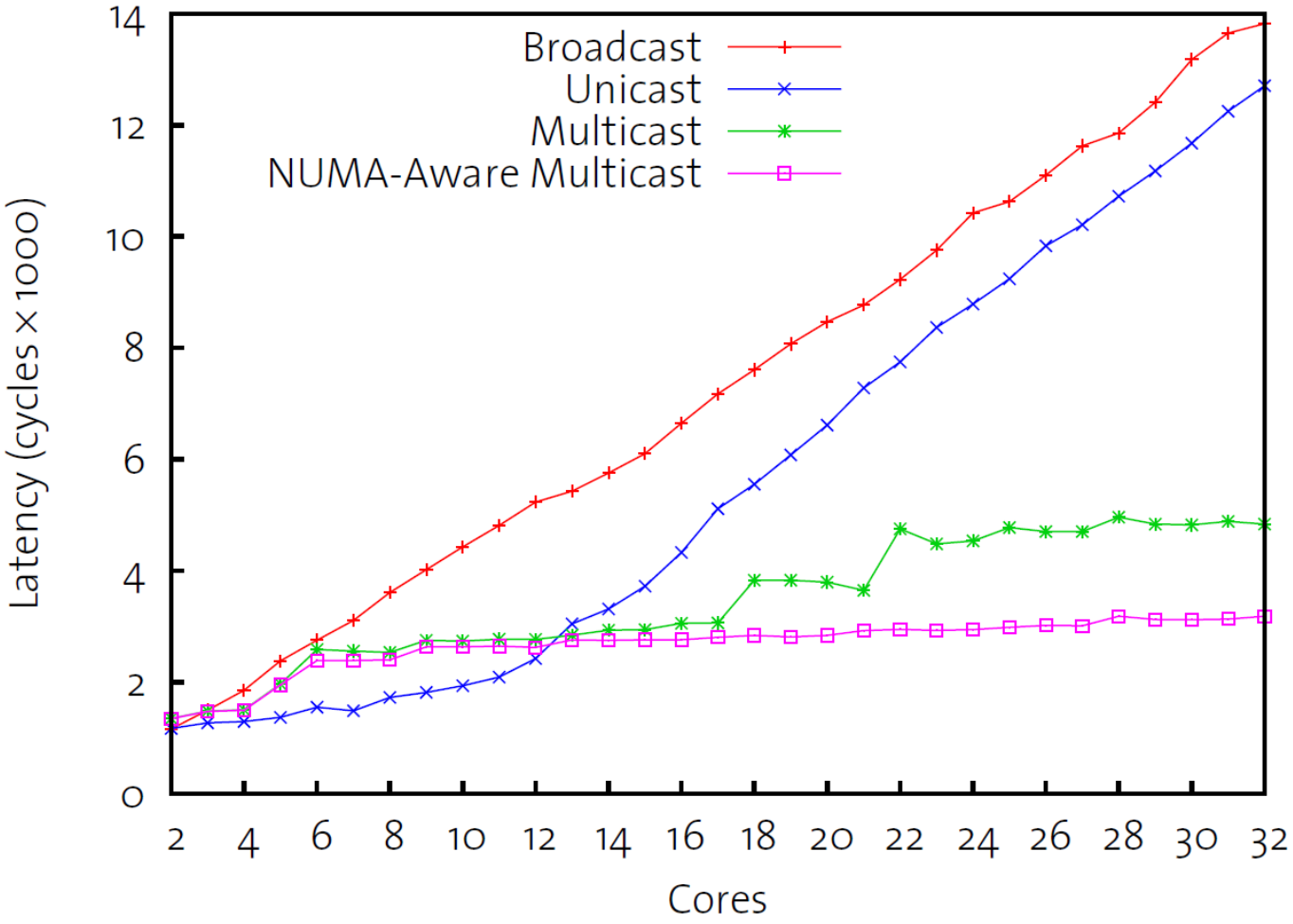
1P commit: NUMA-aware



Aggregation tree in action



Messaging costs



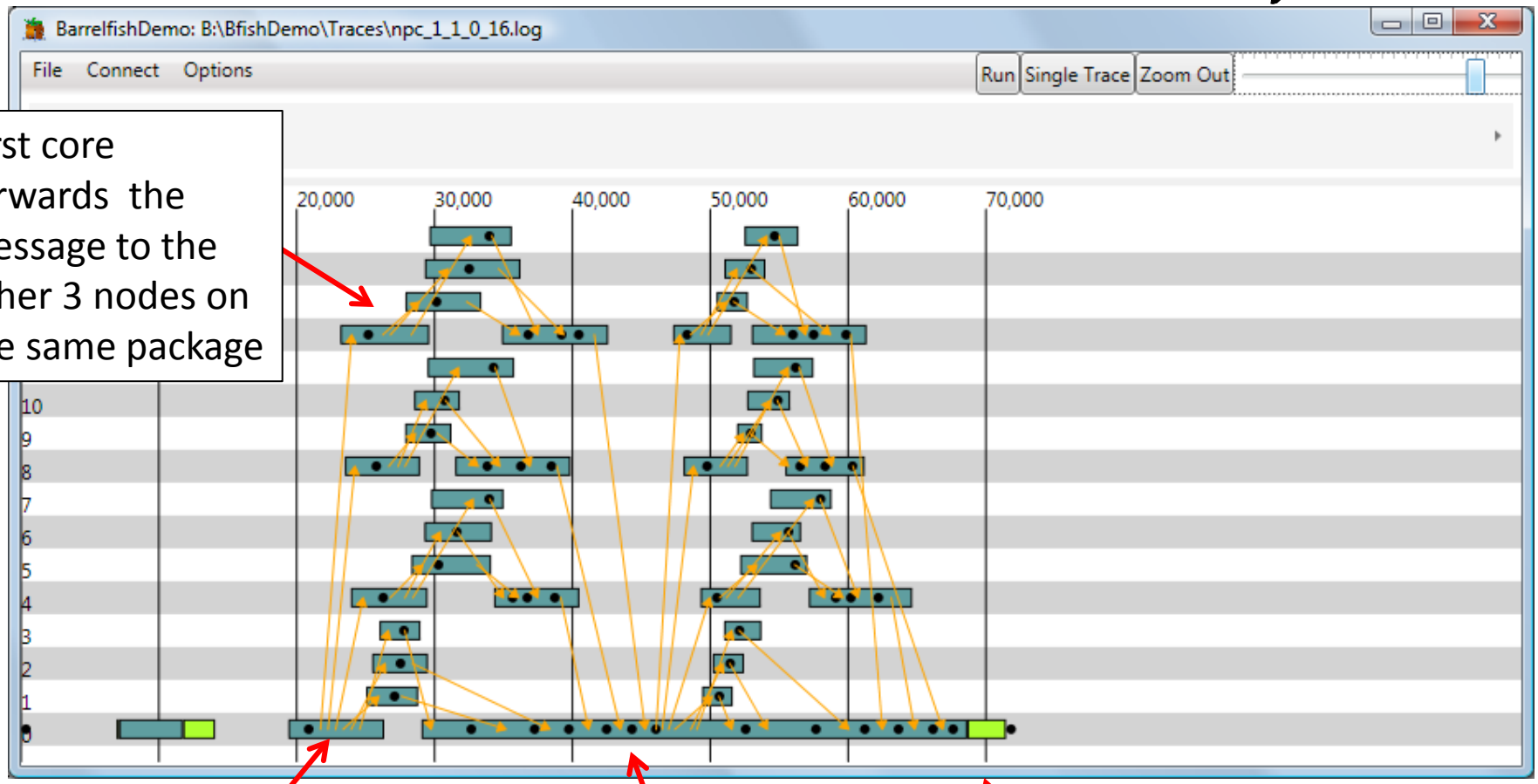
Calculate route + schedule

```
multicast_tree_cost(StartCore,[SendH|SendList], Cost) :-
    multicast_sanity_check,
    % determine package of start core
    cpu_thread(StartCore, StartPackage, _, _)
    % construct list of other packages
    findall(X, (cpu_thread(_,X,_,_), X =\= St
    filter(L,PackageList),
    % compute possible links to those package
    sends(StartCore, PackageList, SendList1),
    % compute links from start core to its ne
    sendNeighbours(StartCore, Neighbours),
    append(SendList1, Neighbours, SendList2),
    % annotate with RTT of each link
    annotate_rtt(SendList2, SendList3),
    % sort by decreasing RTT
    sort(3, >=, SendList3, [SendH|SendList]),
    % determine cost as maximum single-link RTT
    sendto(_,_,Cost) = SendH.

multicast_tree(StartCore,SendList) :-
    minimize(multicast_tree_cost(StartCore, SendList, Cost), Cost).
```

Results in very different (but generally optimal) results on different machines.

Trace: 2PC NUMA-aware multicast



First core forwards the message to the other 3 nodes on the same package

Core 0 sends a message to a core on each remote package

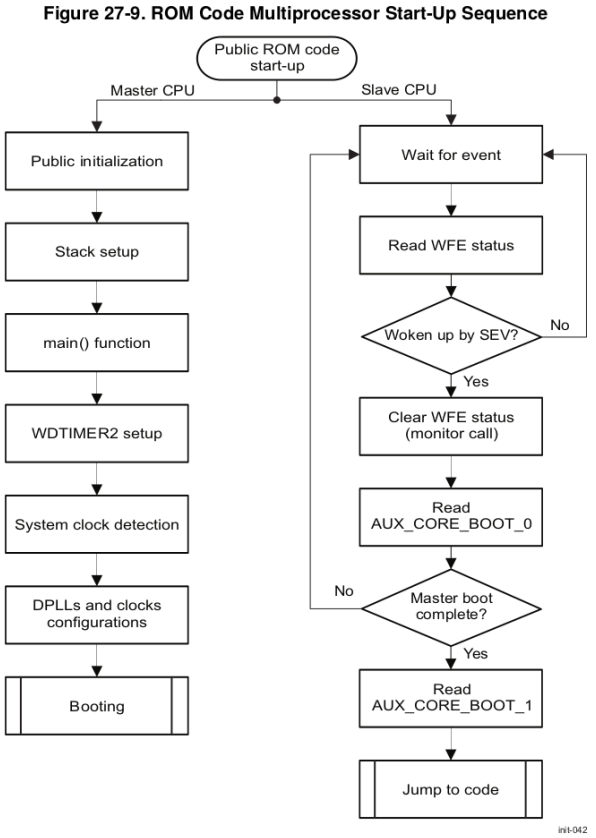
Gathers the replies

Total time ~70,000 cycles

Coreboot

Booting the OMAP4460

- Every machine is a little different.
- The basic idea is the same:
  - Provide a start address
  - Send an event
- But how do you bring a whole kernel up? Bootstrap allocators and so on?



Booting cores is *Common*

Systems@ETH zürich

- Starting and stopping cores for power management is continuous, but *applications* are long-lived.
- In Barrelfish, applications (user code) spawn new *kernels* (CPU drivers) as required.
- Mostly implemented at *privileged* user level.

Corebooting on ARM

1) Allocate memory.
2) Create KCB/Coredata.
3) Load CPU driver.
4) Clean the cache.
5) Call spawn_core

- Remember, on BF, the caller supplies the RAM:
  - Kernel .bss
  - RAM for init
  - RAM for URPC
  - RAM to retype into a KCB.

Corebooting on ARM

1) Allocate memory.
2) Create KCB/Coredata.
3) Load CPU driver.
4) Clean the cache.
5) Call spawn_core

- The KCB object holds the running kernel's state.
- Coredata provides the boot parameters(-ish).
- kcb_clone() copies the running kernel – you'll use this.
- Update coredata with changed parameters e.g. RAM region to use.

Corebooting on ARM

Systems@ETH zürich

1) Allocate memory.
2) Create KCB/Coredata.
3) Load CPU driver.
4) Clean the cache.
5) Call spawn_core

- Load a *new* copy of the kernel into RAM – why?
  - Has its own state.
- If *cloning*, can reuse the text segment (it's *PIC*).
  - Just load the *relocatable segment* – contains .bss.
  - We'll provide some support code for ELF relocation.

Corebooting on ARM

Systems@ETH zürich

1) Allocate memory.
2) Create KCB/Coredata.
3) Load CPU driver.
4) Clean the cache.
5) Call spawn_core

- The kernel starts with MMU uninitialised.
- It's using *uncached* accesses.
- Make sure everything it needs is in RAM:
  - Do a *clean* (more shortly)

Corebooting on ARM

1) Allocate memory.
2) Create KCB/Coredata.
3) Load CPU driver.
4) Clean the cache.
5) Call spawn_core

- A kernel cap invocation.
- Uses the platform-specific boot protocol to start the core.

Corebooting on ARM

Systems@ETH zürich

- Now you’re on your own:
  - You’ll start executing whatever binary you passed as the *monitor* in coredata.
  - You need to figure out how to set up a channel to your initial core, and pass RPCs.