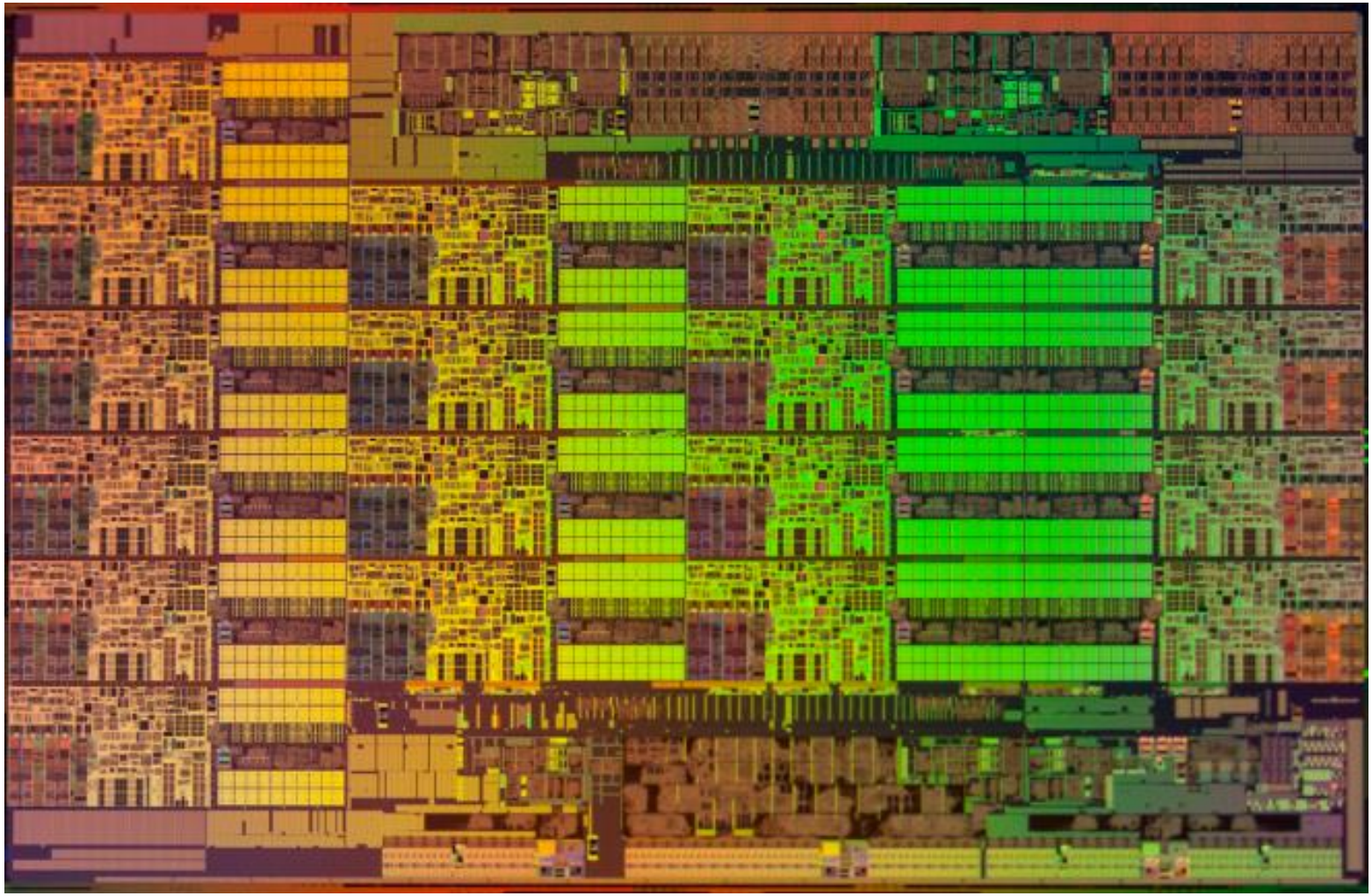
Chapter 6

Parallel Processors from Client to Cloud – Part 2

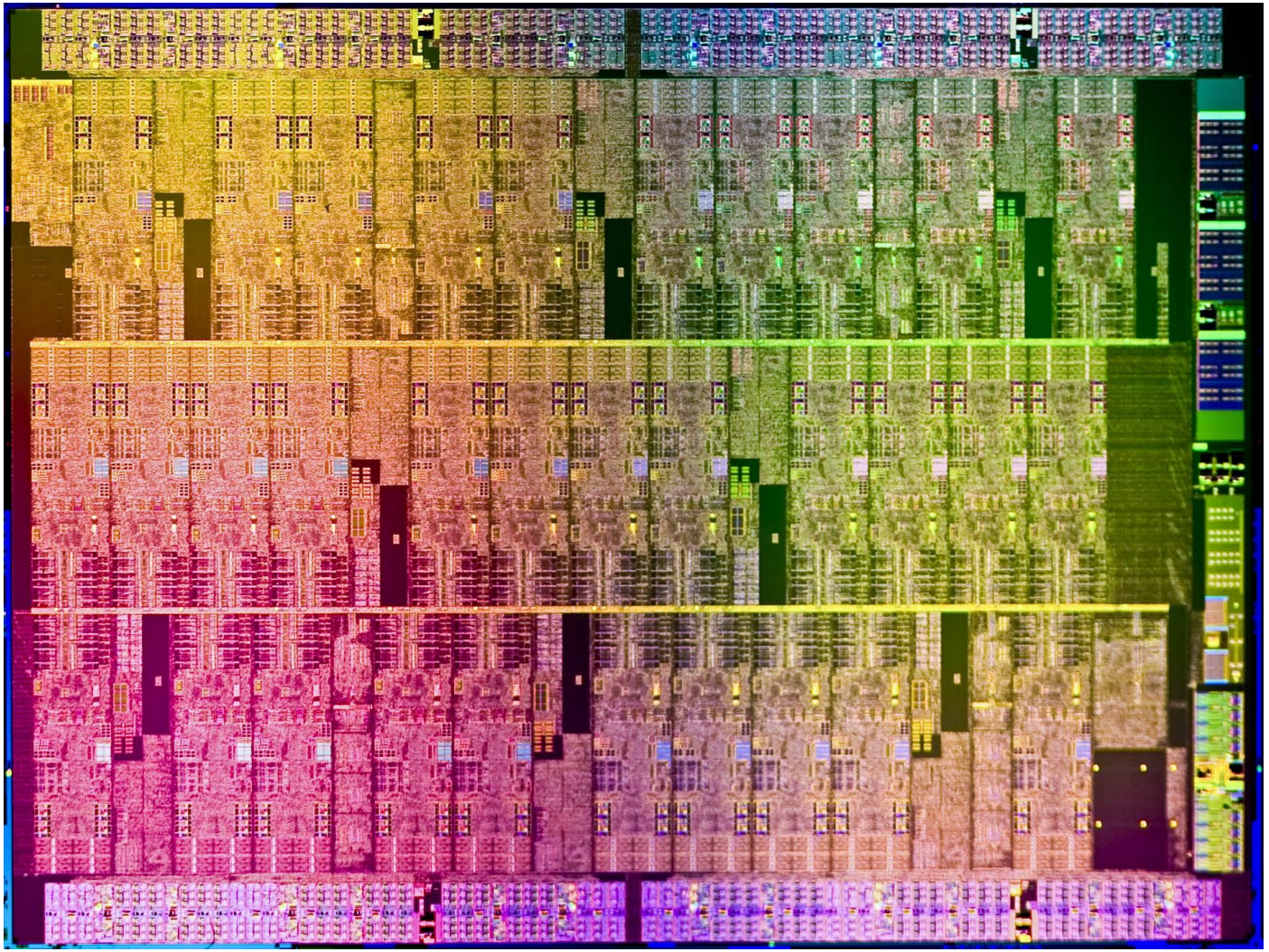
Homogeneous & Heterogeneous Multicore Architectures

Intel XEON 22nm Server



Xeon E5-2699 v3 flagship CPU offers 18 cores and 36 threads plus a 45 MB L3 cache

Intel's Xeon Phi 22nm

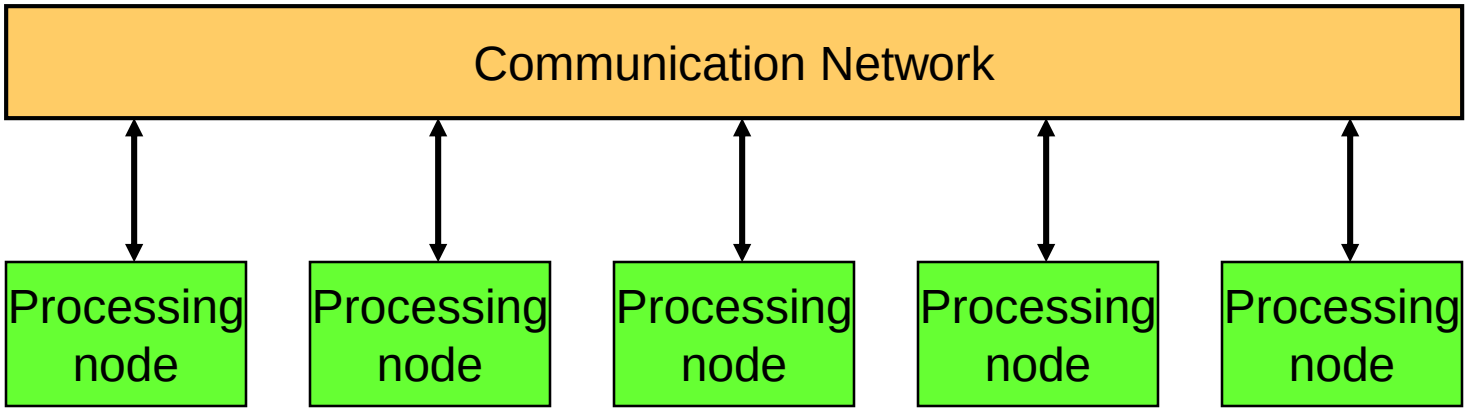


Introduction

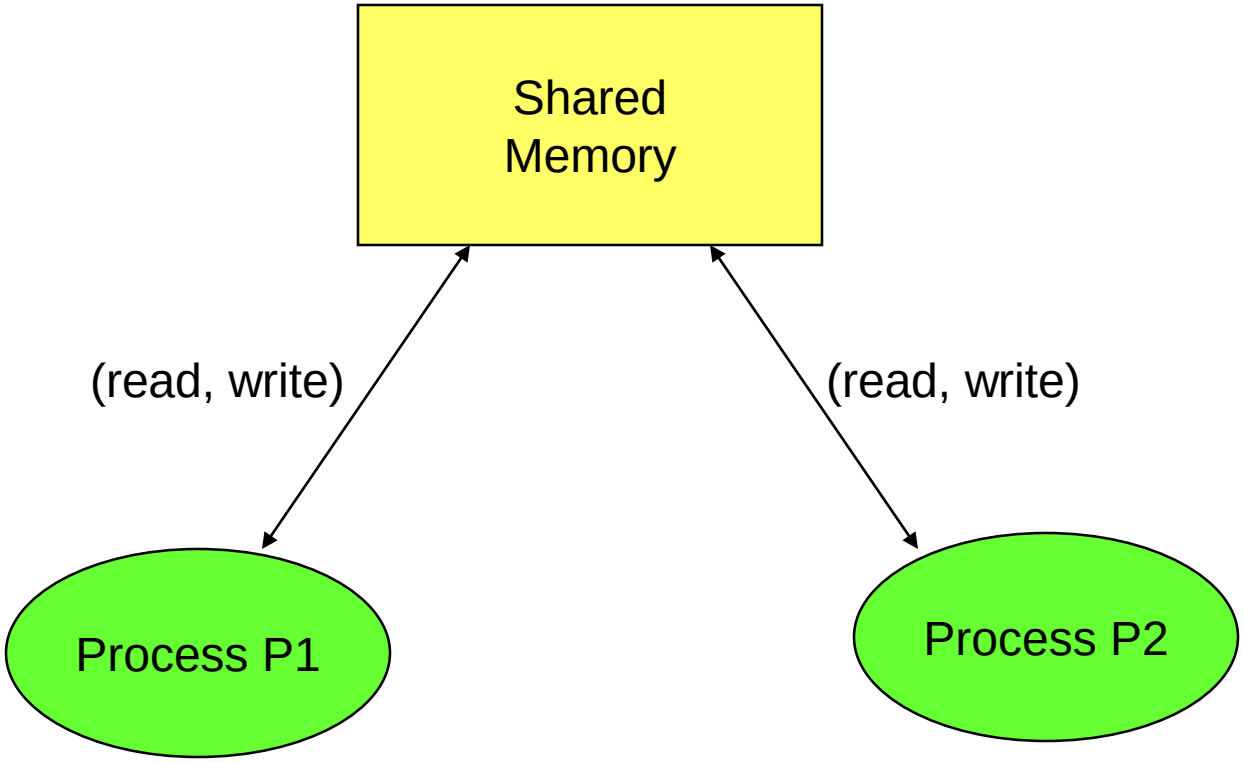
- Thread-Level parallelism
  - Have multiple program counters
  - Uses MIMD model
  - Targeted for tightly-coupled shared-memory multiprocessors
- For *n* processors, need *n* threads
- Amount of computation assigned to each thread = grain size
  - Threads can be used for data-level parallelism, but the overheads may outweigh the benefit

Parallel Architecture

- Parallel Architecture extends traditional computer architecture with a communication network
  - abstractions (HW/SW interface)
  - organizational structure to realize abstraction efficiently



Communication models: Shared Memory



- Coherence problem
- Memory consistency issue
- Synchronization problem

Communication models: **Shared memory**

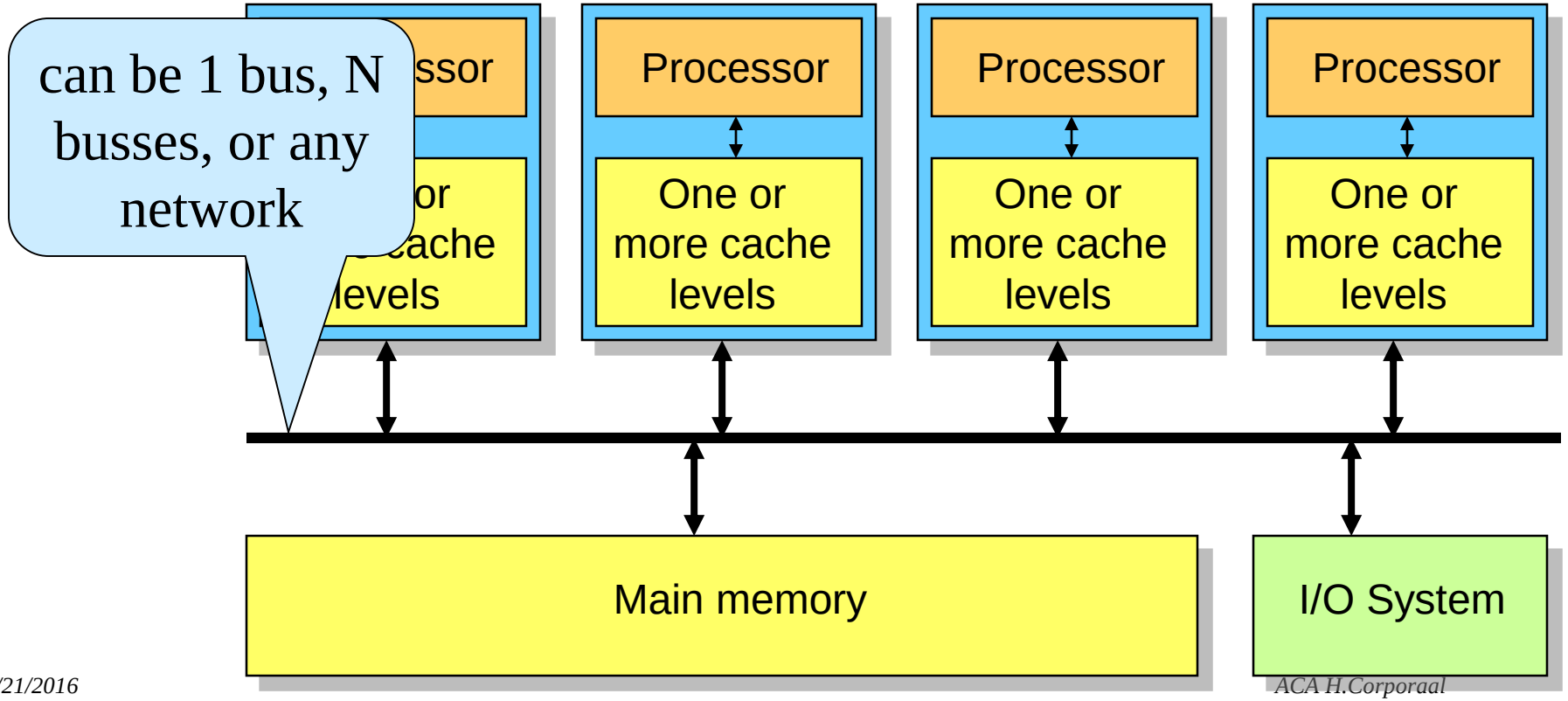
- Shared address space
- Communication primitives:
  - load, store, atomic swap

Two varieties:

- Physically shared => Symmetric Multi-Processors (**SMP**)
  - usually combined with local caching
- Physically distributed => Distributed Shared Memory (**DSM**)

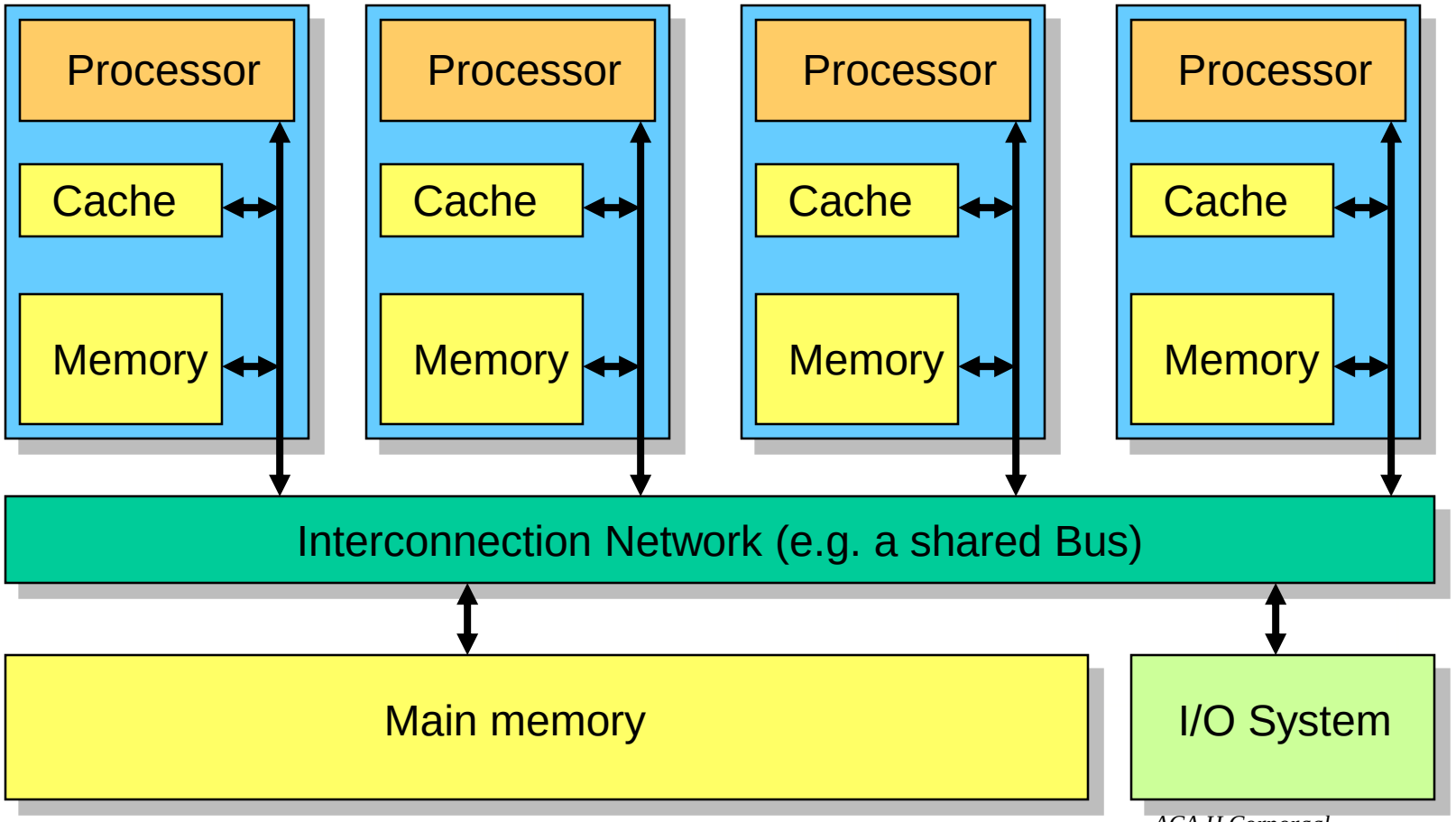
SMP: Symmetric Multi-Processor

- Memory: centralized with uniform access time (***UMA***) and bus interconnect, I/O
- Examples: Sun Enterprise 6000, SGI Challenge, Intel



DSM: Distributed Shared Memory

- Nonuniform access time (***NUMA***) and scalable interconnect (distributed memory)

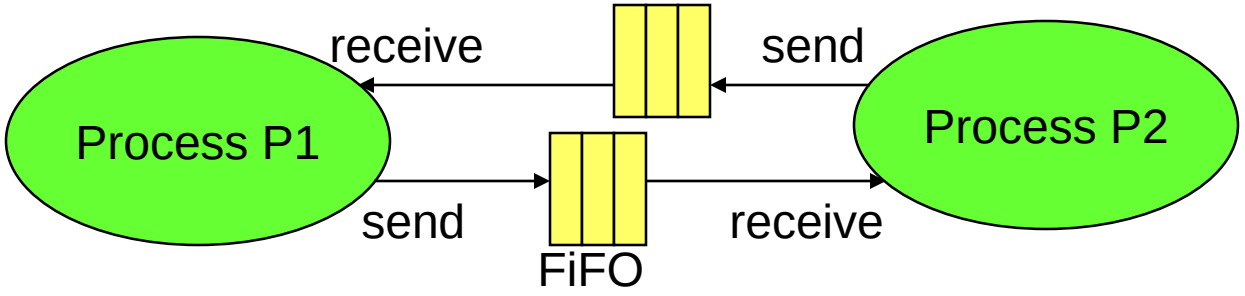


Shared Address Model Summary

- Each processor can address every physical location in the machine
- Each process can address all data it shares with other processes
- Data transfer via load and store
- Data size: byte, word, ... or cache blocks
- Memory hierarchy model applies:
  - communication moves data to local proc. cache

Communication models: **Message Passing**

- Communication primitives
  - e.g., send, receive library calls
  - standard MPI: Message Passing Interface
    - www.mpi-forum.org
- *Note that MP can be build on top of SM and vice versa !*

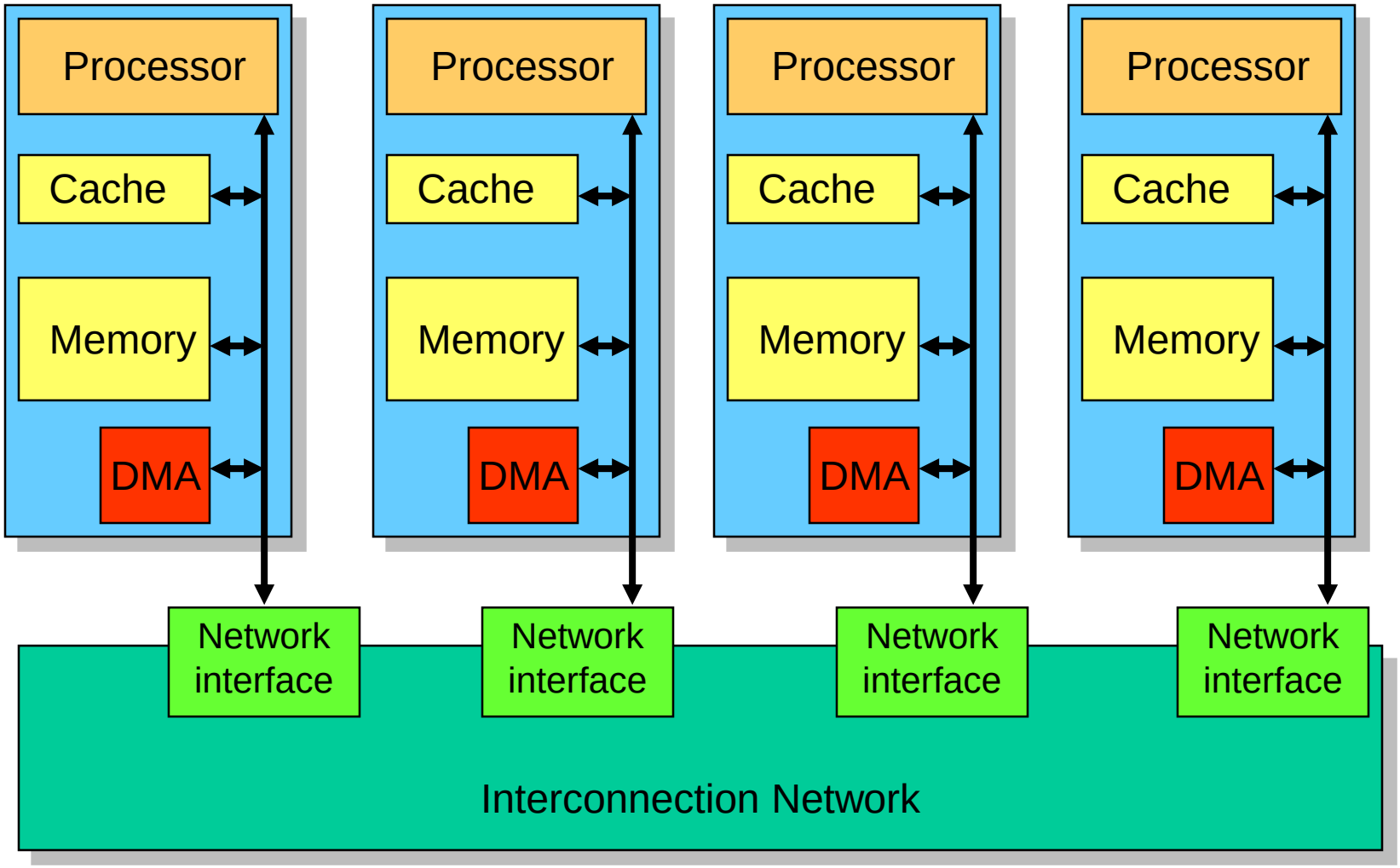


Message Passing Model

- Explicit message send and receive operations
- Send specifies local buffer + receiving process on remote computer
- Receive specifies sending process on remote computer + local buffer to place data
- Typically blocking communication, but may use DMA

Message structure

Message passing communication



Communication Models: Comparison

- Shared-Memory (used by e.g. **OpenMP**)
  - Compatibility with well-understood (language) mechanisms
  - Ease of programming for complex or dynamic communications patterns
  - Shared-memory applications; sharing of large data structures
  - Efficient for small items
  - Supports hardware caching
- Messaging Passing (used by e.g. **MPI**)
  - Simpler hardware
  - Explicit communication
  - Implicit synchronization (with any communication)

Challenges of parallel processing

Q1: can we get linear speedup

Suppose we want speedup 80 with 100 processors.
What fraction of the original computation can be sequential (i.e. non-parallel)?

Answer: fseq = 0.25%

Q2: how important is communication latency

Suppose 0.2 % of all accesses are remote, and require 100 cycles on a processor with base CPI = 0.5

What's the communication impact?

Network: Performance metrics

- Network Bandwidth
  - Need high bandwidth in communication
  - How does it scale with number of nodes?
- Communication Latency
  - Affects performance, since processor may have to wait
  - Affects ease of programming, since it requires more thought to overlap communication and computation

How can a mechanism help hide latency?

- overlap message send with computation,
- prefetch data,
- switch to other task or thread
- pipelining tasks, or pipelining iterations

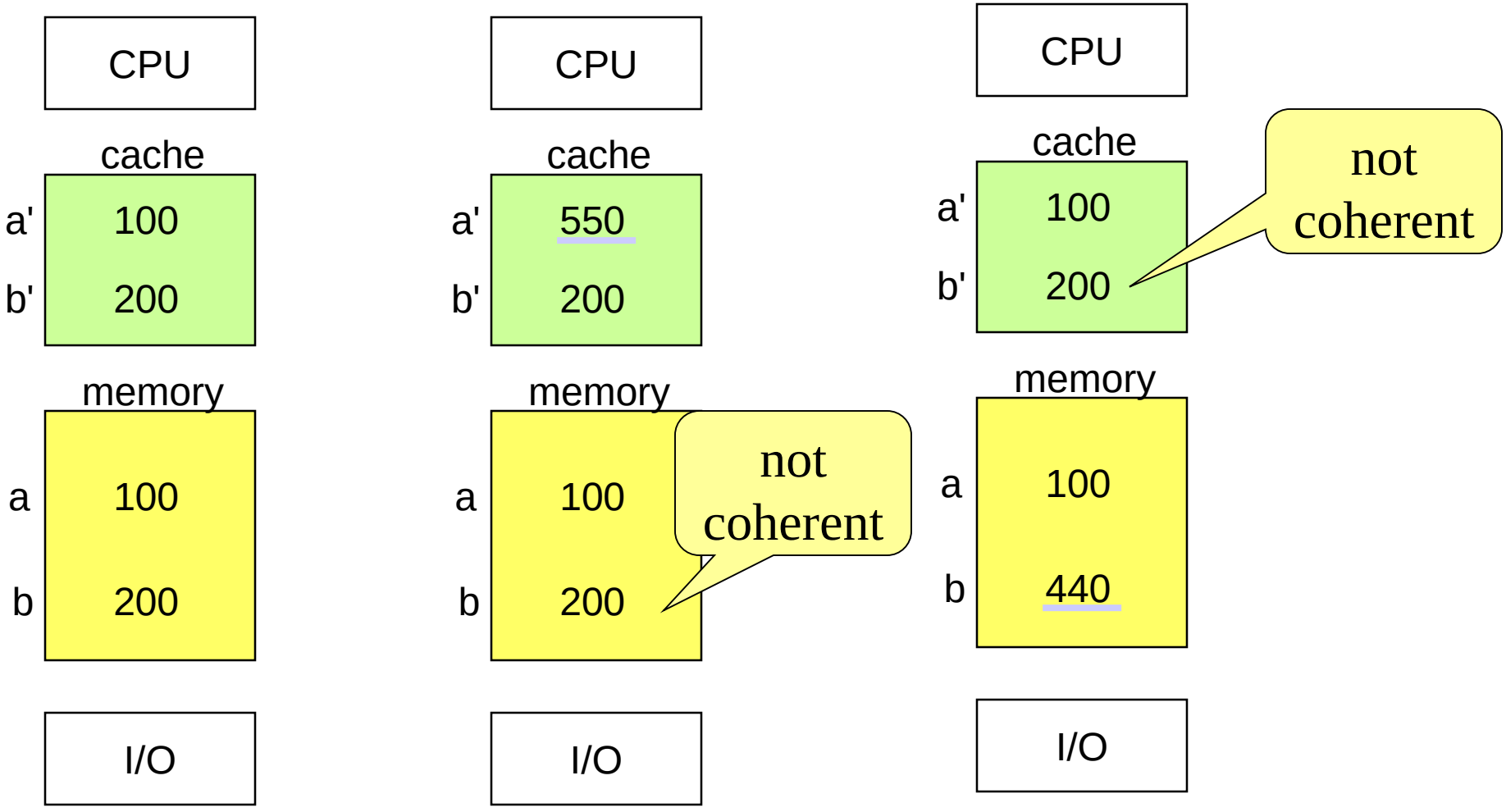
Three fundamental issues for shared memory multiprocessors

- **Coherence**,
  about: *Do I see the most recent data?*

- **Consistency**,
  about: ***When** do I see a written value?*
  - e.g. do different processors see writes at the same time (w.r.t. other memory accesses)?

- **Synchronization**
  How to synchronize processes?
  - how to protect access to shared data?

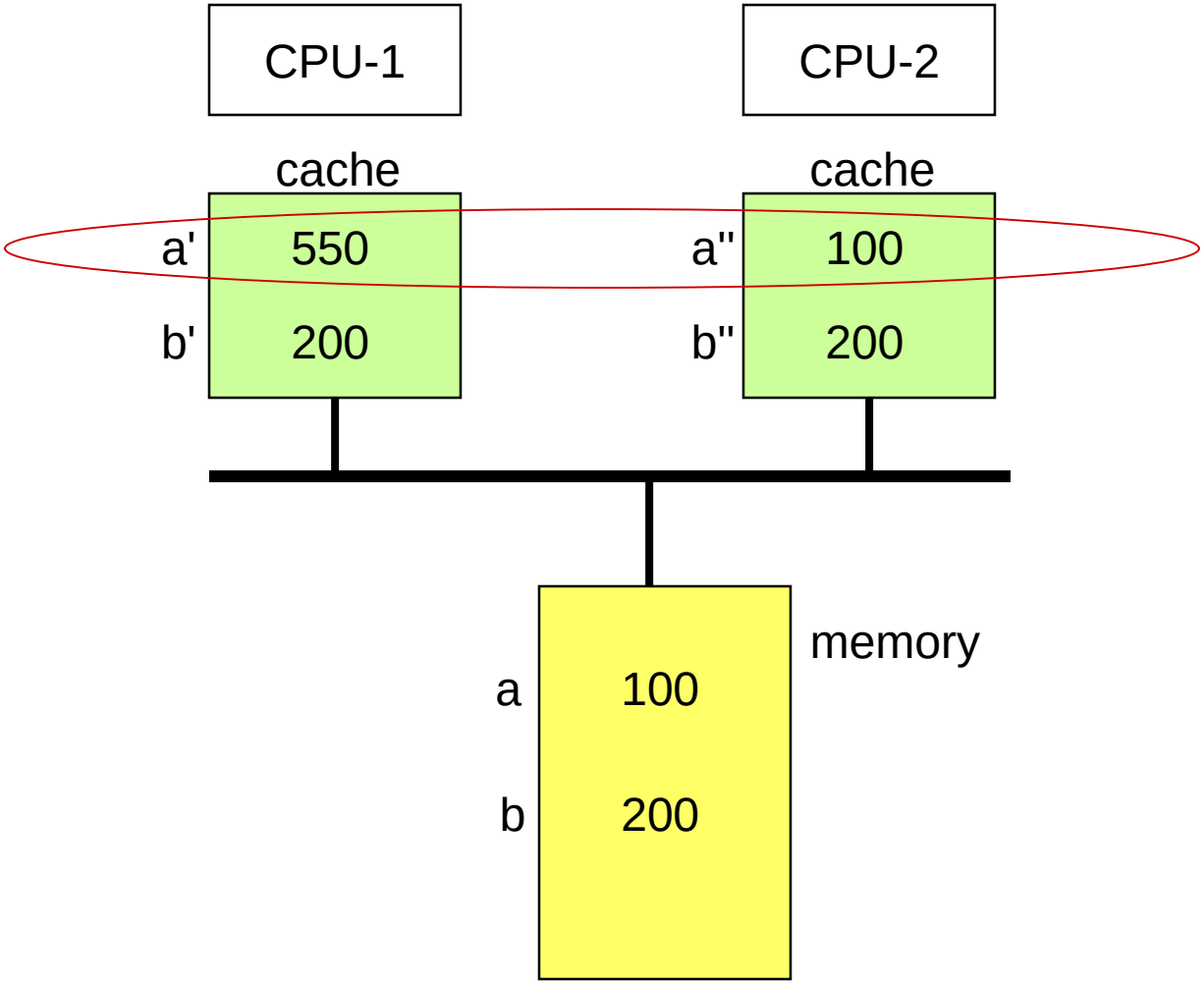
Coherence problem, in single CPU system



1) CPU writes to a

2) IO writes b

Coherence problem, in Multi-Proc system



What Does Coherency Mean?

- Informally:
  - "Any read must return the most recent write (to the same address)"
  - Too strict and too difficult to implement
- Better:
  - A write followed by a read by the same processor P with no writes in between returns the value written
  - "Any write must eventually be seen by a read"
    - If P writes to X and P' reads X then P' will see the value written by P if the read and write are *sufficiently separated in time*
  - Writes to the same location by different processors are seen *in the same order* by all processors ("serialization")
    - Suppose P1 writes location X, followed by P2 writing also to X. If no serialization, then some processors would read value of P1 and others of P2. Serialization guarantees that all processors read the same sequence.

Two rules to ensure coherency

- "If P1 writes x and P2 reads it, P1's write will be seen by P2 if the read and write are sufficiently far apart"

- Writes to a single location are serialized: seen in one order
  - 'Latest' write will be seen
  - Otherwise could see writes in illogical order (could see older value after a newer value)

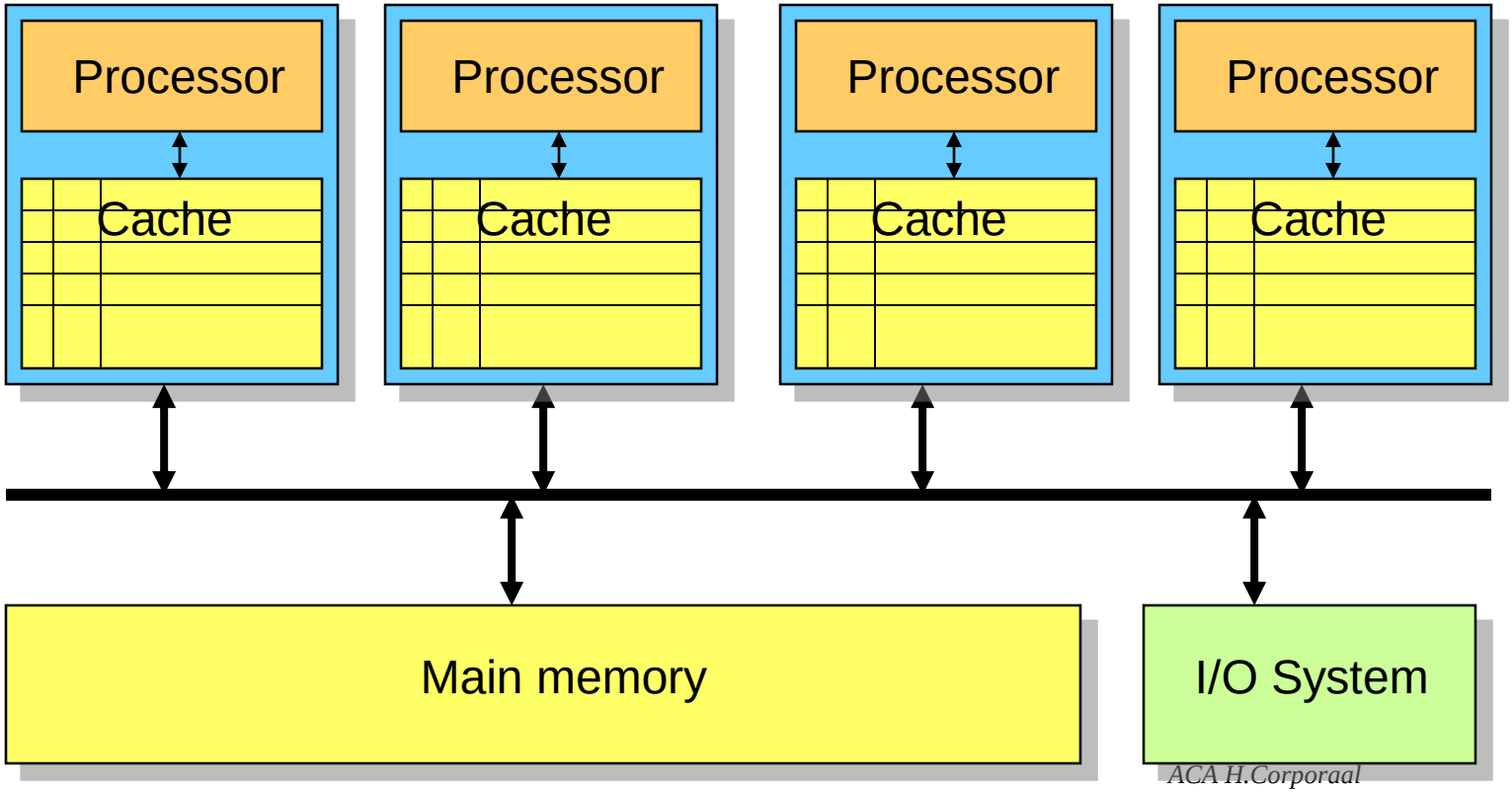
Potential HW Coherency Solutions

- **Snooping** Solution (Snoopy Bus):
  - Send all requests for data to all processors (or local caches)
  - Processors snoop to see if they have a copy and respond accordingly
  - Requires broadcast, since caching information is at processors
  - Works well with bus (natural broadcast medium)
  - Dominates for small scale machines (most of the market)

- **Directory-Based** Schemes
  - Keep track of what is being shared in one centralized place
  - Distributed memory => distributed directory for scalability (avoids bottlenecks, hot spots)
  - Scales better than Snooping
  - Actually existed BEFORE Snooping-based schemes

Example Snooping protocol

- 3 states for each cache line:
  - invalid, shared (read only), modified (also called exclusive, you may write it)
- FSM per cache, gets requests from processor and bus



Snooping Protocol: Write Invalidate

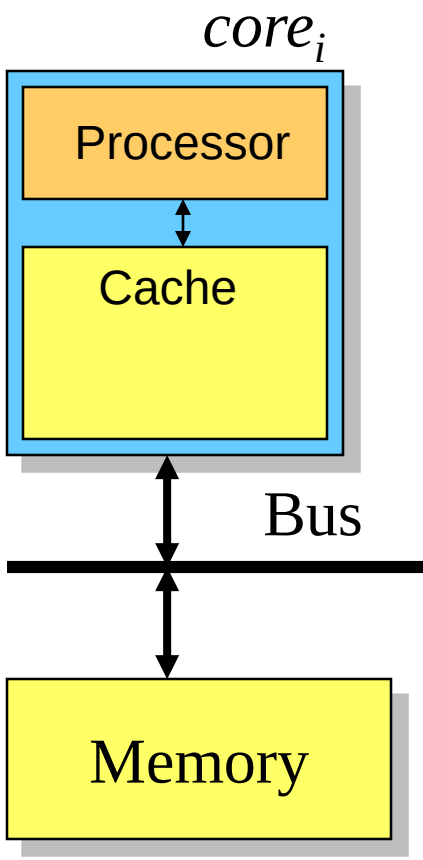
- Get exclusive access to a cache block (invalidate all other copies) before writing it
- When processor reads an invalid cache block it is forced to fetch a new copy
- If two processors attempt to write simultaneously, one of them is first (having a bus helps). The other one must obtain a new copy, thereby enforcing serialization

Processor activity	**Bus activity**	**Cache CPU A**	**Cache CPU B**	**Memory addr. X**
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes 1 to X	Invalidation for X	1	*invalidated*	0
CPU B reads X	Cache miss for X	1	1	1

Example: address X in memory initially contains value '0'

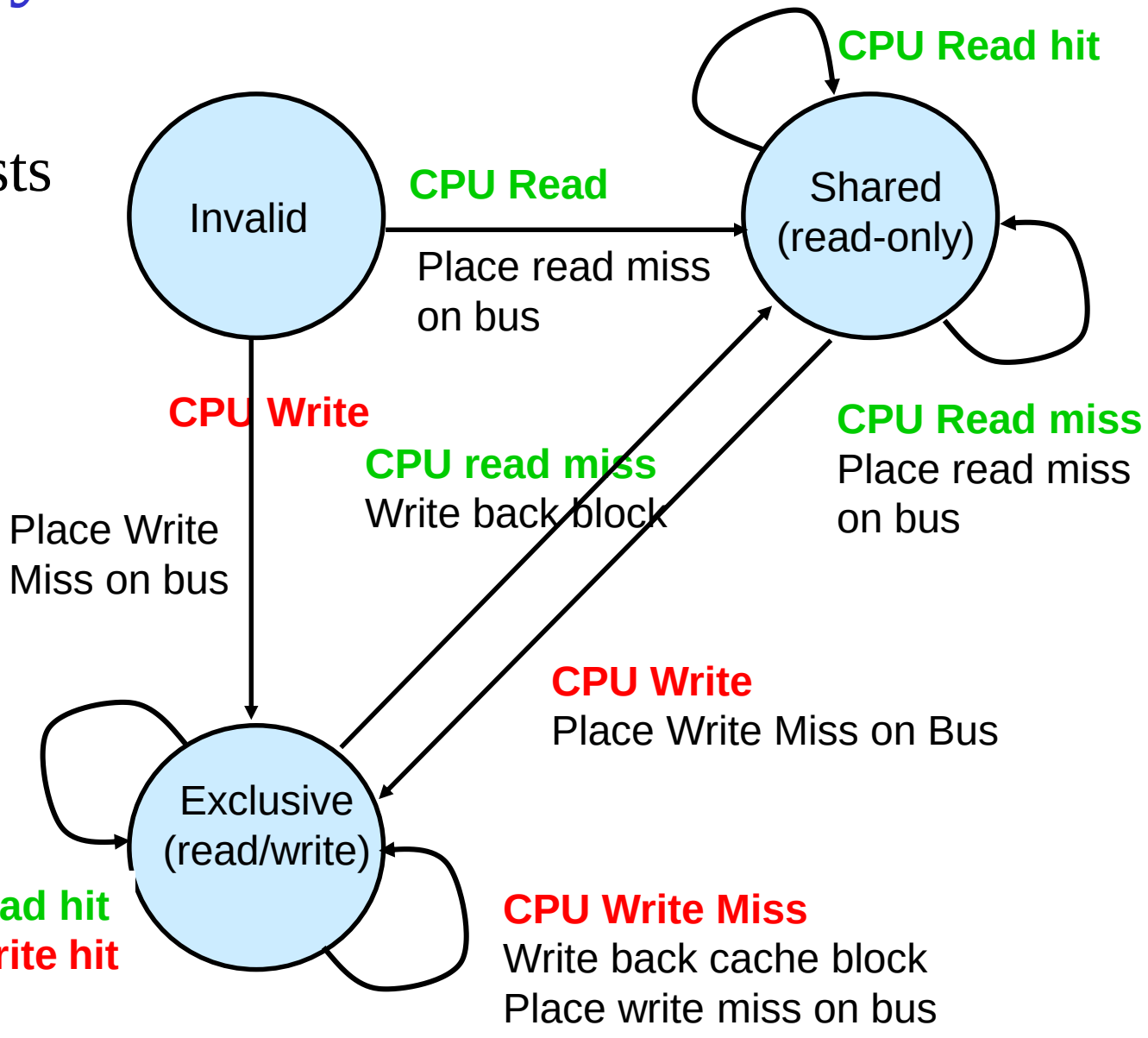
Basics of Write Invalidate

- Use the bus to perform invalidates
  - To perform an invalidate, acquire bus access and broadcast the address to be invalidated
    - all processors snoop the bus, listening to addresses
    - if the address is in my cache, invalidate my copy
- Serialization of bus access enforces write serialization
- Where is the most recent value?
  - Easy for write-through caches: in the memory
  - For write-back caches, again use snooping
- Can use cache tags to implement snooping
  - Might interfere with cache accesses coming from CPU
  - Duplicate tags, or employ multilevel cache with inclusion



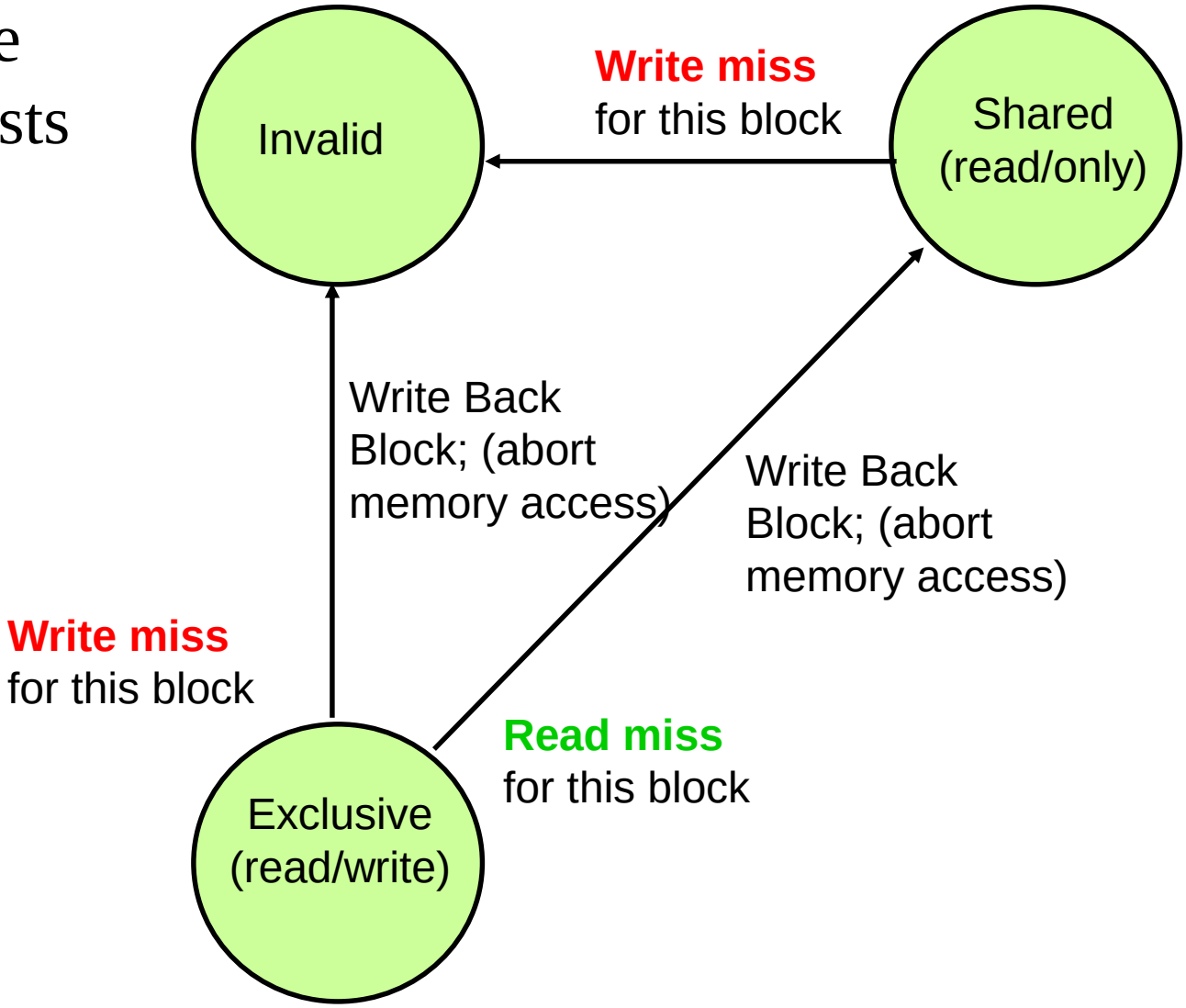
Snoopy-Cache State Machine-I

- State machine for *CPU* requests for each cache block



Snoopy-Cache State Machine-II

- State machine for *bus* requests for each cache block



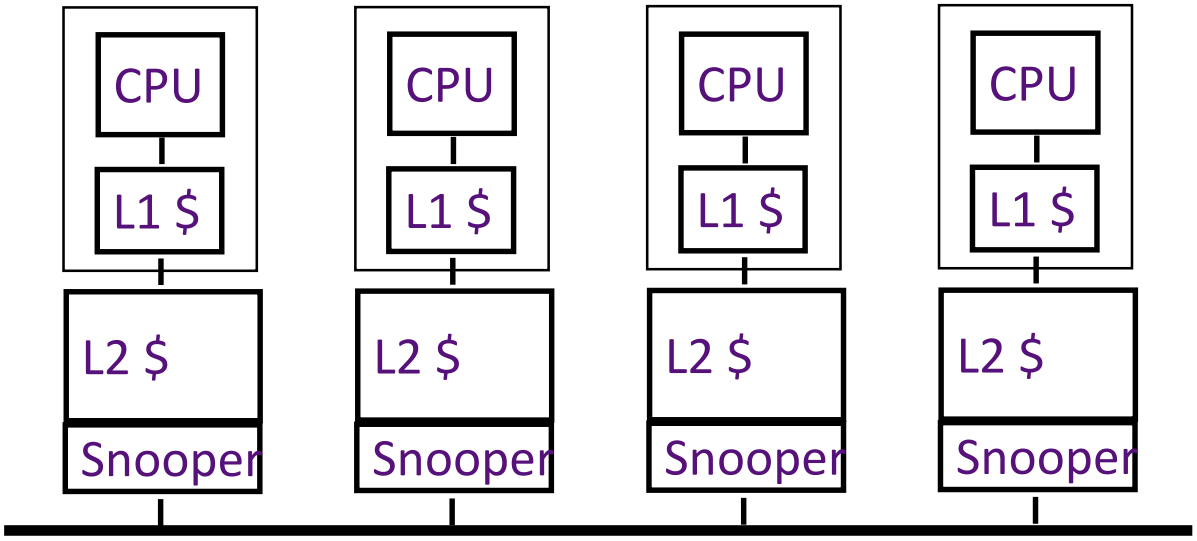
Responds to Events Caused by Processor

Event	**State of block *in cache***	**Action**
Read hit	shared or exclusive	read data from cache
Read miss	invalid	place read miss on bus
Read miss	shared	*wrong block* (conflict miss): place read miss on bus
Read miss	exclusive	conflict miss: write back block then place read miss on bus
Write hit	exclusive	write data in cache
Write hit	shared	place write miss on bus (invalidates all other copies)
Write miss	invalid	place write miss on bus
Write miss	shared	conflict miss: place write miss on bus
Write miss	exclusive	conflict miss: write back block, then place write miss on bus

Responds to Events on Bus

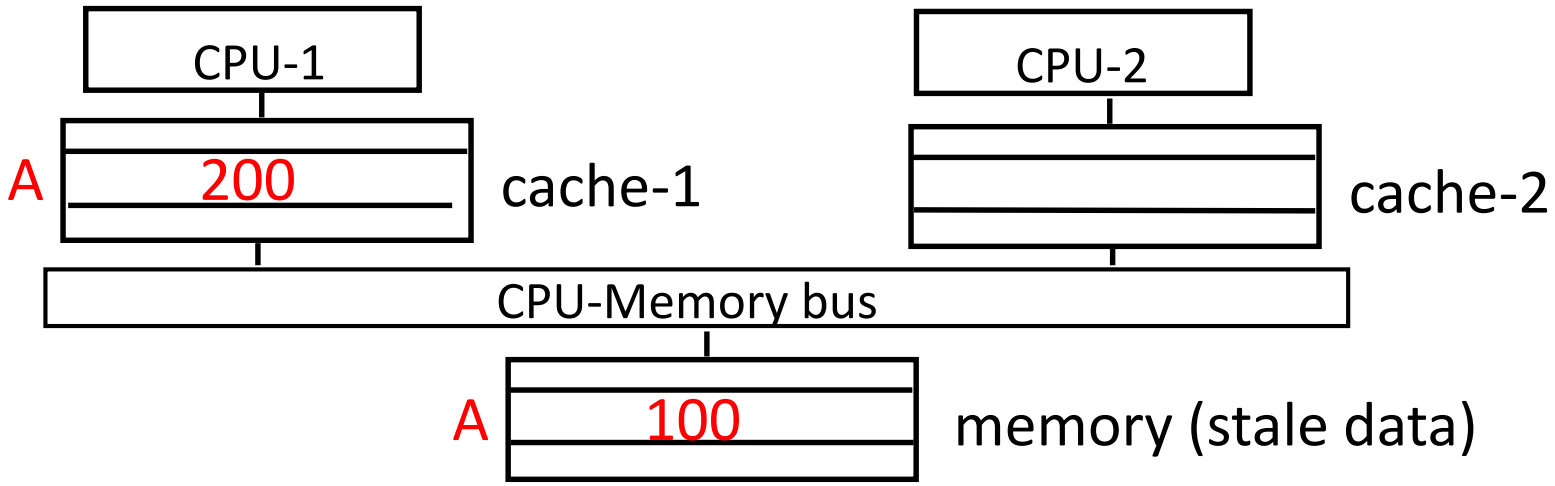
Event	**State of *addressed* cache block**	**Action**
Read miss	shared	No action: memory services read miss
Read miss	exclusive	Attempt to share data: place block on bus and change state to shared
Write miss	shared	Attempt to write: invalidate block
Write miss	exclusive	Another processor attempts to write "my" block: write back the block and invalidate it

Optimized Snoop with Level-2 Caches



- Processors often have two-level caches
  - small L1, large L2 (usually both on chip now)
- Inclusion property: entries in L1 must be in L2
  - invalidation in L2 ⇒ invalidation in L1
- Snooping on L2 does not affect CPU-L1 bandwidth

Intervention



When a read-miss for A occurs in cache-2,
a read request for A is placed on the bus

- Cache-1 needs to supply & change its state to shared
- The memory may respond to the request also!

Does memory know it has stale data?

Cache-1 needs to intervene through memory controller to supply correct data to cache-2

False Sharing

A cache line contains more than one word

Cache-coherence is done at the line-level and not word-level

Suppose M_1 writes $word_i$ and M_2 writes $word_k$ and both words have the same line address.

What can happen?

Performance of Symmetric Multiprocessors (SMPs)

Cache performance is combination of:

- Uniprocessor cache miss traffic
- Traffic caused by communication
  - Results in invalidations and subsequent cache misses
- Coherence misses
  - Sometimes called a Communication miss
  - 4th C of cache misses along with Compulsory, Capacity, & Conflict.

Coherency Misses

- True sharing misses arise from the communication of data through the cache coherence mechanism
  - Invalidates due to 1st write to shared line
  - Reads by another CPU of modified line in different cache
  - Miss would still occur if line size were 1 word
- False sharing misses when a line is invalidated because some word in the line, other than the one being read, is written into
  - Invalidation does not cause a new value to be communicated, but only causes an extra cache miss
  - Line is shared, but no word in line is actually shared
    $\Rightarrow$ miss would not occur if line size were 1 word

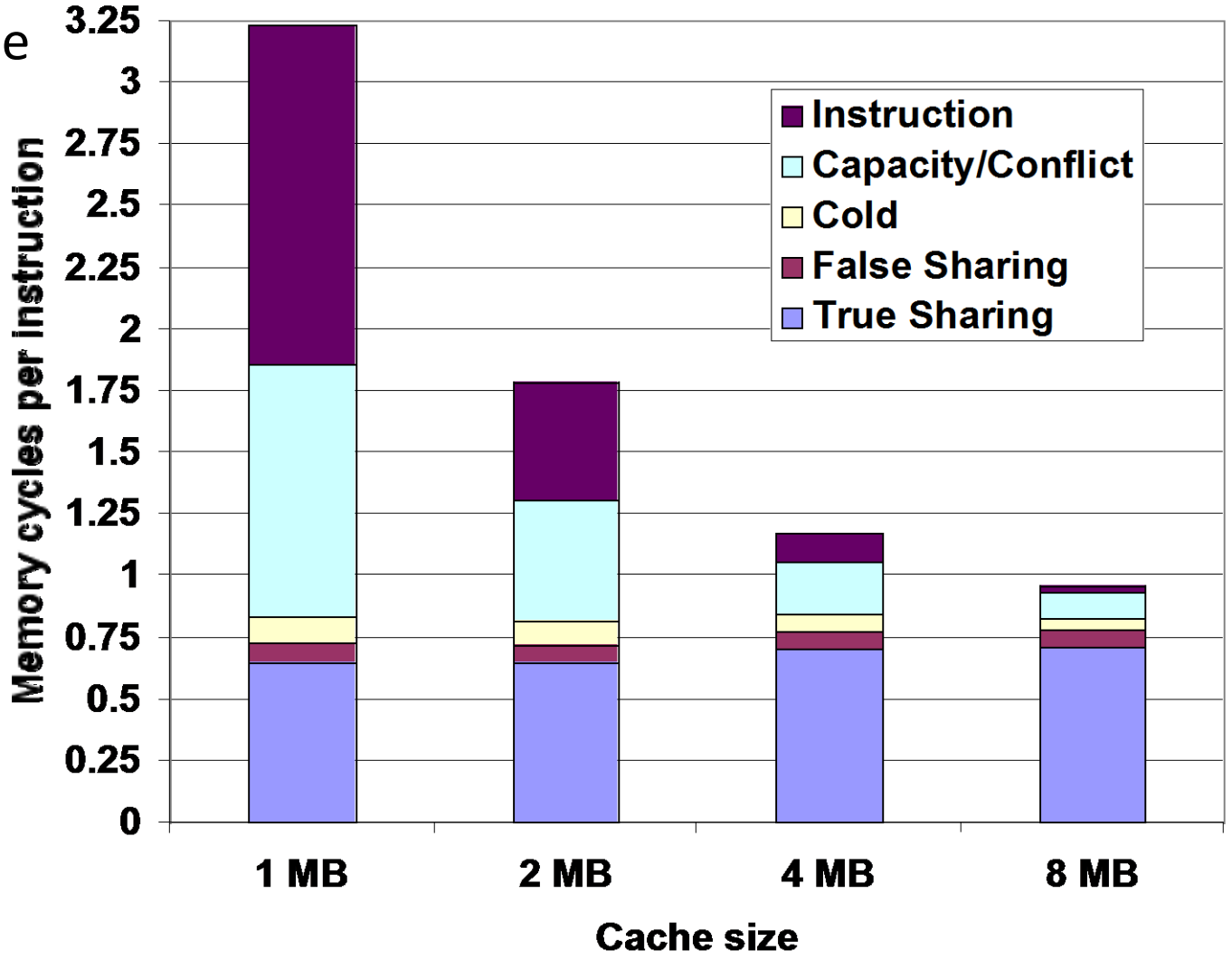
Example: True v. False Sharing v. Hit?

- Assume x1 and x2 in same cache line.
  P1 and P2 both read x1 and x2 before.

Time	P1	P2	True, False, Hit? Why?
1	Write x1		True miss; invalidate x1 in P2
2		Read x2	False miss; x1 irrelevant to P2
3	Write x1		False miss; x1 irrelevant to P2
4		Write x2	True miss; x2 not writeable
5	Read x2		True miss; invalidate x2 in P1

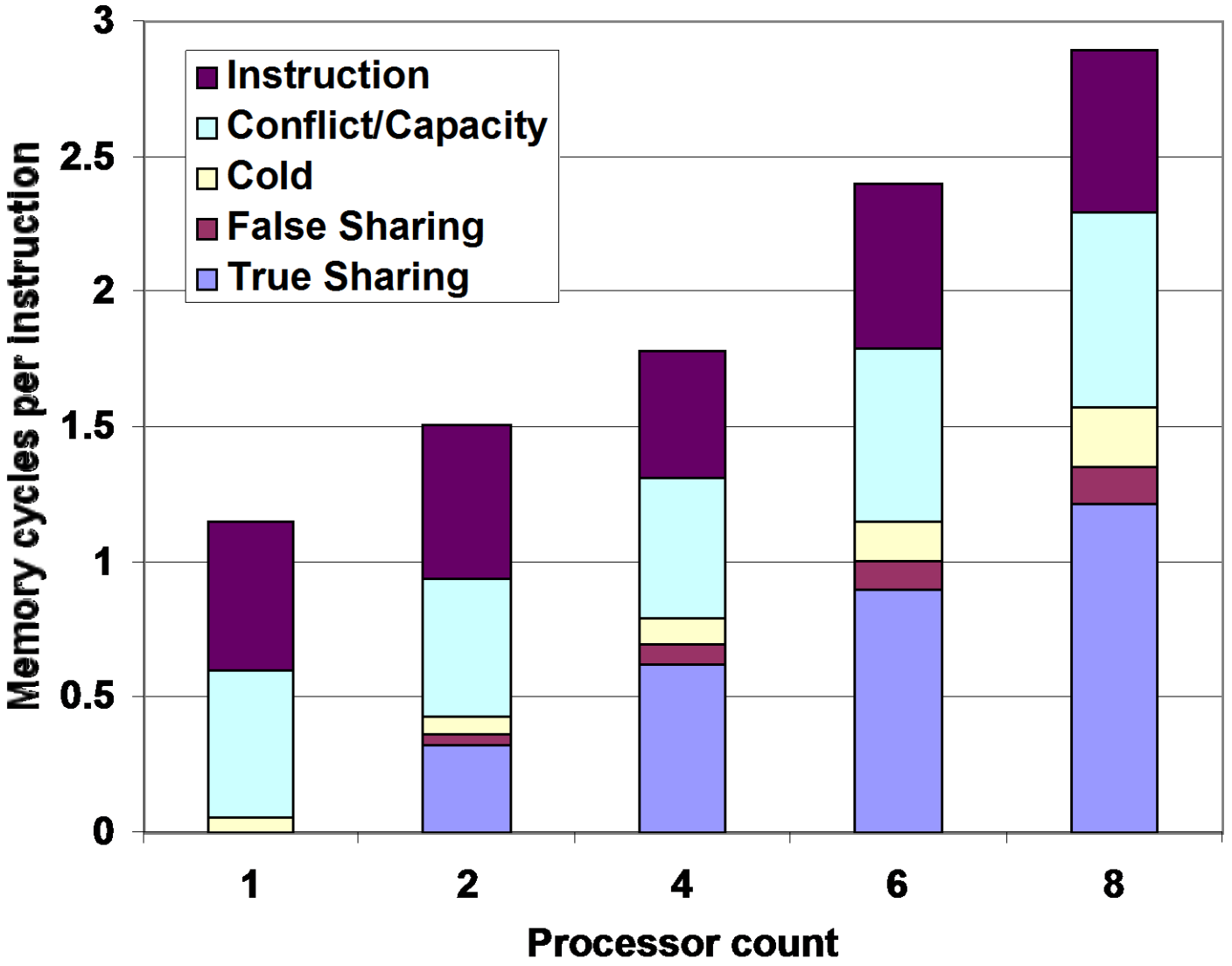
MP Performance 4-Processor Commercial Workload: OLTP, Decision Support (Database), Search Engine

- Uniprocessor cache misses improve with cache size increase (Instruction, Capacity/Conflict, Compulsory)
- True sharing and false sharing unchanged going from 1 MB to 8 MB (L3 cache)



MP Performance 2MB Cache Commercial Workload: OLTP, Decision Support (Database), Search Engine

- True sharing, false sharing increase going from 1 to 8 CPUs



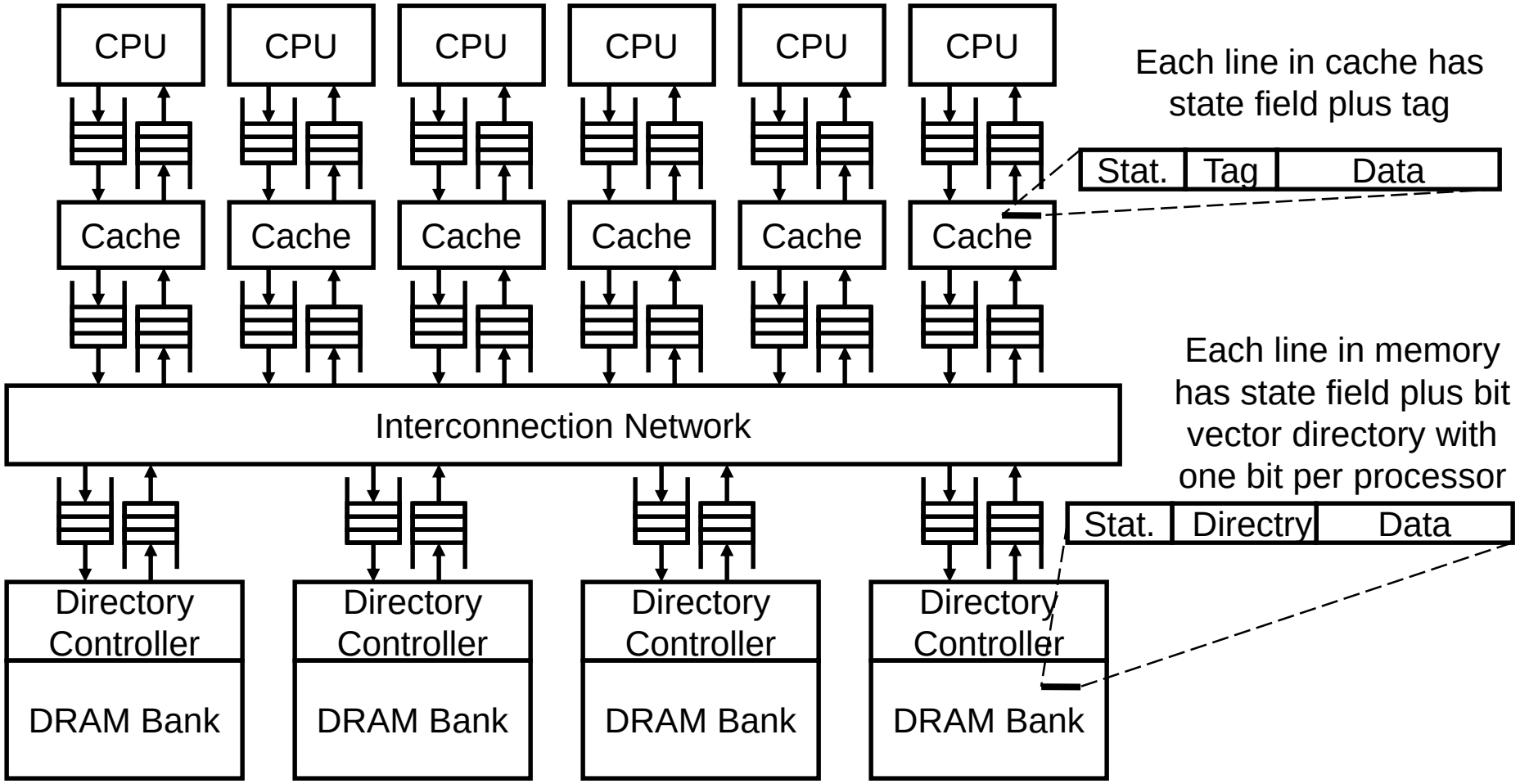
Scaling Snoopy/Broadcast Coherence

- When any processor gets a miss, must probe every other cache
- Scaling up to more processors limited by:
  - Communication bandwidth over bus
  - Snoop bandwidth into tags
- Can improve bandwidth by using multiple interleaved buses with interleaved tag banks
  - E.g, two bits of address pick which of four buses and four tag banks to use – (e.g., bits 7:6 of address pick bus/tag bank, bits 5:0 pick byte in 64-byte line)
- Buses don't scale to large number of connections, so can use point-to-point network for larger number of nodes, but then limited by tag bandwidth when broadcasting snoop requests.
- **Insight**: Most snoops fail to find a match!

Scalable Approach: Directories

- Every memory line has associated directory information
  - keeps track of copies of cached lines and their states
  - on a miss, find directory entry, look it up, and communicate only with the nodes that have copies if necessary
  - in scalable networks, communication with directory and copies is through network transactions
- Many alternatives for organizing directory information

Directory Cache Protocol



- Assumptions: Reliable network, FIFO message delivery between any given source-destination pair

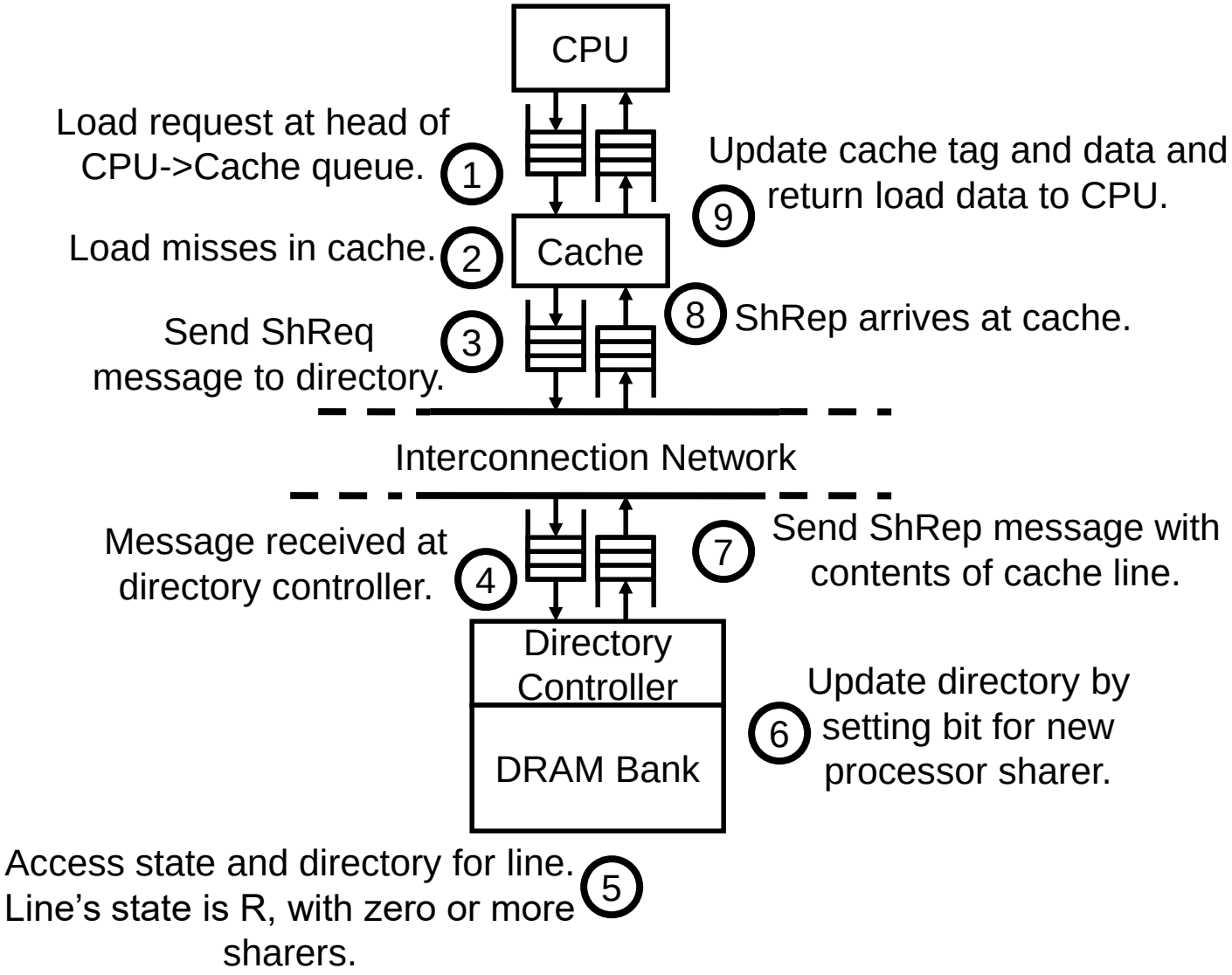
Cache States

- For each cache line, there are 4 possible states:
  - C-invalid (= Nothing): The accessed data is not resident in the cache.
  - C-shared (= Sh): The accessed data is resident in the cache, and possibly also cached at other sites. The data in memory is valid.
  - C-modified (= Ex): The accessed data is exclusively resident in this cache, and has been modified. Memory does not have the most up-to-date data.
  - C-transient (= Pending): The accessed data is in a transient state (for example, the site has just issued a protocol request, but has not received the corresponding protocol reply).

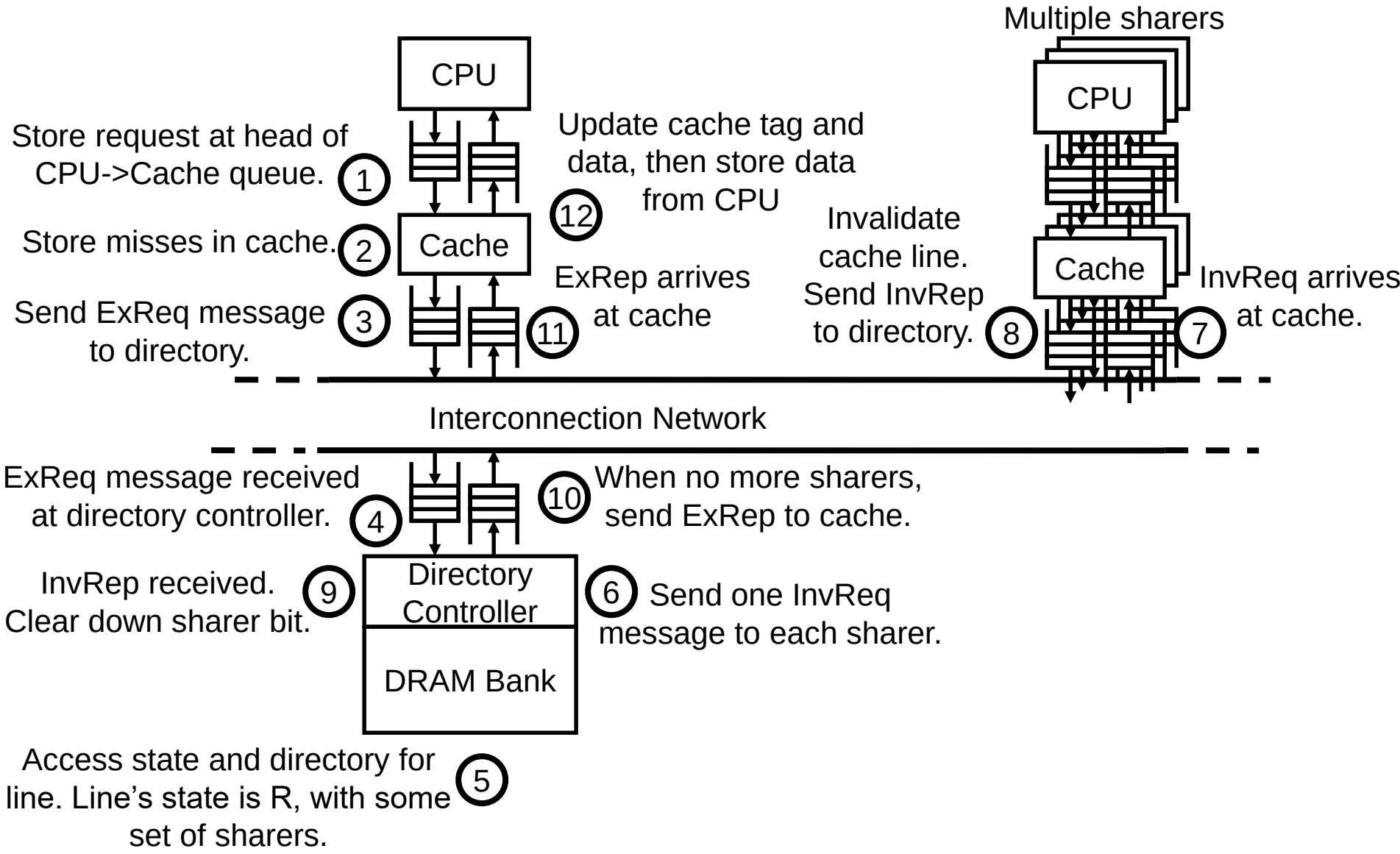
Home directory states

- For each memory line, there are 4 possible states:
  - R(dir): The memory line is shared by the sites specified in dir (dir is a set of sites). The data in memory is valid in this state. If dir is empty (i.e., dir = ε), the memory line is not cached by any site.
  - W(id): The memory line is exclusively cached at site id, and has been modified at that site. Memory does not have the most up-to-date data.
  - TR(dir): The memory line is in a transient state waiting for the acknowledgements to the invalidation requests that the home site has issued.
  - TW(id): The memory line is in a transient state waiting for a line exclusively cached at site id (i.e., in C-modified state) to make the memory line at the home site up-to-date.

Read miss, to uncached or shared line



Write miss, to read shared line



Concurrency Management

- Protocol would be easy to design if only one transaction in flight across entire system
- But, want greater throughput and don't want to have to coordinate across entire system
- Great complexity in managing multiple outstanding concurrent transactions to cache lines
  - Can have multiple requests in flight to same cache line!

Three fundamental issues for shared memory multiprocessors

- **Coherence**,
  about: *Do I see the most recent data?*

- **Synchronization**
  How to synchronize processes?
  - how to protect access to shared data?

- **Consistency**,
  about: ***When** do I see a written value?*
  - e.g. do different processors see writes at the same time (w.r.t. other memory accesses)?

What's the Synchronization problem?

- Assume: Computer system of bank has credit process (P_c) and debit process (P_d)

```
/* Process P_c */
shared int balance
private int amount

balance += amount

lw      $t0,balance
lw      $t1,amount
add     $t0,$t0,t1
sw      $t0,balance
```

```
/* Process P_d */
shared int balance
private int amount

balance -= amount

lw      $t2,balance
lw      $t3,amount
sub     $t2,$t2,$t3
sw      $t2,balance
```

Critical Section Problem

- n processes all competing to use some shared data
- Each process has code segment, called *critical section*, in which shared data is accessed.
- **Problem** – ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section
- Structure of process

```
while (TRUE){
   entry_section ();
   critical_section ();
   exit_section ();
   remainder_section ();
}
```

HW support for synchronization

Atomic Instructions to Fetch and Update Memory :

- Atomic **exchange**: interchange a value in a register for a value in memory
  - 0 => synchronization variable is free
  - 1 => synchronization variable is locked and unavailable
- **Test-and-set**: tests a value and sets it if the value passes the test
  - similar: **Compare-and-swap**
- **Fetch-and-increment**: it returns the value of a memory location and atomically increments it
  - 0 => synchronization variable is free

Three fundamental issues for shared memory multiprocessors

- **Coherence**,
  about: *Do I see the most recent data?*

- **Synchronization**
  How to synchronize processes?
  - how to protect access to shared data?

- **Consistency**,
  about: ***When** do I see a written value?*
  - e.g. do different processors see writes at the same time (w.r.t. other memory accesses)?

Memory Consistency: The Problem

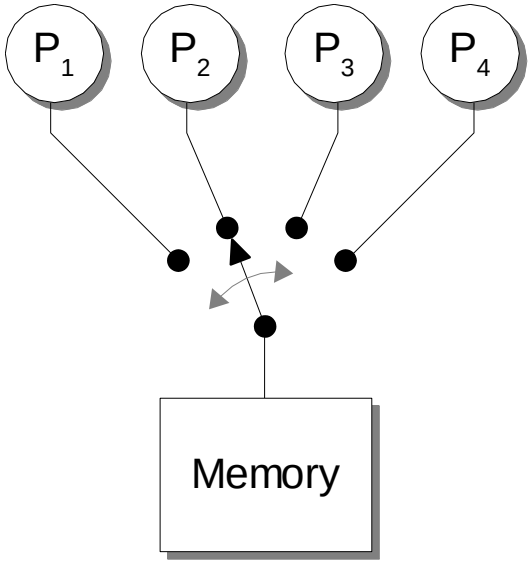
Process P1	Process P2
`A = 0;`	`B = 0;`
`...`	`...`
`A = 1;`	`B = 1;`
`L1: if (B==0) ...`	`L2: if (A==0) ...`

- Observation: If writes take effect immediately (are immediately seen by all processors), it is impossible that both if-statements evaluate to true

- But what if write invalidate is delayed …….
  - Should this be allowed, and if so, under what conditions?

Sequential Consistency

Lamport (1979): A multiprocessor is sequentially consistent if the result of any execution is the same as if the (memory) operations of all processors were executed in some sequential order, and the operations of each individual processor occur in this sequence in the order specified by its program



This means that all processors 'see' all loads and stores happening in the same order !!

How to implement Sequential Consistency

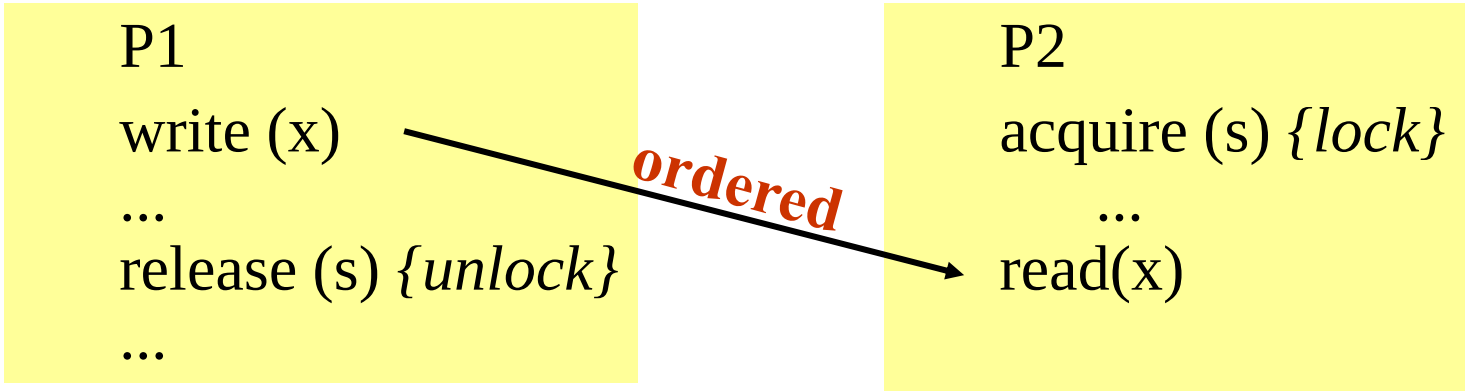
- Delay the completion of any memory access until all invalidations caused by that access are completed

- Delay next memory access until previous one is completed
  - delay the read of A and B (A==0 or B==0 in the example) until the write has finished (A=1 or B=1)

- Note: Under sequential consistency, we cannot place the (local) write in a write buffer and continue

Sequential consistency overkill?

- Schemes for faster execution then sequential consistency
- Observation: Most programs are synchronized
  - A program is synchronized if all accesses to shared data are ordered by synchronization operations
- Example:



Cost of Sequential Consistency (SC)

- Enforcing SC can be quite expensive
  - Assume write miss = 40 cycles to get ownership,
  - 10 cycles = to issue an invalidate and
  - 50 cycles = to complete and get acknowledgement
  - Assume 4 processors share a cache block, how long does a write miss take for the writing processor if the processor is sequentially consistent?
- Waiting for invalidates : each write =sum of ownership time + time to complete invalidates
  - 10+10+10+10= 40 cycles to issue invalidate
  - 40 cycles to get ownership + 50 cycles to complete
  - =130 cycles ➔very long !

- Solutions:
  - Exploit latency-hiding techniques
  - Employ relaxed consistency

Relaxed Memory Consistency Models

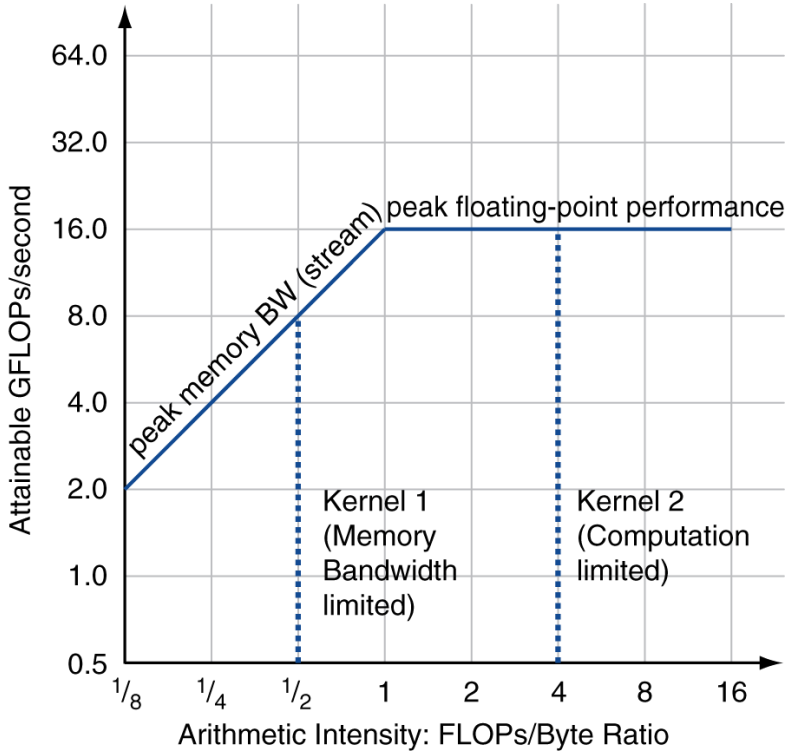
- Key: ***(partially) allow reads and writes to complete out-of-order***
- Orderings that can be relaxed:
  - relax W $\Rightarrow$ R ordering
    - allows reads to bypass earlier writes (to different memory locations)
    - called ***processor consistency*** or ***total store ordering***
  - relax W $\Rightarrow$ W
    - allow writes to bypass earlier writes
    - called ***partial store order***
  - relax R $\Rightarrow$ W *and* R $\Rightarrow$ R
    - ***weak ordering***, ***release consistency***, Alpha, PowerPC
- Note, seq. consistency means:
  - W $\Rightarrow$ R, W $\Rightarrow$ W, R $\Rightarrow$ W and R $\Rightarrow$ R

PERFORMANCE ANALYSIS AND SCALABILITY CONSIDERATIONS

Modeling Performance

- Assume performance metric of interest is achievable GFLOPs/sec
  - Measured using computational kernels from Berkeley Design Patterns
- Arithmetic intensity of a kernel
  - FLOPs per byte of memory accessed
- For a given computer, determine
  - Peak GFLOPS (from data sheet)
  - Peak memory bytes/sec (using Stream benchmark)

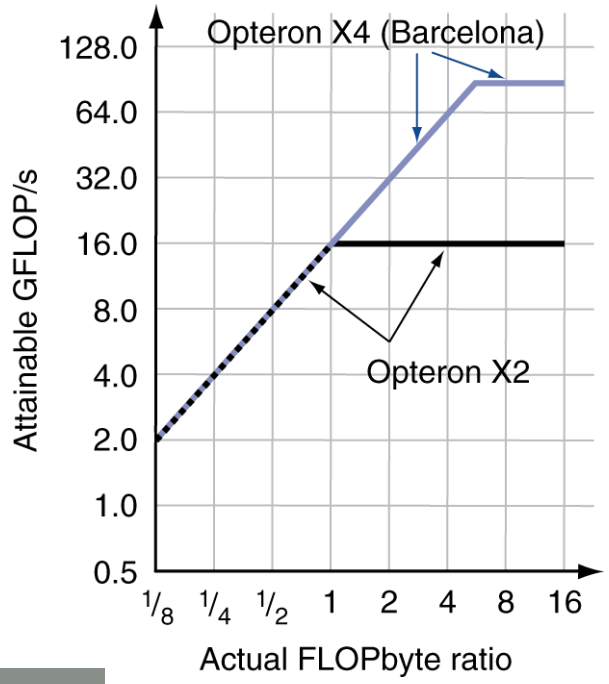
Roofline Diagram



Attainable GPLOPs/sec
= Max (Peak Memory BW × Arithmetic Intensity, Peak FP Performance)

Comparing Systems

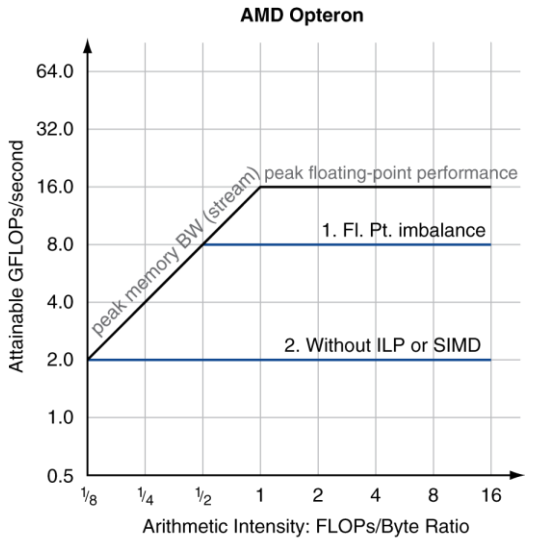
- Example: Opteron X2 vs. Opteron X4
  - 2-core vs. 4-core, 2× FP performance/core, 2.2GHz vs. 2.3GHz
  - Same memory system

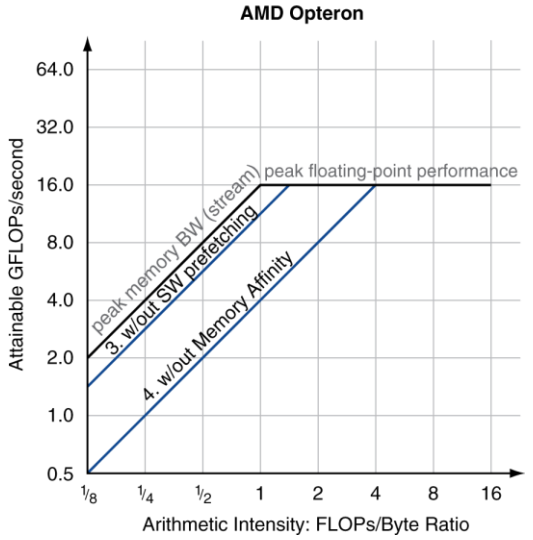


- To get higher performance on X4 than X2
  - Need high arithmetic intensity
  - Or working set must fit in X4's 2MB L-3 cache

Optimizing Performance

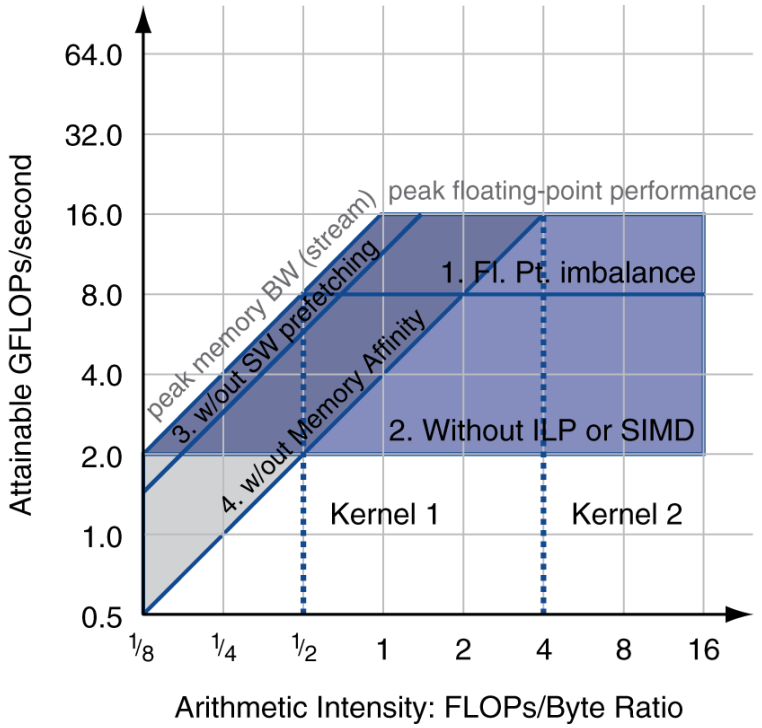
- Optimize FP performance
  - Balance adds & multiplies
  - Improve superscalar ILP and use of SIMD instructions
- Optimize memory usage
  - Software prefetch
    - Avoid load stalls
  - Memory affinity
    - Avoid non-local data accesses





Optimizing Performance

- Choice of optimization depends on arithmetic intensity of code



- Arithmetic intensity is not always fixed
  - May scale with problem size
  - Caching reduces memory accesses
    - Increases arithmetic intensity

Fallacies

- Amdahl's Law doesn't apply to parallel computers
  - Since we can achieve linear speedup
  - But only on applications with weak scaling
- Peak performance tracks observed performance
  - Marketers like this approach!
  - But compare Xeon with others in example
  - Need to be aware of bottlenecks

Pitfalls

- Not developing the software to take account of a multiprocessor architecture
  - Example: using a single lock for a shared composite resource
    - Serializes accesses, even if they could be done in parallel
    - Use finer-granularity locking

Concluding Remarks

- Goal: higher performance by using multiple processors
- Difficulties
  - Developing parallel software
  - Devising appropriate architectures
- SaaS importance is growing and clusters are a good match
- Performance per dollar and performance per Joule drive both mobile and WSC

Concluding Remarks (con't)

- SIMD and vector operations match multimedia applications and are easy to program