Lecture 10
Thread Level Parallelism (4)

EEC 171 Parallel Architectures

John Owens
UC Davis

Credits

- © John Owens / UC Davis 2007–17.
- Thanks to many sources for slide material: Computer Organization and Design (Patterson & Hennessy) © 2005, Computer Architecture (Hennessy & Patterson) © 2007, Inside the Machine (Jon Stokes) © 2007, © Dan Connors / University of Colorado 2007, © Kathy Yelick / UCB 2007, © Wen-Mei Hwu/David Kirk, University of Illinois 2007, © David Patterson / UCB 2003–7, © John Lazzaro / UCB 2006, © Mary Jane Irwin / Penn State 2005, © John Kubiatowicz / UCB 2002, © Krste Asinovic/Arvind / MIT 2002, © Morgan Kaufmann Publishers 1998.

LBNL talk

CS Seminar: Hybrid MPI and OpenMP Parallel Programming on Clusters of Multi-Core Nodes

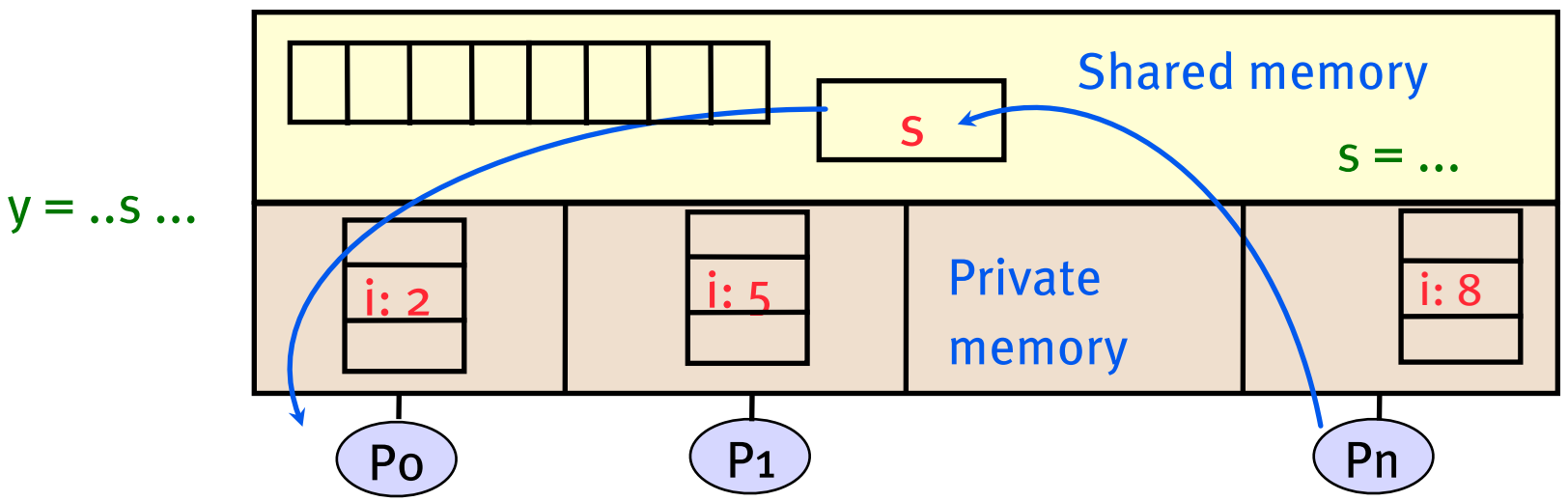
Berkeley Lab – Computing Sciences Seminar: Friday, May 3, 2013
Speaker: Gabriele Jost, Supersmith, Monterey, CA

Software developers who wish to write efficient parallel software for current large scale parallel systems are faced with highly hierarchical system designs. The software developer is faced with several levels of caches, shared and distributed memory, cores, sockets and nodes. Most recently, heterogeneity is added through processors equipped with accelerators. MPI/OpenMP and pure MPI on clusters of multi-core SMP (Shared Memory Processor) nodes involve several mismatch problems between the parallel programming models and the hardware architectures. Measurements of communication characteristics between cores on the same socket, on the same SMP node, and between SMP nodes on several platforms show that machine topology has a significant impact on performance for all parallelization strategies and that topology awareness should be built into all applications in the future. The talk will describe potentials and challenges of the hybrid MPI/OpenMP programming model on hierarchically structured hardware. OpenMP 4.0 support for nodes with accelerators and the OpenACC programming model will be included.

Programming Model 1: Shared Memory

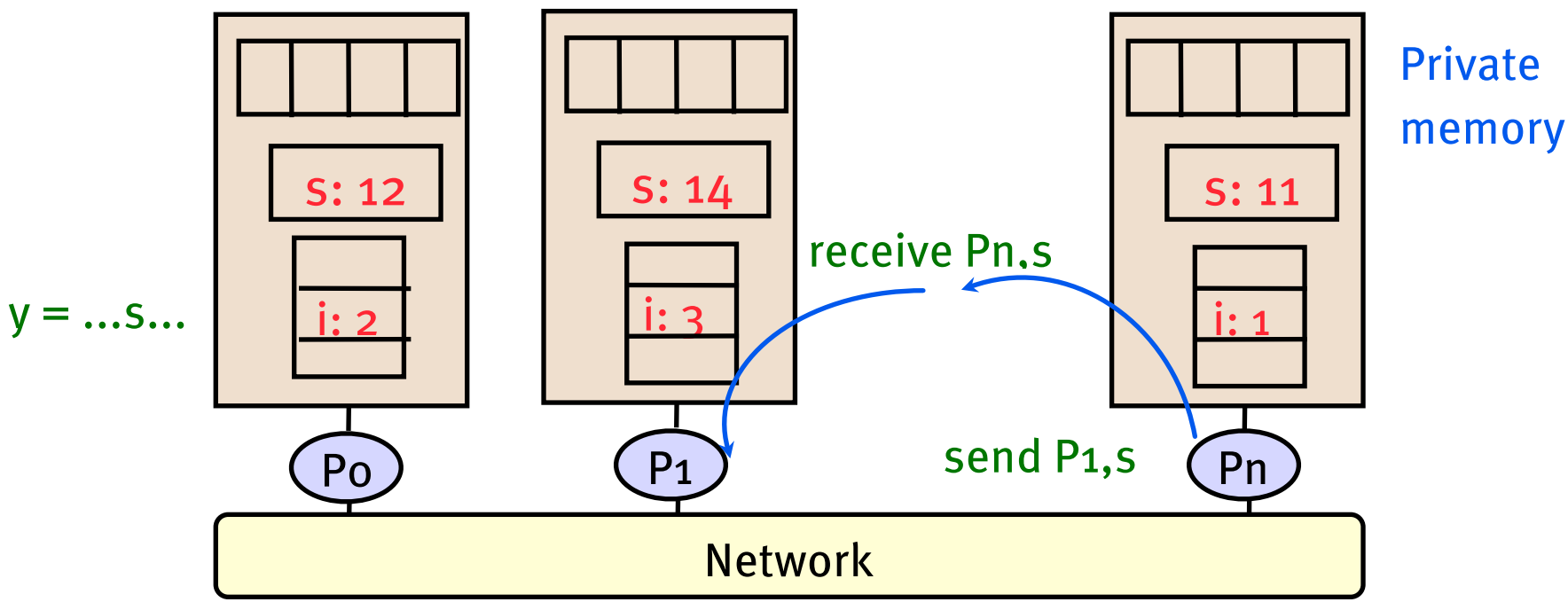
- Program is a collection of threads of control.
  - Can be created dynamically, mid-execution, in some languages
- Each thread has a set of private variables, e.g., local stack variables
- Also a set of shared variables, e.g., static variables, shared common blocks, or global heap.
  - Threads communicate implicitly by writing and reading shared variables.
  - Threads coordinate by synchronizing on shared variables

Shared Memory



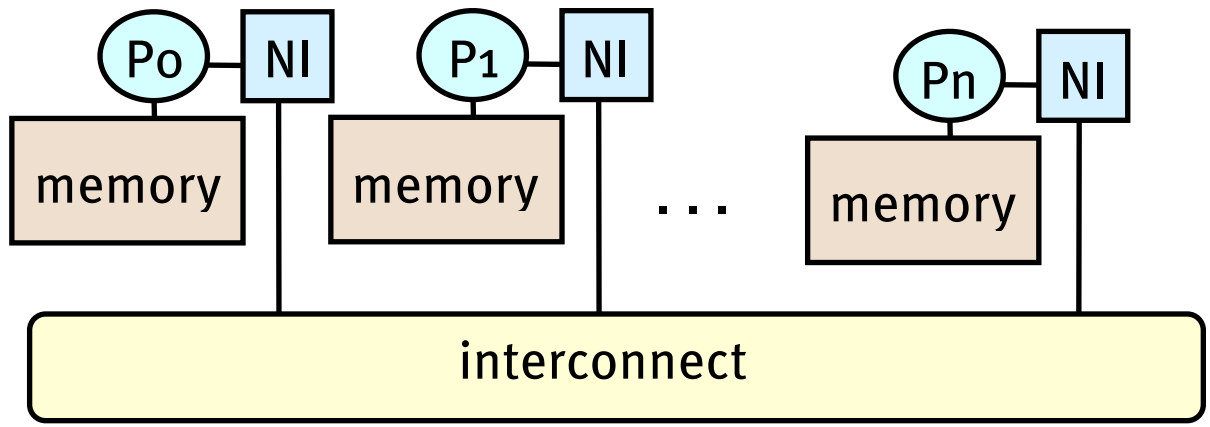
Programming Model 2: Message Passing

- Program consists of a collection of named processes.
  - Usually fixed at program startup time
  - Thread of control plus local address space—NO shared data.
  - Logically shared data is partitioned over local processes.



Machine Model 2a: Distributed Memory

- Cray T3E, IBM SP2
- PC Clusters (Berkeley NOW, Beowulf)
- IBM SP-3, Millennium, CITRIS are distributed memory machines, but the nodes are SMPs.
- Each processor has its own memory and cache but cannot directly access another processor's memory.
- Each "node" has a Network Interface (NI) for all communication and synchronization.



Challenges of Parallel Processing

- Application parallelism ⇒ primarily via new algorithms that have better parallel performance
- Long remote latency impact ⇒ both by architect and by the programmer
- For example, reduce frequency of remote accesses either by
  - Caching shared data (HW)
  - Restructuring the data layout to make more accesses local (SW)
- Today's lecture on HW to help latency via caches. If every memory access was a remote access, our programs would have poor perf.

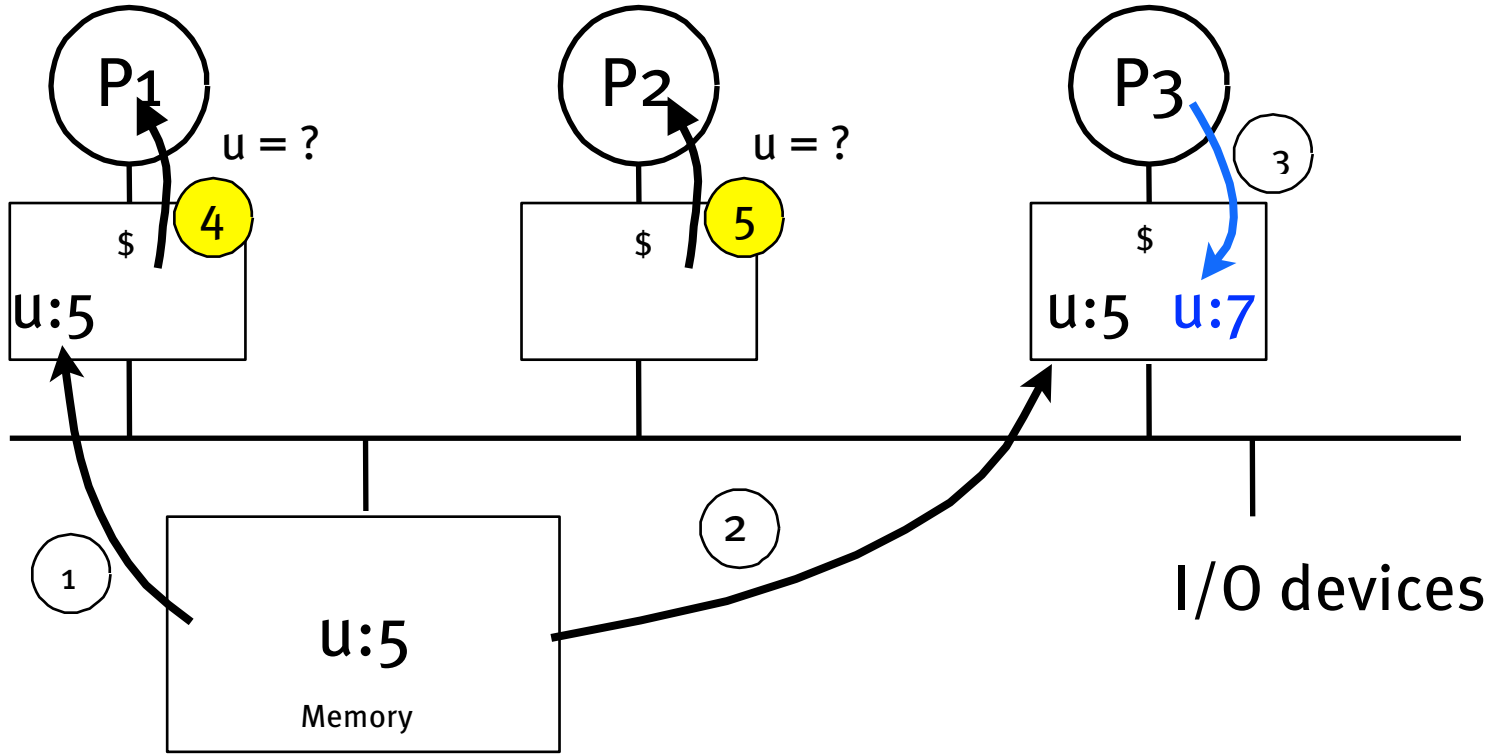
Fundamental Problem

- Many processors working on a task
- Those processors share data, need to communicate, etc.
- For efficiency, we use caches
- This results in multiple copies of the data
- Are we working with the right copy?

Symmetric Shared-Memory Architectures

- From multiple boards on a shared bus to multiple processors inside a single chip
- Caches:
  - Private data are used by a single processor
  - Shared data are used by multiple processors
- We would like to cache shared data:
  - reduces latency to shared data, memory bandwidth for shared data, and interconnect bandwidth
  - introduces a cache coherence problem

Example Cache Coherence Problem



- Processors see different values for u after event 3
- With write back caches, value written back to memory depends on happenstance of which cache flushes or writes back value when
  - Processes accessing main memory may see very stale value
- Unacceptable for programming, and it's frequent!

Intuitive Memory Model

- Reading an address should return the last value written to that address
  - Easy in uniprocessors, except for I/O
- Too vague and simplistic; 2 issues
  - Coherence defines values returned by a read (*what* value is returned on a read). *Coherence makes caches invisible*. (Today's focus.)
  - Consistency determines when a written value will be returned by a read (*when* that value is available). *Consistency makes shared memory look like a single memory module*.

Consistency & Coherence: Complex!

A Primer on Memory Consistency and Cache Coherence

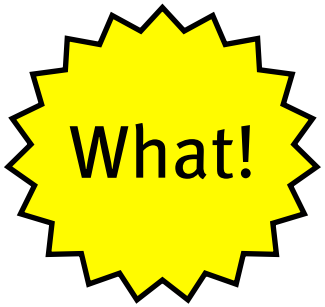
This lecture is intended to be a primer on coherence and consistency. We expect this material could be covered in a graduate class in about nine 75-minute classes, e.g., one lecture per Chapter 2 to Chapter 9 plus one lecture for advanced material). Goat simulation is also covered.

Daniel J. Sorin
Mark D. Hill
David A. Wood

Real World: Consistency

- Schedule says EEC 171 is in Bainer 1134
- Registrar moves class to Bainer 0134
  - Sends email to web administrator to change online schedule on web
  - Sends email to registered students to check online schedule
  - We would say this is a “weak consistency model”: there’s a point where different people might see different data (programmer must manage synchronization)
- You check website before administrator changes website
- Consistency rules say if this is correct or incorrect
- Consistency deals with loads/stores and ordering, not caches or coherence

Real World: Coherence



- You check online schedule, which says EEC 171 is in Bainer 1134
- You write it down in your notebook
- Registrar moves class to Bainer 0134, updates online schedule
- You check your notebook on the first day of class, see 1134
- Issue: Multiple copies of data; yours is stale

Defining Coherent Memory System

- Preserve Program Order: A read by processor P to location X that follows a write by P to X, with no writes of X by another processor occurring between the write and the read by P, always returns the value written by P
  - P writes D to X
  - Nobody else writes to X
  - P reads X -› always gives D

Defining Coherent Memory System

- Coherent view of memory: Read by a processor to location X that follows a write by another processor to X returns the written value if the read and write are sufficiently separated in time and no other writes to X occur between the two accesses
  - P1 writes D to X
  - Nobody else writes to X
  - … wait a while …
  - P2 reads X, should get D

Defining Coherent Memory System

- Write serialization: 2 writes to same location by any 2 processors are seen in the same order by all processors
  - If not, a processor could keep value 1 since saw as last write
  - For example, if the values 1 and then 2 are written to a location, processors can never read the value of the location as 2 and then later read it as 1

Write Consistency

- For now assume
  - A write does not complete (and allow the next write to occur) until all processors have seen the effect of that write
  - The processor does not change the order of any write with respect to any other memory access
- ⇒ if a processor writes location A followed by location B, any processor that sees the new value of B must also see the new value of A
- These restrictions allow the processor to reorder reads, but forces the processor to finish writes in program order

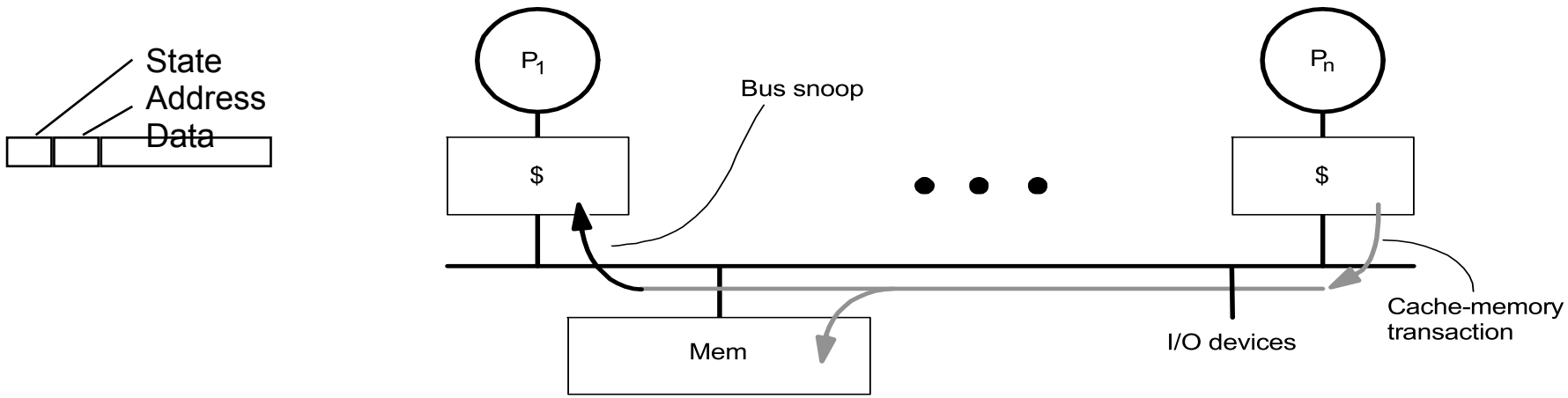
Basic Schemes for Enforcing Coherence

- Big picture: Caches have the potential to help our performance. How do we use them correctly?
- Program on multiple processors will normally have copies of the same data in several caches
  - Unlike I/O, where it's rare
- SMPs use a HW protocol to maintain coherent caches
  - Migration and Replication key to performance of shared data

2 Classes of Cache Coherence Protocols

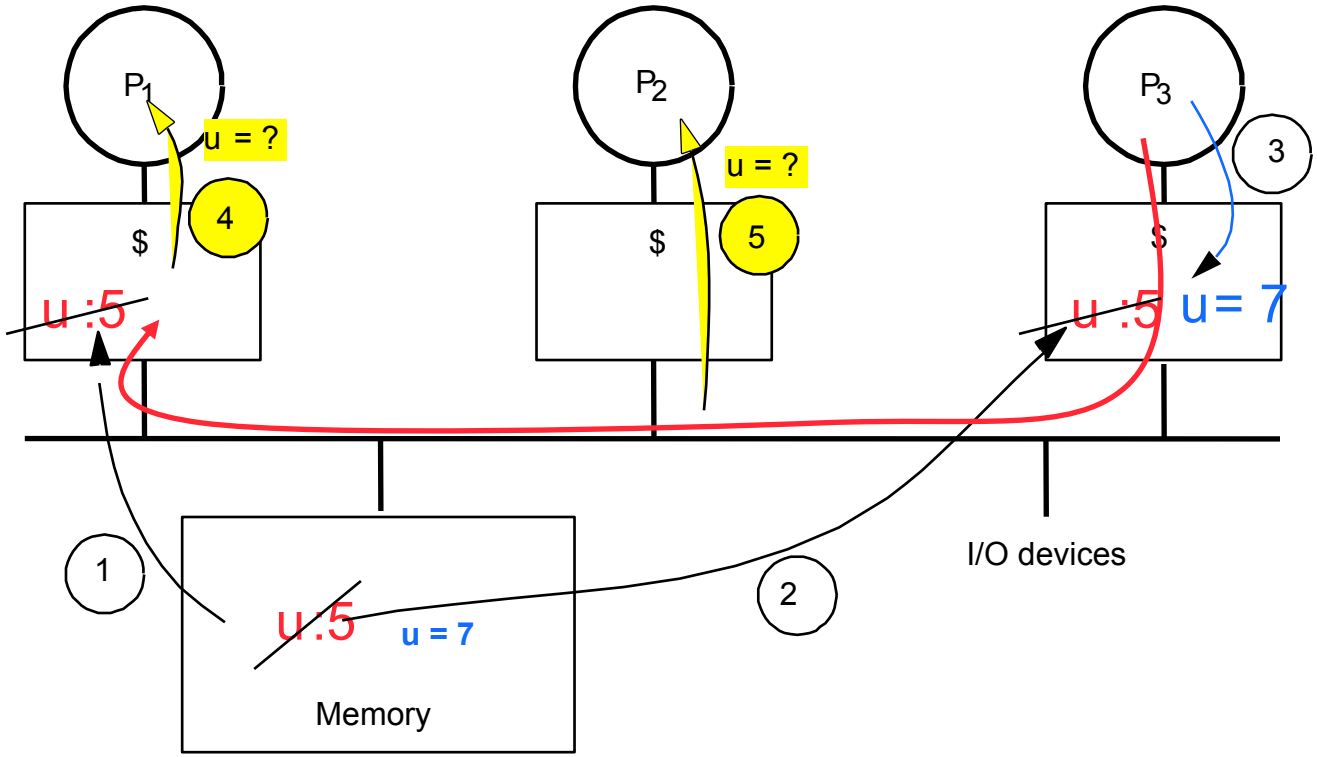
- Directory based — Sharing status of a block of physical memory is kept in just one location, the directory
- Snooping — Every cache with a copy of data also has a copy of sharing status of block, but no centralized state is kept
  - All caches are accessible via some broadcast medium (a bus or switch)
  - All cache controllers monitor or snoop on the medium to determine whether or not they have a copy of a block that is requested on a bus or switch access

Snoopy Cache-Coherence Protocols



- Cache Controller "snoops" all transactions on the shared medium (bus or switch)
  - "Does this transaction concern data that I have?"
  - If so, take action to ensure coherence
    - invalidate (my val), update (my val), or supply (my value) (when, when, and when?)
  - depends on state of the block and the protocol
- Either get exclusive access before write via write invalidate ("write invalidate") or update all copies on write ("write update") ("write update" != "update")

Example: Write-thru Invalidate



- Must invalidate before step 3
- Write update uses more broadcast medium BW
  ⇒ all recent MPUs use write invalidate

Architectural Building Blocks

- Cache block state transition diagram
  - FSM specifying how disposition of block changes
    - invalid, valid, exclusive
- Broadcast Medium Transactions (e.g., bus)
  - Fundamental system design abstraction
  - Logically single set of wires connect several devices
  - Protocol: arbitration, command/address, data
  - Every device observes every transaction

Architectural Building Blocks

- Problem: 2 processors try to write to same location at same time
- Broadcast medium enforces serialization of read or write accesses
  ⇒ Write serialization
  - 1st processor to get medium invalidates others copies
  - Implies cannot complete write until it obtains bus
  - All coherence schemes require serializing accesses to same cache block
- Also need to find up-to-date copy of cache block (on read for instance) [next slide]

Locate up-to-date copy of data

- Write-through: get up-to-date copy from memory
  - Write through simpler if enough memory BW
- Write-back harder
  - Most recent copy can be in a cache
- Can use same snooping mechanism
  - Snoop every address placed on the bus
  - If a processor has dirty copy of requested cache block, it provides it in response to a read request and aborts the memory access
  - Complexity from retrieving cache block from cache, which can take longer than retrieving it from memory
- Write-back needs lower memory bandwidth
  ⇒ Support larger numbers of faster processors

  ⇒ Most multiprocessors use write-back

Cache Resources for WB Snooping

- Normal cache tags can be used for snooping
- Valid bit per block makes invalidation easy
  - We already have this in our caches
- We want to read: if we miss, snooping makes it easy to get the right data
- We want to write ⇒ Need to know if any other copies of the block are cached (“shared”)
  - No other copies ⇒ No need to place write on bus for WB
  - Other copies ⇒ Need to place invalidate on bus

Cache Resources for WB Snooping

- To track whether a cache block is shared, add extra state bit associated with each cache block, like valid bit and dirty bit
  - Write to Shared block ⇒ Need to place invalidate on bus and mark cache block as private (if an option)
  - No further invalidations will be sent for that block
  - This processor called owner of cache block
  - Owner then changes state from shared to unshared (or exclusive)

Cache behavior in response to bus

- Every bus transaction must check the cache-address tags
  - could potentially interfere with processor cache accesses
- A way to reduce interference is to duplicate tags
  - One set for caches access, one set for bus accesses
- Another way to reduce interference is to use L2 tags
  - Since L2 less heavily used than L1
  - ⇒ Every entry in L1 cache must be present in the L2 cache, called the inclusion property
  - If Snoop gets a hit in L2 cache, then it must arbitrate for the L1 cache to update the state and possibly retrieve the data, which usually requires a stall of the processor

Example Protocol

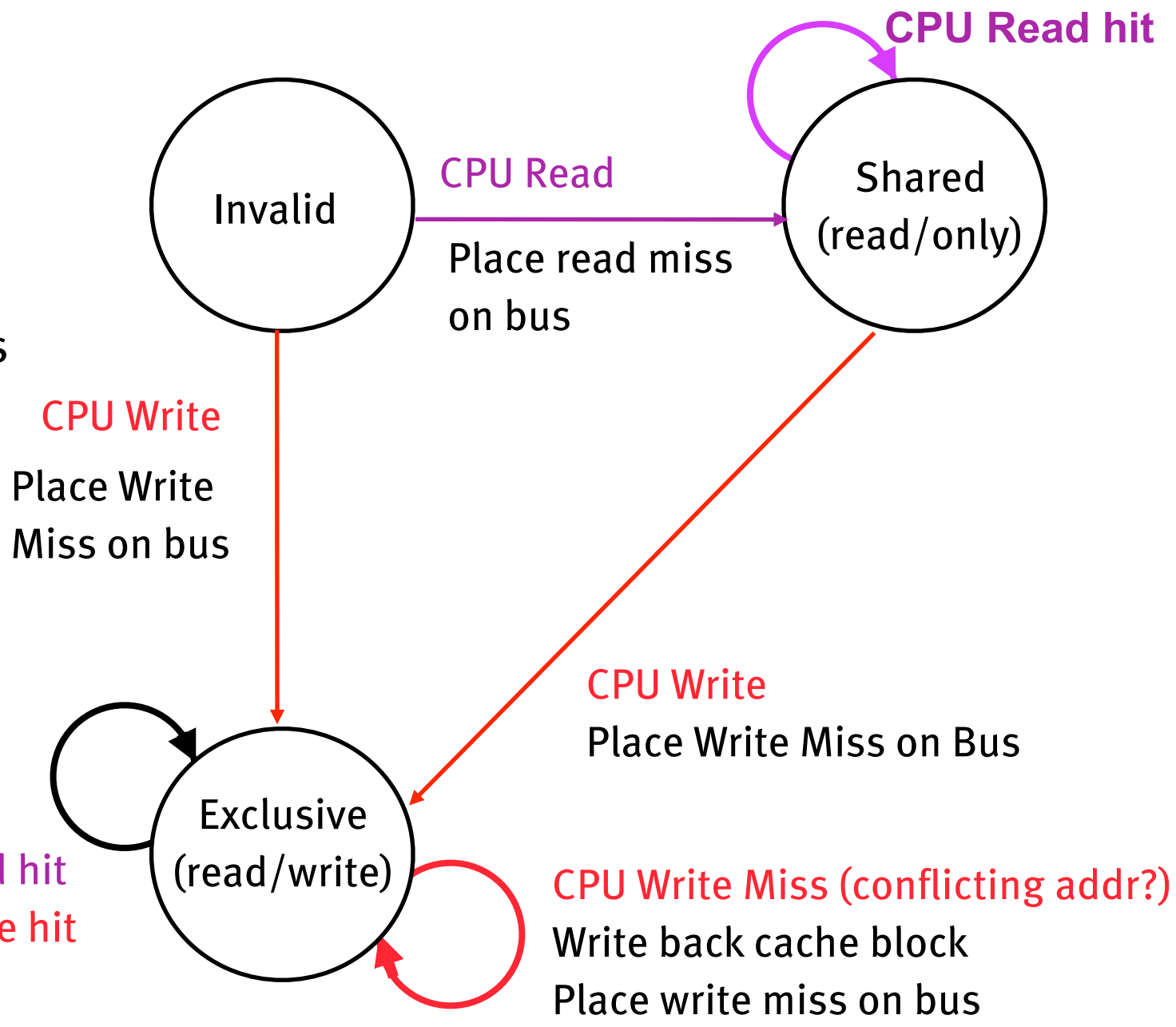
- Snooping coherence protocol is usually implemented by incorporating a finite-state controller in each node
- Logically, think of a separate controller associated with each cache block
  - That is, snooping operations or cache requests for different blocks can proceed independently
- In implementations, a single controller allows multiple operations to distinct blocks to proceed in interleaved fashion
  - that is, one operation may be initiated before another is completed, even through only one cache access or one bus access is allowed at time

Example Write Back Snoopy Protocol

- Invalidation protocol, write-back cache
  - Snoops every address on bus
  - If it has a dirty copy of requested block, provides that block in response to the read request and aborts the memory access
- Each memory block is in one state (we don't explicitly track these):
  - Clean in all caches and up-to-date in memory (Shared)
  - OR Dirty in exactly one cache (Exclusive)
  - OR Not in any caches
- Each cache block is in one state (track these):
  - Shared : block can be read
  - OR Exclusive : cache has only copy, it's writeable, and dirty
  - OR Invalid : block contains no data (in uniprocessor cache too)
- Processor registers a read miss: all caches snoop bus (and reply if necessary)
- Processor writes to clean block: Treat this as a miss

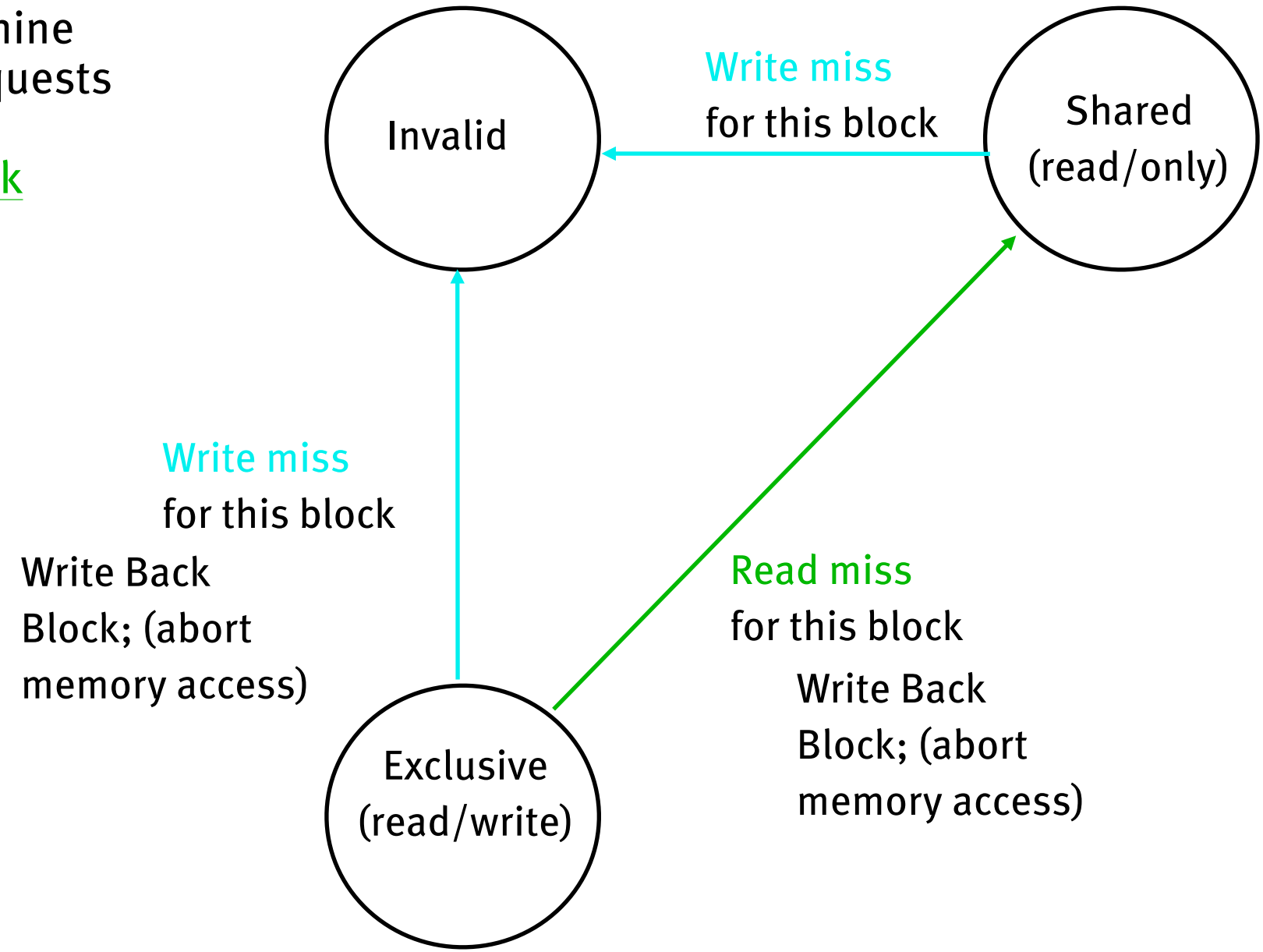
Write-Back State Machine—CPU

- State machine for CPU requests for each cache block
- Non-resident blocks invalid



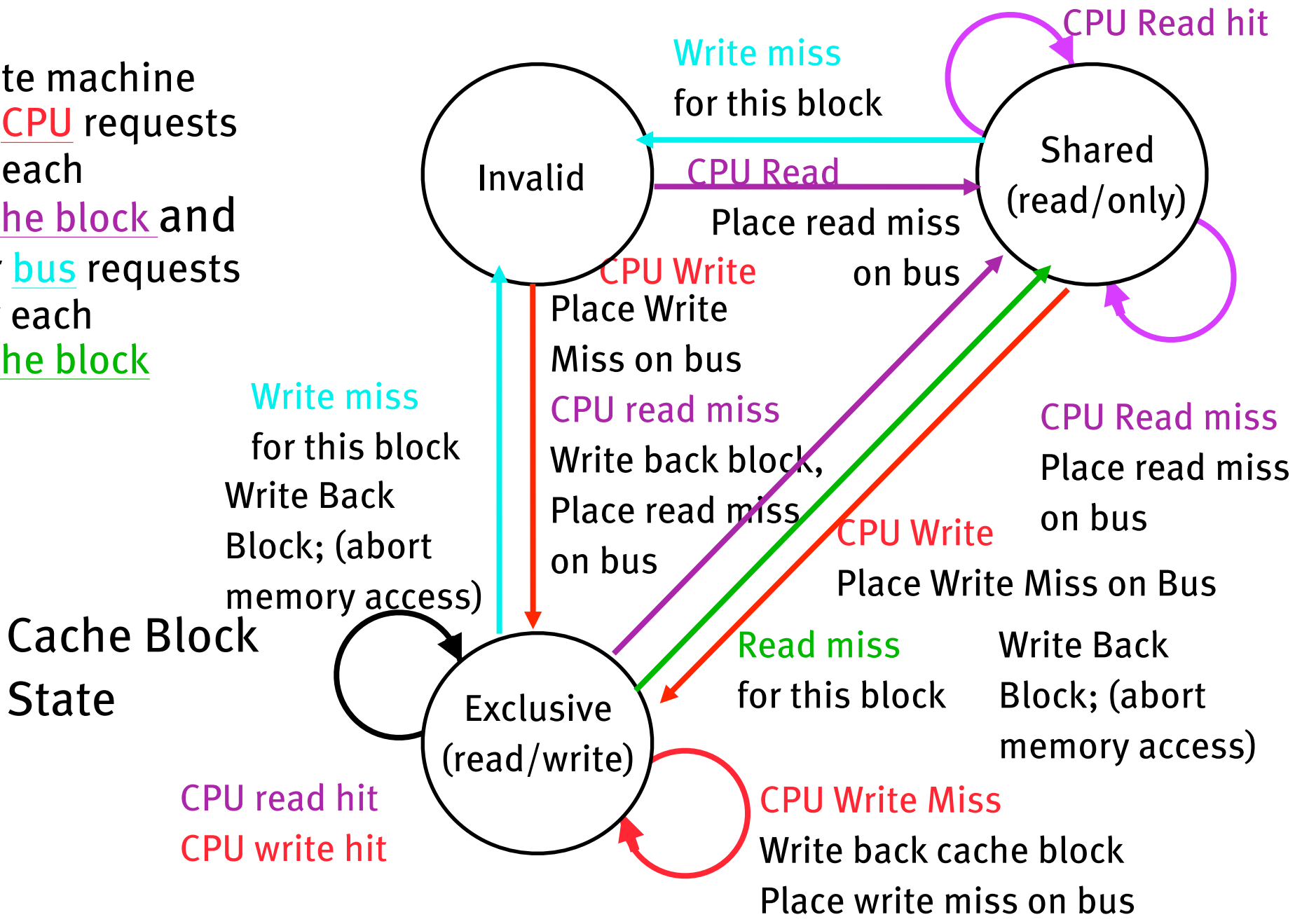
Write-Back State Machine—Bus request

- State machine for bus requests for each cache block



Write-back State Machine-III

- State machine for CPU requests for each cache block and for bus requests for each cache block



Example

	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1 Write 10 to A1												
P1: Read A1												
P2: Read A1												
P2: Write 20 to A1												
P2: Write 40 to A2												

Assumes A1 and A2 map to same cache block,
initial cache state is invalid

Example

	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1 Write 10 to A1	Excl.	A1	10				WrMs	P1	A1			
P1: Read A1												
P2: Read A1												
P2: Write 20 to A1												
P2: Write 40 to A2												

Assumes A1 and A2 map to same cache block

Example

	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1 Write 10 to A1	Excl.	A1	10				WrMs	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1												
P2: Write 20 to A1												
P2: Write 40 to A2												

Assumes A1 and A2 map to same cache block

Example

	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1 Write 10 to A1	Excl.	A1	10				WrMs	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1				Shar.	A1		RdMs	P2	A1			
	Shar.	A1	10				WrBk	P1	A1	10	A1	10
				Shar.	A1	10	RdDa	P2	A1	10	A1	10
P2: Write 20 to A1												
P2: Write 40 to A2												

Assumes A1 and A2 map to same cache block

Example

	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1 Write 10 to A1	Excl.	A1	10				WrMs	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1				Shar.	A1		RdMs	P2	A1			
	Shar.	A1	10				WrBk	P1	A1	10	A1	10
				Shar.	A1	10	RdDa	P2	A1	10	A1	10
P2: Write 20 to A1	Inv.			Excl.	A1	20	WrMs	P2	A1		A1	10
P2: Write 40 to A2												

Assumes A1 and A2 map to same cache block

Example

	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1 Write 10 to A1	Excl.	A1	10				WrMs	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1				Shar.	A1		RdMs	P2	A1			
	Shar.	A1	10				WrBk	P1	A1	10	A1	10
				Shar.	A1	10	RdDa	P2	A1	10	A1	10
P2: Write 20 to A1	Inv.			Excl.	A1	20	WrMs	P2	A1		A1	10
P2: Write 40 to A2							WrMs	P2	A2		A1	10
				Excl.	A2	40	WrBk	P2	A1	20	A1	20

Assumes A1 and A2 map to same cache block,
but A1 != A2

Performance of Symmetric Shared-Memory Multiprocessors

- Cache performance is combination of
  - Uniprocessor cache miss traffic
  - Traffic caused by communication
    - Results in invalidations and subsequent cache misses
- 4th C: coherence miss
  - Joins 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 block
  - Reads by another CPU of modified block in different cache
  - Miss would still occur if block size were 1 word
- False sharing misses when a block is invalidated because some word in the block, 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
  - Block is shared, but no word in block is actually shared
    ⇒ miss would not occur if block size were 1 word

Coherency Miss Example

- False sharing misses when a block is invalidated because some word in the block, 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
  - Block is shared, but no word in block is actually shared
    ⇒ miss would not occur if block size were 1 word
- Block b = [b0 b1 b2 b3] is in caches of processor P1 and P2 (both in “shared” state)
  - P1 modifies b0
  - P2 reads b2

Review

- Caches contain all information on state of cached memory blocks
- Snooping cache over shared medium for smaller MP by invalidating other cached copies on write
- Sharing cached data $\Rightarrow$ Coherence (values returned by a read), Consistency (when a written value will be returned by a read)

A Cache Coherent System Must:

- Provide set of states, state transition diagram, and actions
- Manage coherence protocol
  - (o) Determine when to invoke coherence protocol
  - (a) Find info about state of block in other caches to determine action
    - whether need to communicate with other cached copies
  - (b) Locate the other copies
  - (c) Communicate with those copies (invalidate/update)
- (o) is done the same way on all systems
  - state of the line is maintained in the cache
  - protocol is invoked if an “access fault” occurs on the line
- Different approaches distinguished by (a) to (c)

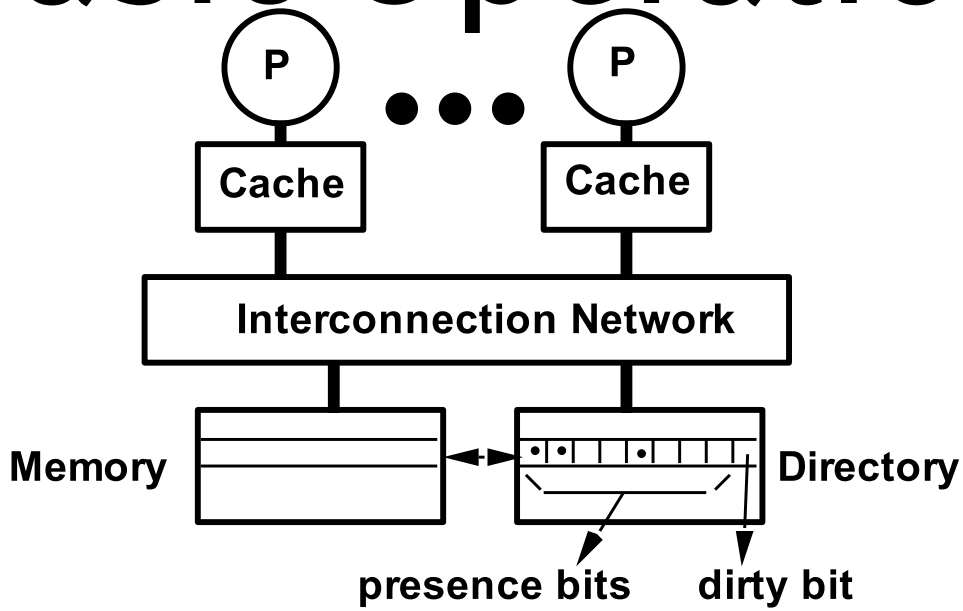
Bus-based Coherence

- All of (a), (b), (c) done through broadcast on bus
  - faulting processor sends out a “search”
  - others respond to the search probe and take necessary action
- Could do it in scalable network too
  - broadcast to all processors, and let them respond
- Conceptually simple, but broadcast doesn’t scale with *p* processors
  - on bus, bus bandwidth doesn’t scale
  - on scalable network, every fault leads to at least *p* network transactions
- Scalable coherence:
  - can have same cache states and state transition diagram
  - different mechanisms to manage protocol

Scalable Approach: Directories

- Every memory block has associated directory information
  - keeps track of copies of cached blocks 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

Basic Operation of Directory



- k processors.
- With each cache-block in memory: k presence-bits, 1 dirty-bit
- With each cache-block in cache: 1 valid bit, and 1 dirty (owner) bit

- Read from main memory by processor i:
  - If dirty-bit OFF then { read from main memory; turn p[i] ON; }
  - If dirty-bit ON then { recall line from dirty proc (cache state to shared); update memory; turn dirty-bit OFF; turn p[i] ON; supply recalled data to i;}
- Write to main memory by processor i:
  - If dirty-bit OFF then { supply data to i; send invalidations to all caches that have the block; turn dirty-bit ON; turn p[i] ON; ... }
  - • ...

Directory Protocol

- Similar to Snoopy Protocol: Three states
  - Shared: ≥ 1 processors have data, memory up-to-date
  - Uncached (no processor has it; not valid in any cache)
  - Exclusive: 1 processor (owner) has data; memory out-of-date
- In addition to cache state, must track which processors have data when in the shared state (usually bit vector, 1 if processor has copy)
  - Why isn't this necessary for snoopy protocol?
- Keep it simple:
  - Writes to non-exclusive data => write miss
  - Processor blocks until access completes
  - Assume messages received and acted upon in order sent

Directory Protocol

- No bus and don’t want to broadcast:
  - interconnect no longer single arbitration point
  - all messages have explicit responses
- Terms: typically 3 processors involved
  - Local node where a request originates
  - Home node where the memory location of an address resides
  - Remote node has a copy of a cache block, whether exclusive or shared
- Example messages on next slide (next lecture):
  P = processor number, A = address