CS4223
Cache Coherence Protocols

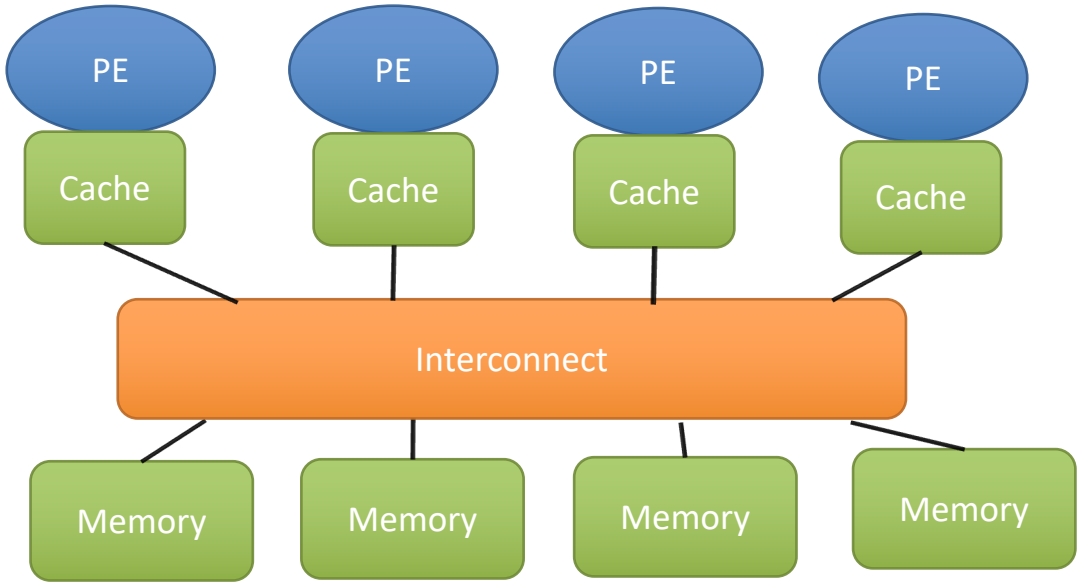
Trevor E. Carlson
National University of Singapore
tcarlson@comp.nus.edu.sg
(Slides from Tulika Mitra)

Learning Objectives

- Cache coherence protocols
  - Formal definition of coherence
  - Invalidation protocols
    - MSI, MESI
  - Update protocol
    - Dragon
  - Directory-based cache coherence

CC-UMA or SMP

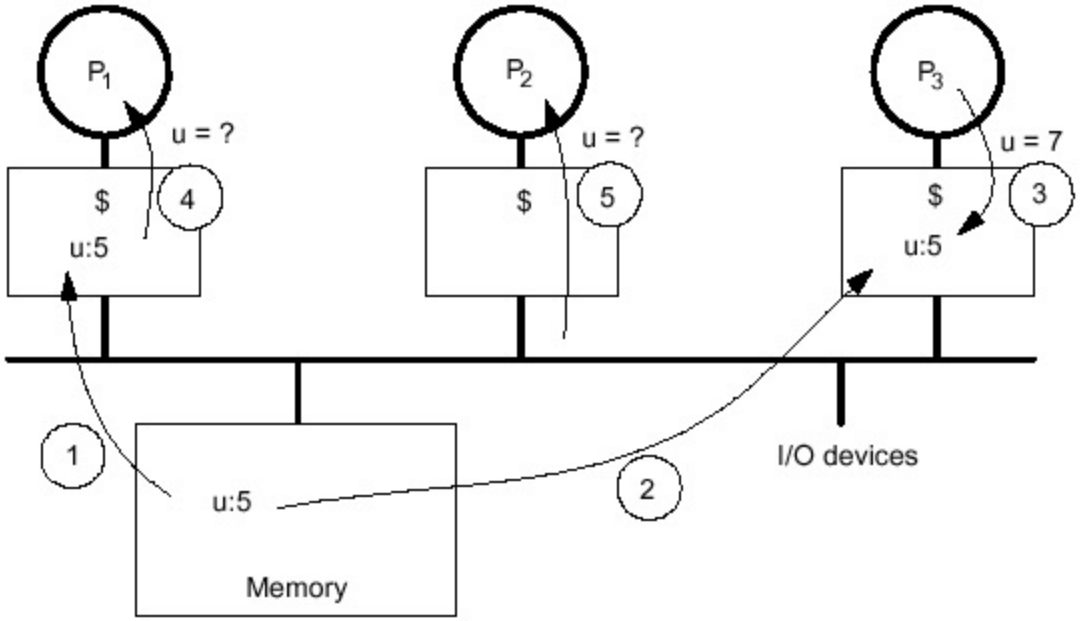
- Cache-Coherent Uniform Memory Access
- Also known as Symmetric Multiprocessor (SMP)
- Examples: Intel quad-core, Sun Starfire



Cache Coherence

- Caches reduce data access time and bandwidth requirement on the interconnect
- But private caches create problem
  - Copies of a variable in multiple caches
  - A write by one processor may not become visible to others
    - They may keep accessing stale data in their caches
  - Cache coherence problem

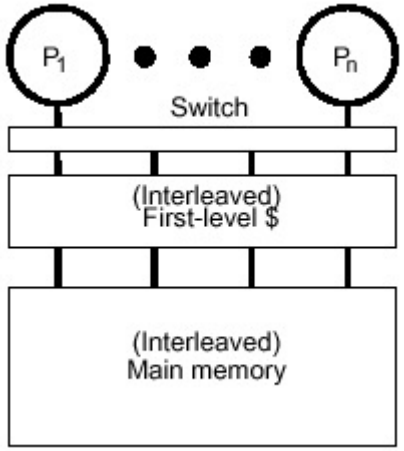
Cache Coherence Problem



- Processors see different values of u after event 3
- With write-back caches, main memory will have the stale value till the entry is replaced

Why not have a shared cache?

- High bandwidth and negative interference (conflicts)
- Increased latency due to cache size



Coherent Memory System: Intuition

- Reading a location should return the latest value written by any process
- Easy in uni-processor except for I/O
  - Infrequent, so software solutions work; un-cacheable memory, flush pages, pass I/O data through caches
- Processes run on multiple processors should behave as if they were interleaved on uni-processor
- Software solution does not work
  - Pervasive
  - Performance critical
  - Must be treated as basic hardware design issue

Problems with the Intuition

- Recall: Value returned by read should be last value written
  - But “last” is not well-defined
- In sequential case, last defined in terms of program order
- In parallel case, program order defined within a process, but need to make sense of orders across processes
- Must define a meaningful semantics

Sharpening the Intuition

- Imagine a shared memory and no caches
  - Every read and write to a location accesses the same physical address
- Memory imposes a serial or total order on operations to a particular location
  - Operations to the location from a given processor are in program order
  - Order of operations to the location from different processors is some interleaving that preserves the individual program orders
- "Last" means most recent in a hypothetical serial order maintaining these properties

Formal Definition of Coherence

- A memory system is coherent if the results of any execution of a program are such that for each location, it is possible to construct a hypothetical serial order of all operations to the location that is consistent with the results of execution and in which:
  - Operations issued by any particular process occur in program order
  - The value returned by a read is the value returned by the last write to that location in serial order

Example 1

(initially, U = 0)

Processor 1	**Processor 2**
U = 2	U = 1
read U (A)	read U (B)

Example 2

Processor 1	Processor 2
U = 1	read U (A)
U = 2	read U (B)

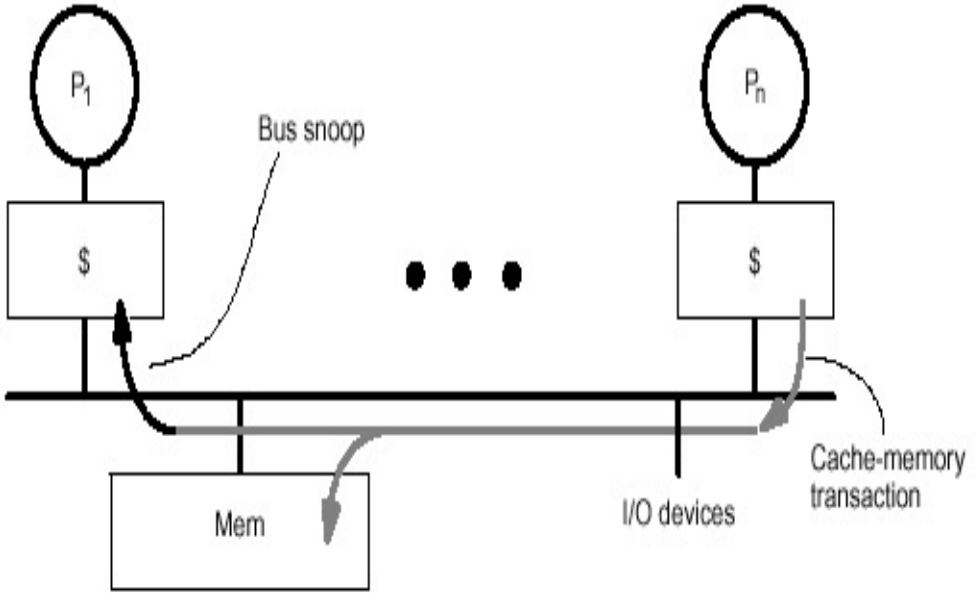
Necessary Features

- Write Propagation:
  - Value written must become visible to others
- Write Serialization:
  - Writes to a location seen in same order by all
    - If I see w1 after w2, you should not see w1 before w2
    - No need for analogous read serialization

Snooping Cache Coherence

- Bus-based cache coherence
- All the processors on the bus can observe every bus transactions (Write Propagation)
- Bus transactions are visible to the processors in the same order (Write Serialization)
- All the cache controllers “snoop” on the bus and monitor the transactions
- A snooping cache controller takes action if the bus transaction is relevant to it
  - i.e., if it has the memory block in cache
- Granularity of cache coherence is cache block

Snooping Cache Coherence



Design Space for Snoopy Protocols

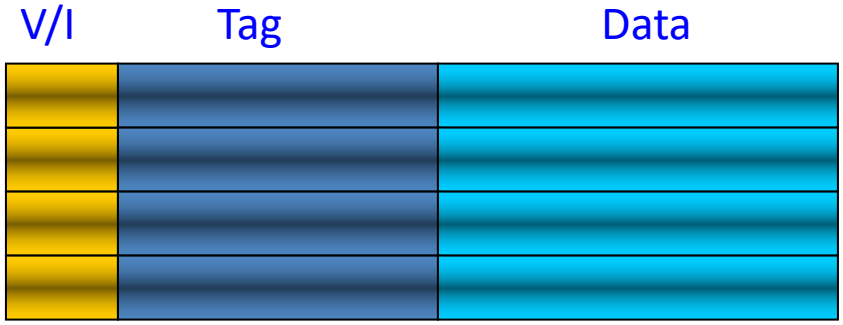
- No change in processor, main memory
- Only extend cache controller
  - Add extra states to a cache block
- Design space
  - Invalidation-based protocols
    - Invalidate all other cache copies before write
  - Update-based protocols
    - Update all copies on a write operation
  - Write-back cache
  - Write-through cache

Cache Coherence Protocol

- Cache controllers receive input from processors as well as snoopers
- Cache controllers update states, respond with data, and generate new bus transactions
- Distributed algorithm: cooperating state machines
- Granularity of coherence is cache block

Basics of Write-through Cache

- Each write operation goes to main memory
  - Write-allocate: On write miss, allocate and read cache block; then write data to cache and main memory
  - Write-no-allocate: On write miss, only write to memory
  - On a write hit, data is written to both cache and memory
- Each cache block has two states:
  - valid (V) and invalid (I)

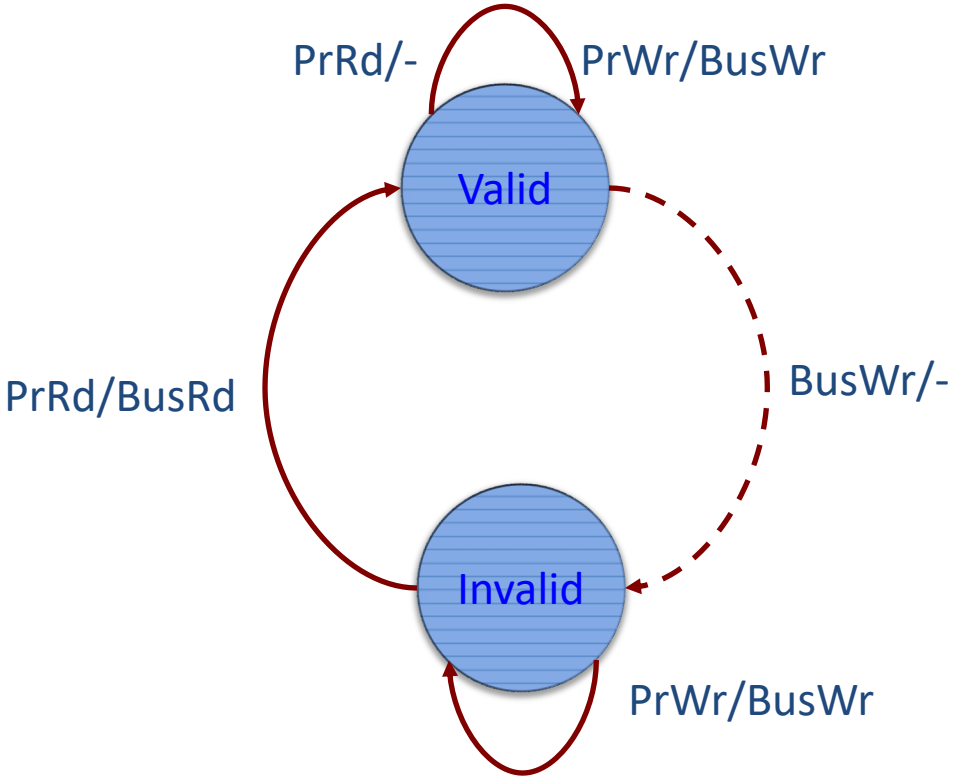


Write-through Cache Coherence

- Each block has two states
  - Valid: memory block is resident in the cache
  - Invalid: memory block is not resident in cache
- Each write operation causes a bus transaction

  (write propagation)
- If a snooping cache has a copy of the block, it can
  - Invalidate its copy
  - Update its copy
- Next time the processor accesses the cache block, it will get the most recent copy
  - The copy has been invalidated => cache miss => new copy
  - The copy has been updated

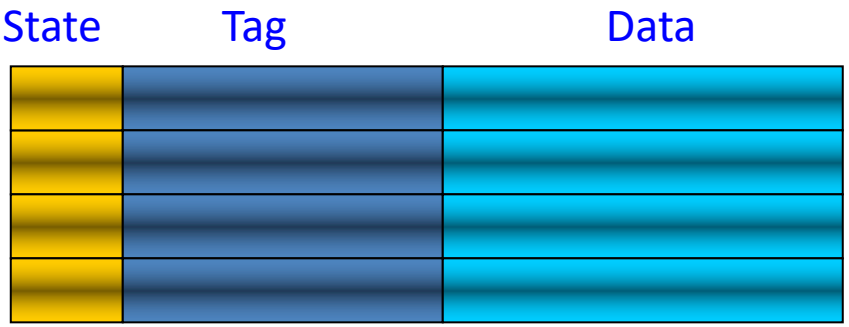
Write-through, Write-no-allocate State Transition Diagram



- Multiple simultaneous readers may coexist
- A write will eliminate all other cached copies

Basics of Write-back Cache

- Each cache block has three states
  - Valid, invalid, and dirty
- Write hit: write to the cache and set state as dirty
- Write back a dirty block when it is evicted from cache
- Write miss: often use write-allocate policy
- Significantly reduces memory bandwidth



Extension for Cache Coherence

- Dirty implies exclusive ownership
  - Only this cache has a valid copy
  - Free to modify this copy without notifying anybody else
  - Responsible to supply this copy when requested

- Valid means other caches may have copies too
  - Invalidation-based protocol: Before modifying, the cache should communicate with other caches to obtain an exclusive ownership i.e. invalidate other copies
  - Update-based protocol: A write to a "valid" copy should generate a bus transaction so that other caches can update their copies

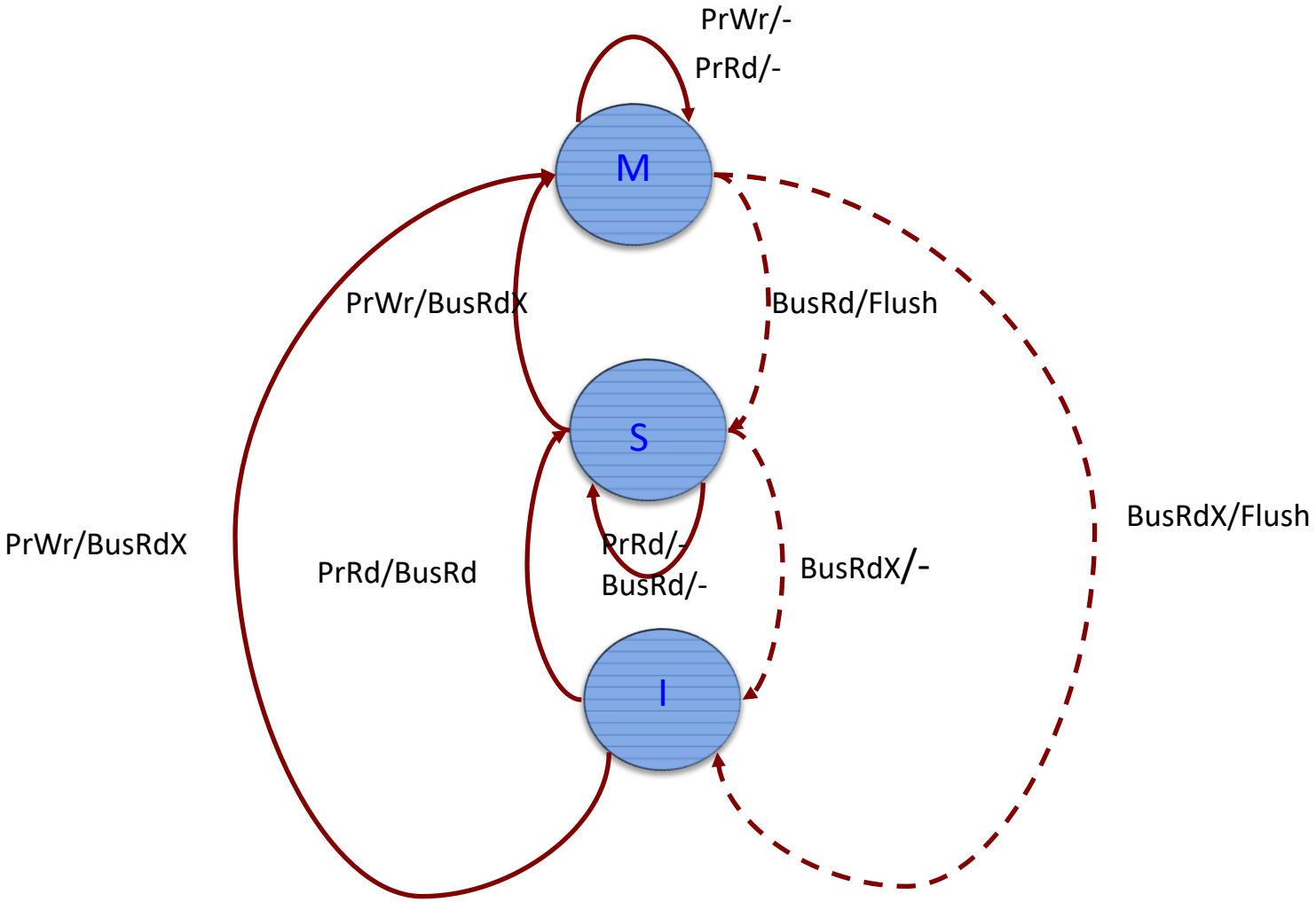
Invalidation versus Update

- Invalidation advantages:
  - First write gets exclusive ownership, other writes local
  - Hence, multiple writes by the same processor does not generate useless traffic
    - In update protocol, it will generate multiple update transactions
- Update advantages:
  - Other processor don't miss on next access, reduced latency
    - In invalidation, they would miss causing more transactions
  - Single bus transaction to update several cache copies can save bandwidth; only the word written is transferred, not the whole block
- Detailed trade-off complex; Invalidation protocols more popular

MSI Invalidation Protocol

- Three States
  - Invalid (I)
  - Shared (S): one or more copies
  - Dirty or Modified (M): one only
- Processor Events
  - PrRd (read)
  - PrWr (write)
- Bus Transactions
  - BusRd: asks for a copy with no intent to modify
  - BusRdX: asks for copy with intent to modify
  - BusWB: updates memory

MSI State Transition Diagram



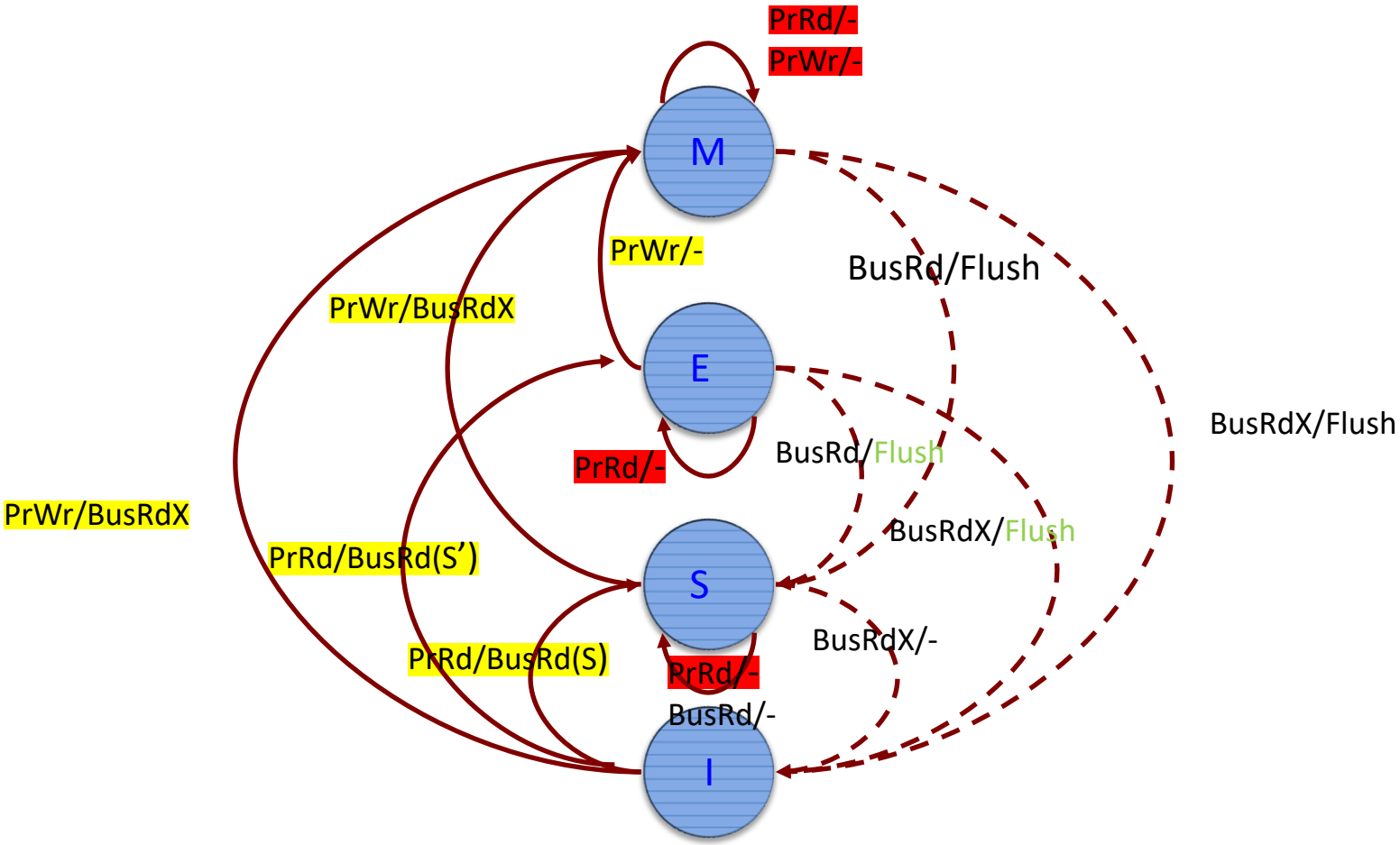
Lower-Level Protocol Choices

- BusRd observed in M state: what transition to make?
- Depends on expectation of access patterns
  - S: I will read again soon, rather than others writing
    - Good for mostly read data
    - What about migratory data?
      - I read and write, you read and write, X reads and writes
      - Better to go to I state, so I don't need to invalidate on your write
    - Synapse transitions to I state
    - Sequent Symmetry and MIT Alewife use adaptive protocol

MESI (4-state) Invalidation Protocol

- Problem with MSI protocol
  - Read and modify data in two bus Tx, even without sharing
  - BusRd (I->S) followed by BusRdX (S->M)
- Add exclusive/exclusive-clean/exclusive-unowned state: only this cache has copy but not modified
  - Can modify without bus transaction
  - Need not supply data upon read request for the block
  - Memory up-to-date, cache is not owner
- States:
  - Invalid
  - Exclusive (only this cache has unmodified copy)
  - Shared (two or more caches have copies)
  - Modified (dirty)
- I->E on PrRd if nobody else has copy
  - Needs shared signal: wired-OR line asserted in response to BusRd

MESI State Transition Diagram



Lower-level Protocol Choices

- Who supplies data on miss when not in M state: memory or cache ?
- Illinois MESI: cache, since faster than memory
  - Cache-to-cache sharing
- Cache-to-cache sharing adds complexity
  - How does memory know it should supply data?
  - Selection algorithm if multiple caches have valid data

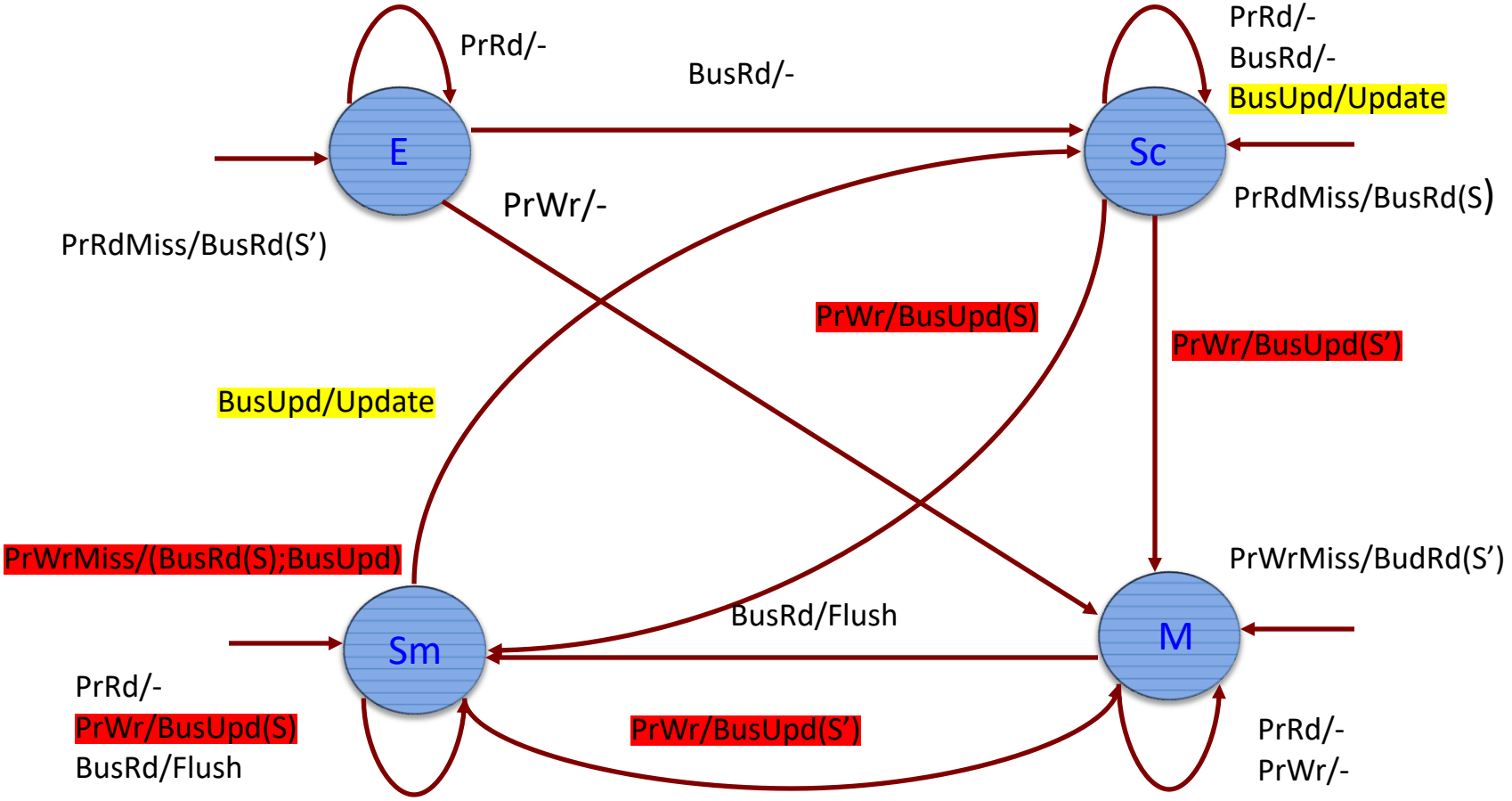
Dragon Write-back Update Protocol

- 4 states
  - Exclusive (E): this cache and memory have it
  - Shared Clean (Sc): this cache, others, maybe memory, but this cache is not the owner
  - Shared Modified (Sm): this cache and others, but not memory, and this cache is the owner
  - Modified or Dirty (D): this cache and nobody else
- Only one cache can be in Sm state for a block
- No invalid state
  - If in cache, cannot be invalid
  - If not present in cache, can view as not present or invalid state

New Actions

- Processor Events: PrRdMiss, PrWrMiss
  - Introduced to specify actions when block not present in cache
- Bus Transaction: BusUpd
  - Broadcasts single word written on bus; updates other relevant caches

Dragon State Transition Diagram



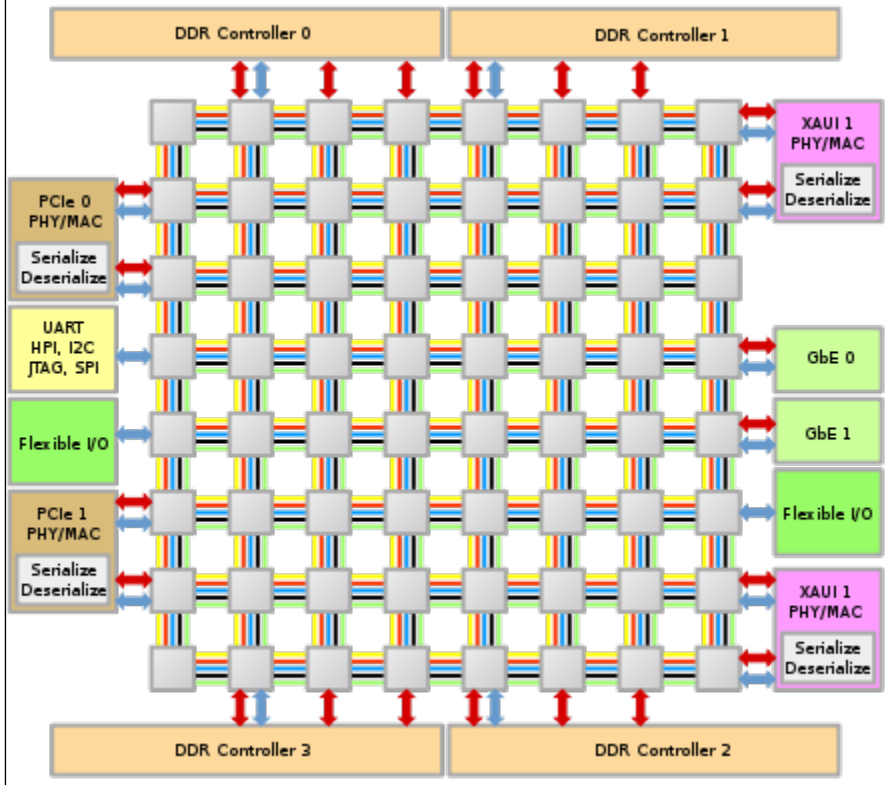
Lower-level Protocol Choices

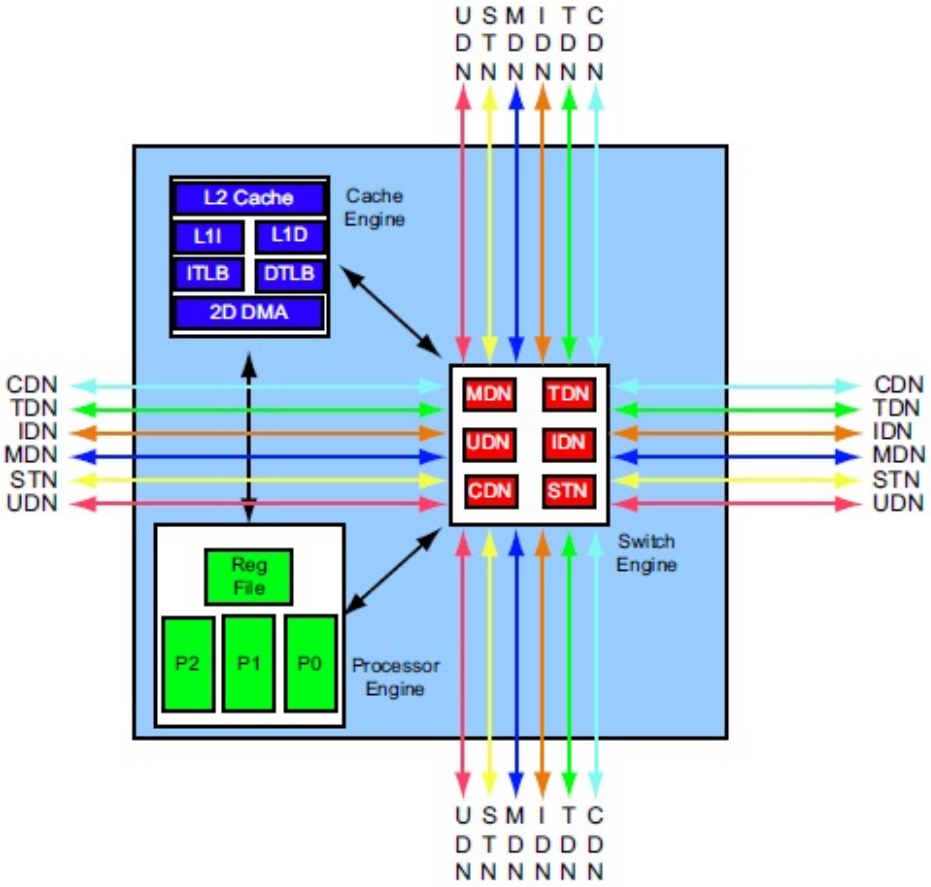
- Can shared-modified state be eliminated?
  - If update memory as well on BusUpd transaction (DEC Firefly)
  - Dragon protocol assumes DRAM memory slow to update
- Should replacement of an Sc block be broadcast ?
  - Would allow last copy to go to E state and not generate updates
  - Replacement bus transaction is not on critical path, later updates may be

Scalable Cache Coherence

- PEs connected via interconnection network
- Memory can be physically shared or physically distributed (UMA / NUMA)
- Bus-based protocols will require a broadcast
  - Does not scale with number of processors

CC-UMA: Scaling to Many Core Tilera



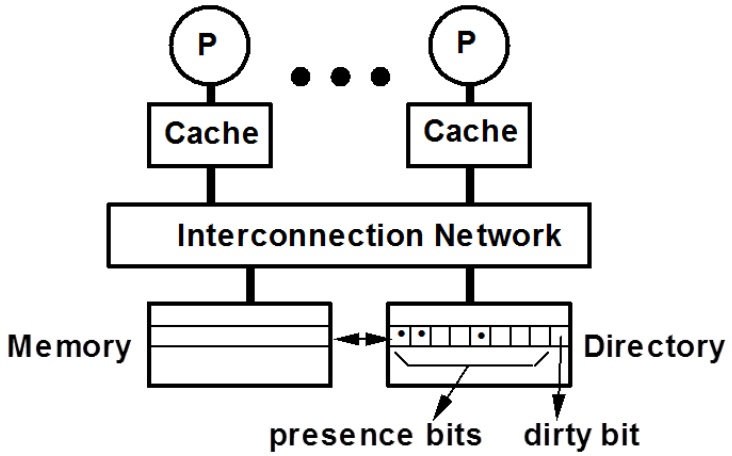


Scalable Approach: Directories

- Every memory block has associated directory information
  - Keeps track of copies of cached blocks and their states
  - On miss, find directory entry, look it up, and communicate only with the nodes that have copies, but only if necessary
  - In scalable networks, communication with directory and copies is through network transactions
- Many alternatives for organizing directory information

Directory contents

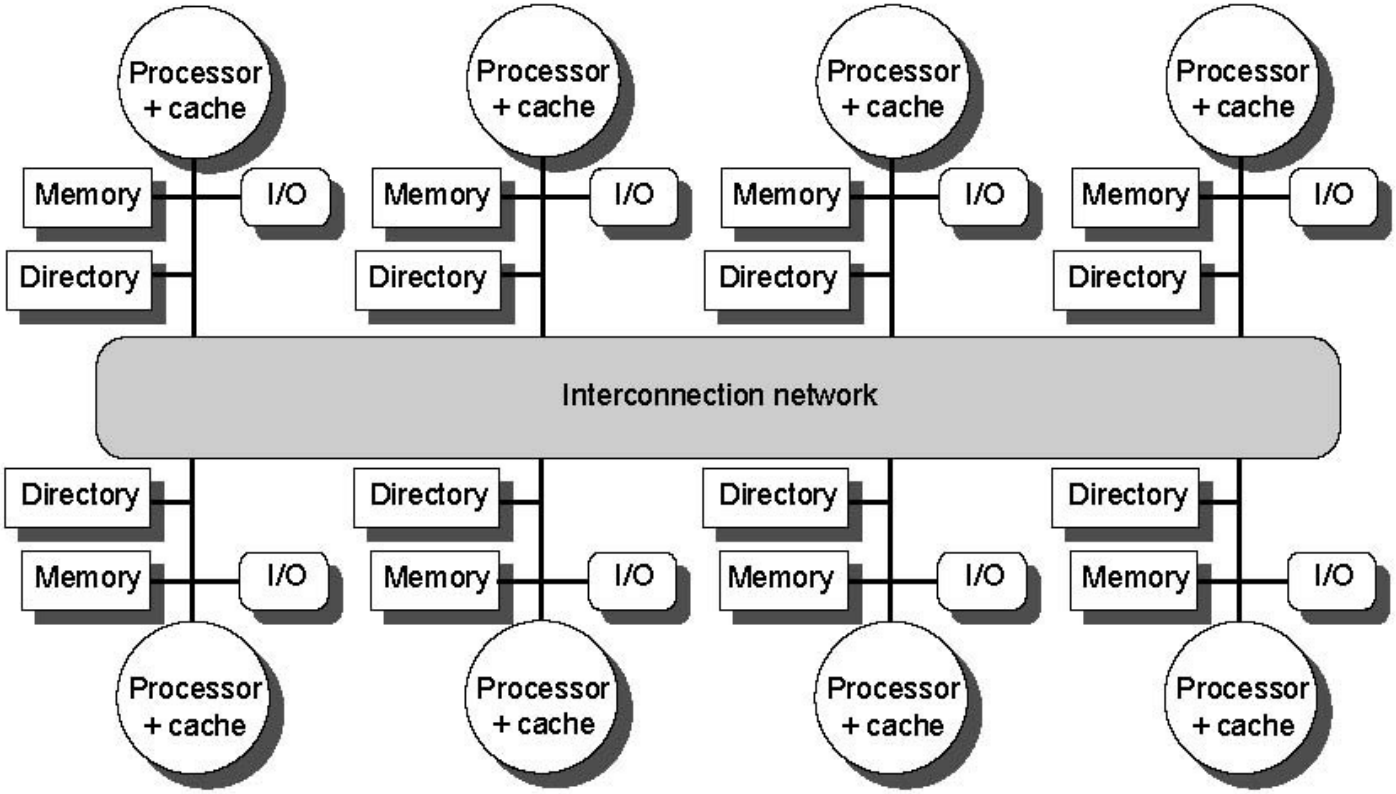
- For each memory block, the directory shows:
  - is it cached?
  - if cached, where?
  - if cached, clean or dirty?
- Full directory has complete info for all blocks
  - n processor coherence => n bit vector
  - bit i set => block is cached at processor i



Directory organization

- Centralized vs. distributed
  - Centralized directory becomes a bandwidth bottleneck
  - Provide a banked directory structure
    - Associate each directory bank with its memory bank
  - But: memory is distributed, so this leads to a distributed directory structure where each node holds the directory entries corresponding to the memory blocks for which it is the home

Distributed Directory Structure



Directory States

- Similar to Snoopy Protocol: Three states
  - Uncached (no processor has it; not valid in any cache)
  - Shared: ≥ 1 processors have data, memory up-to-date
  - Modified: 1 processor (owner) has data; memory out-of-date
- Bit vector tracks which processors have data when in shared state

Directory Protocol Terminology

- Terms: typically 3 processors involved
  - Local node where a request originates
  - Home node where the memory location of an address resides
  - Remote node has copy of a cache block, whether exclusive or shared
- No bus and don't want to broadcast:
  - Interconnect no longer single arbitration point
  - All messages have explicit responses

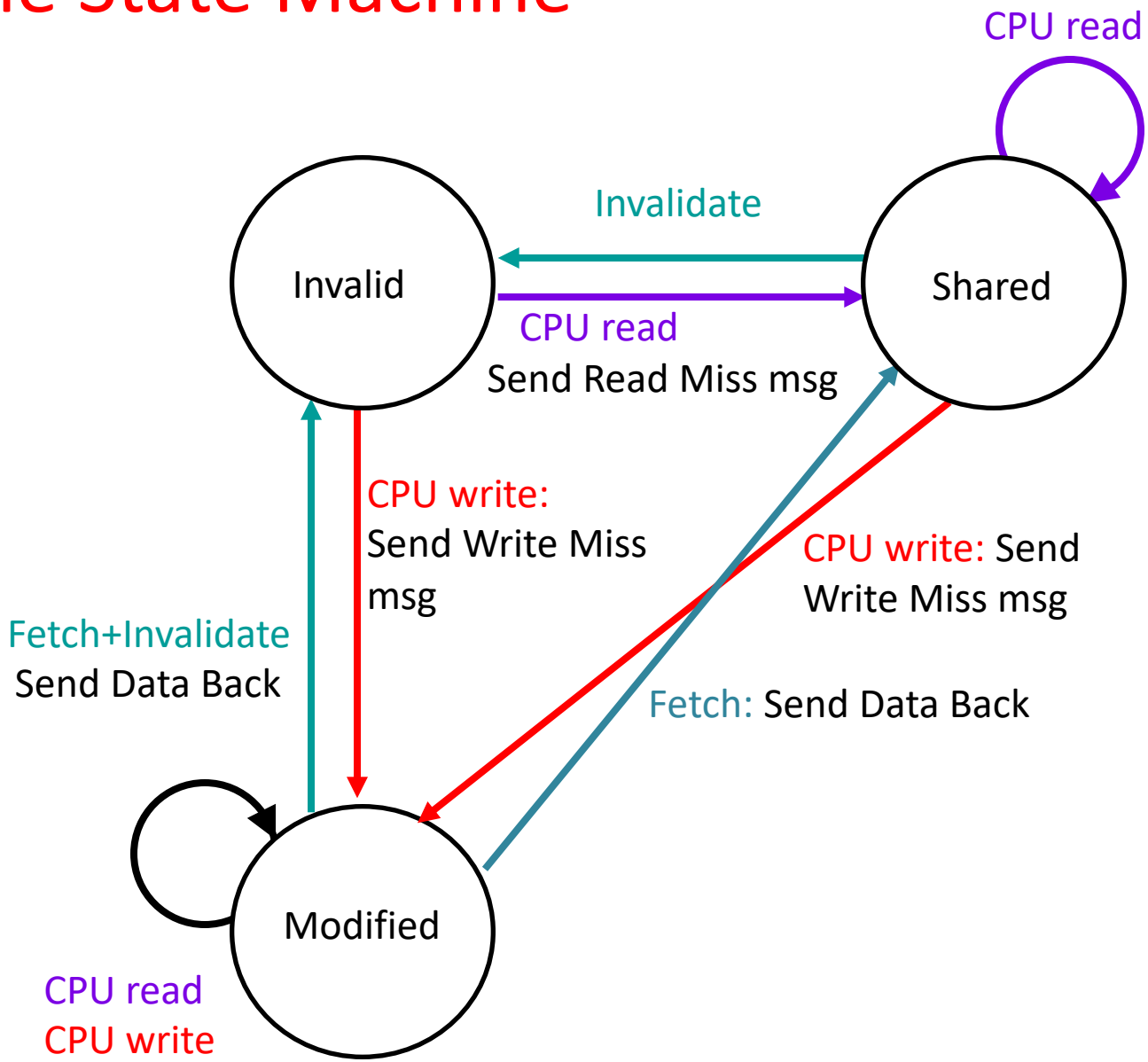
The easy cases: Local cache

- Read hit (clean or dirty) or Write hit (dirty)
  - no directory action required
  - serve from local cache

State-Transition Diagram for One Block in Directory-Based System

- Cache states identical to snoopy case
- Cache transitions very similar
- Write hit/misses that were broadcast on the bus for snooping $\Rightarrow$ explicit invalidate & data fetch requests
- Transitions caused by read miss, write miss, write hit (clean), invalidates, data fetch requests
- Messages sent to home node from local node

CPU-Cache State Machine

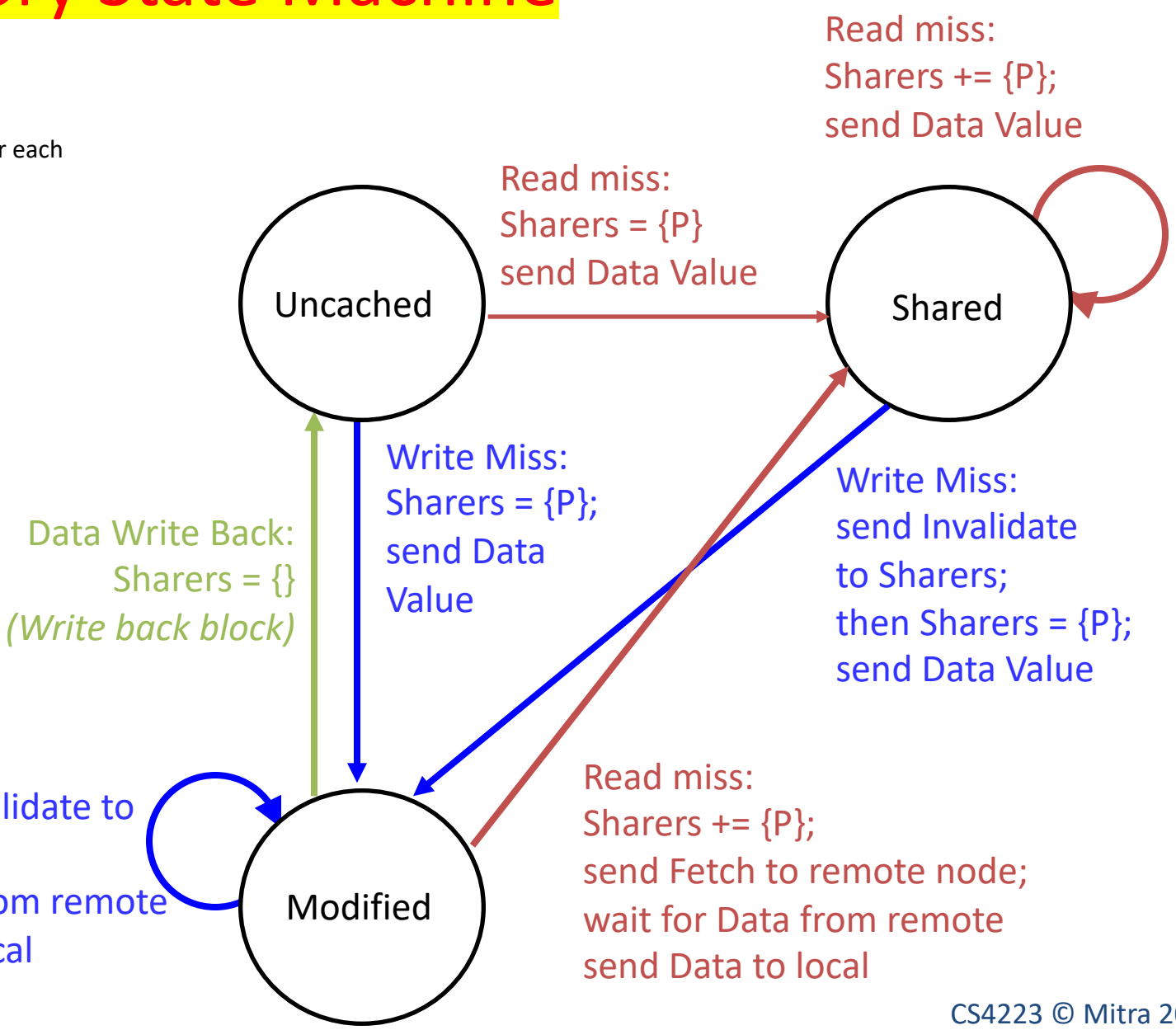


Transition Diagram for Directory

- Transition diagram for each memory block
- Two actions
  1. Update directory state
  2. Send messages to satisfy requests
- Tracks all copies of memory block
- Also update the sharing set Sharers

Directory State Machine

- State machine for *Directory* requests for each memory block
- Uncached state if in memory



Example Directory Protocol

- Uncached: copy in memory is current value; only possible requests for that block are:
  - Read miss:
    - Sent data from memory to requestor
    - requestor made only sharing node
    - state of directory block is made Shared
  - Write miss:
    - Send data from memory to requestor
    - Requestor moves to Modified state
    - State of directory block is made Modified to indicate only valid copy is cached. Sharers indicates identity of owner.

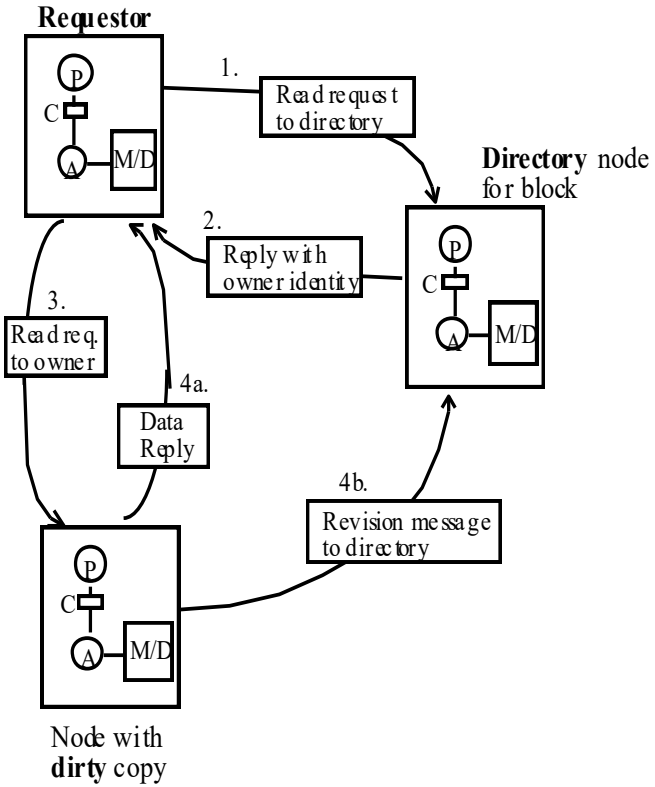
Example Directory Protocol

- Block is Shared $\Rightarrow$ memory value is up to date:
  - Read miss: requesting processor is sent data from memory & added to sharing set
  - Write miss:
    - requesting processor is sent value
    - All processors in Sharers are sent invalidate messages
    - Sharers set to just requesting processor
    - State of directory block is made Modified

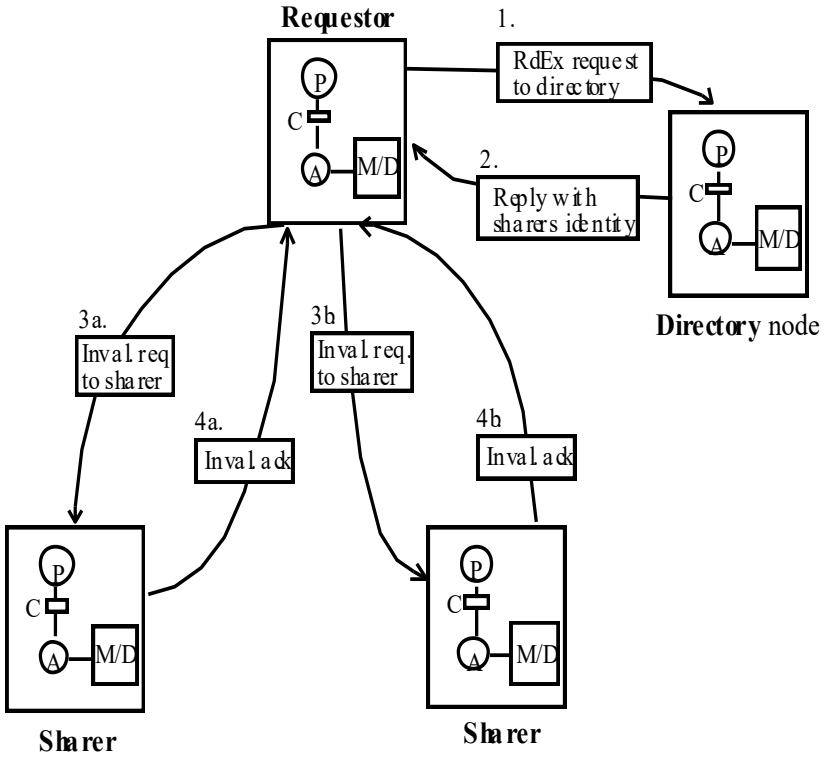
Example Directory Protocol

- Block is Modified: current value held in the cache of the processor identified by the set Sharers (the owner) ⇒ three possible directory requests:
  - Read miss: directory sends data fetch msg to owner, changing state in owner's cache to Shared and causing owner to send data to directory, where it is written to memory & sent back to requestor. Requestor added to Sharers, which still contains processor that was the owner. Directory state changes to Shared
  - Write miss: block has new owner. Message sent to old owner, causing owner to invalidate itself, send value to directory and thence to requestor, which becomes new owner. Sharers set to new owner, and directory state becomes Modified.
  - Data write-back: owner is replacing block and hence must write it back, making memory copy up-to-date. Block is now Uncached and Sharers is empty.

Basic Directory Transactions

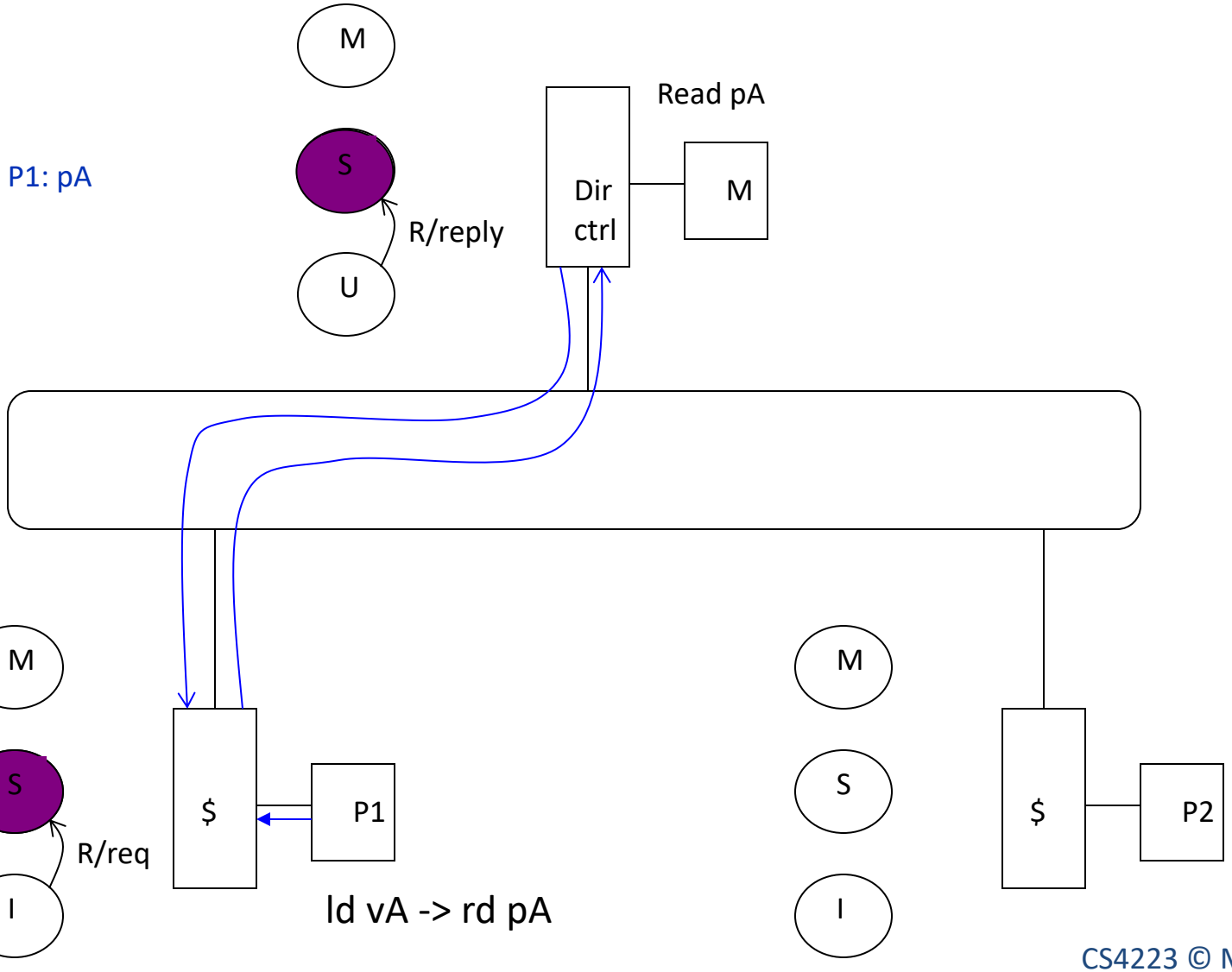


(a) Read miss to a block in dirty state

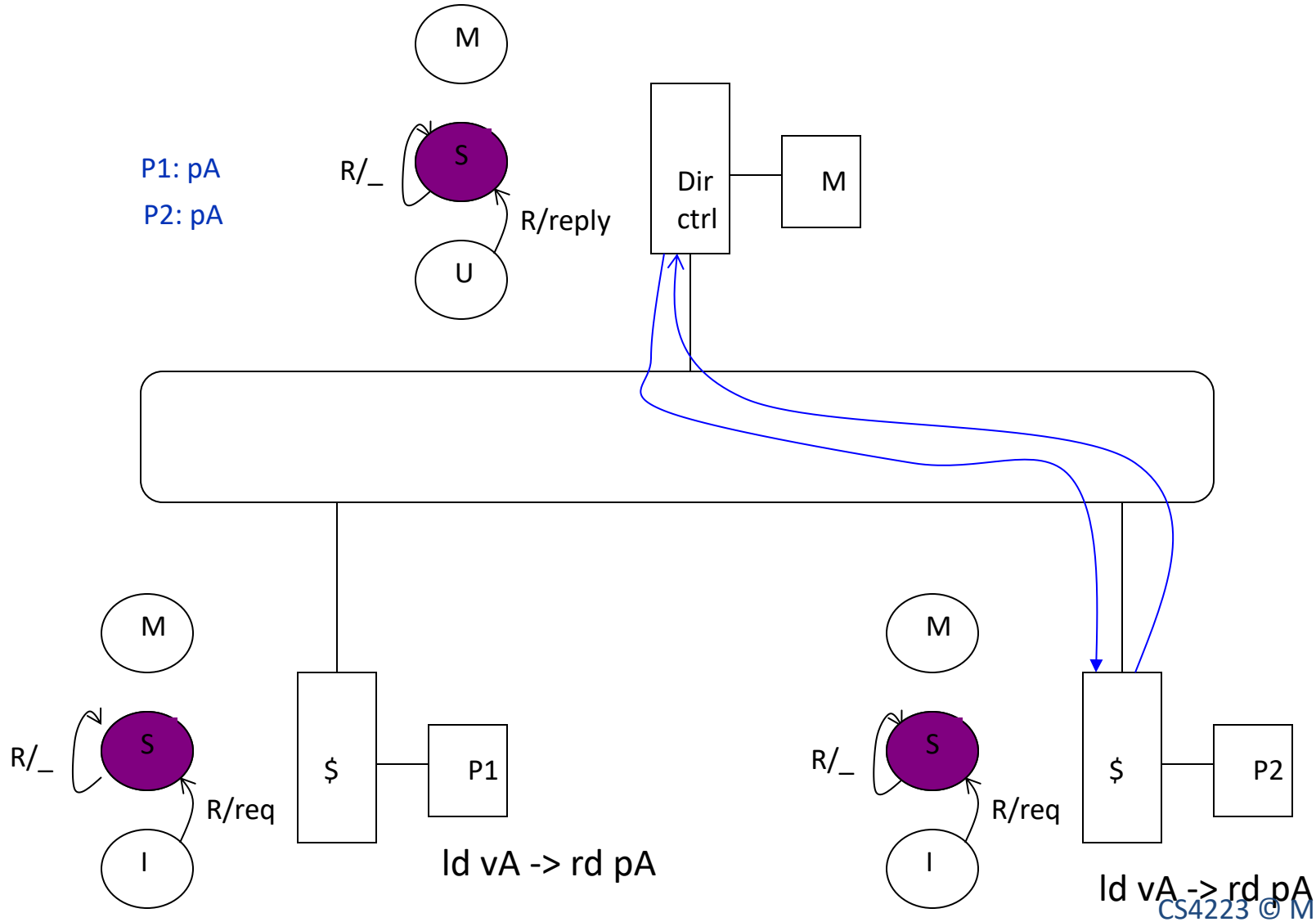


(b) Write miss to a block with two sharers

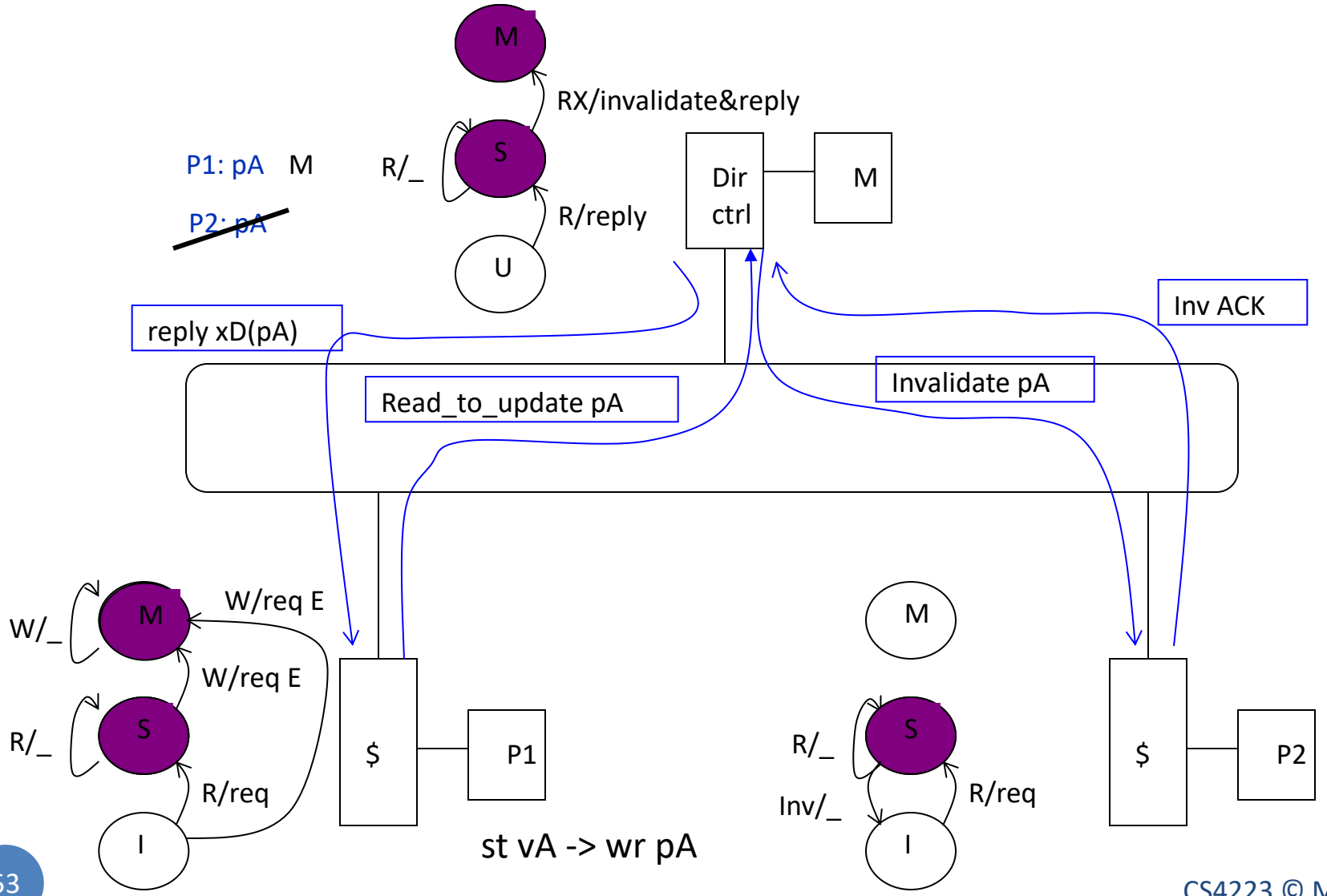
Example Directory Protocol (1st Read)



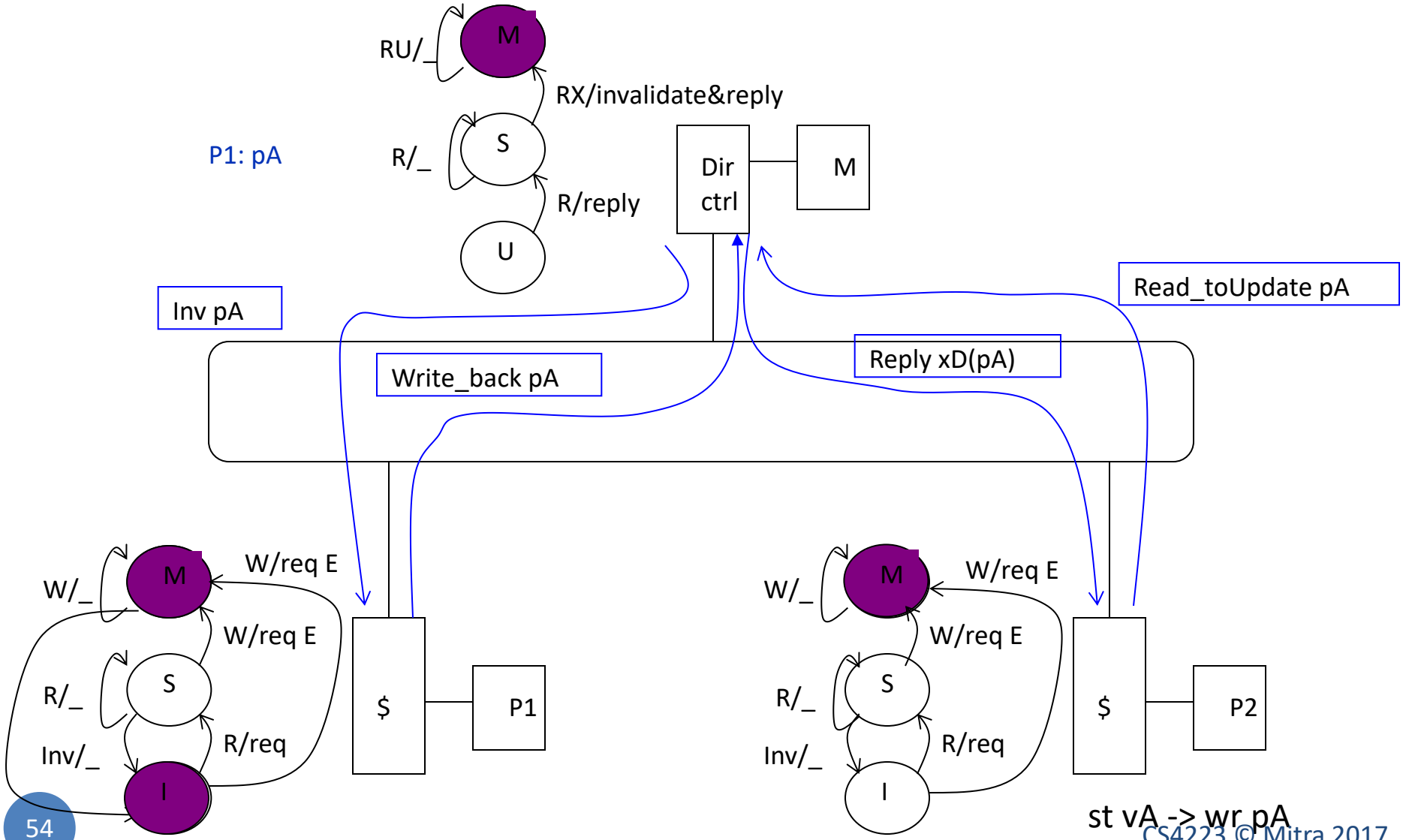
Example Directory Protocol (Read Share)



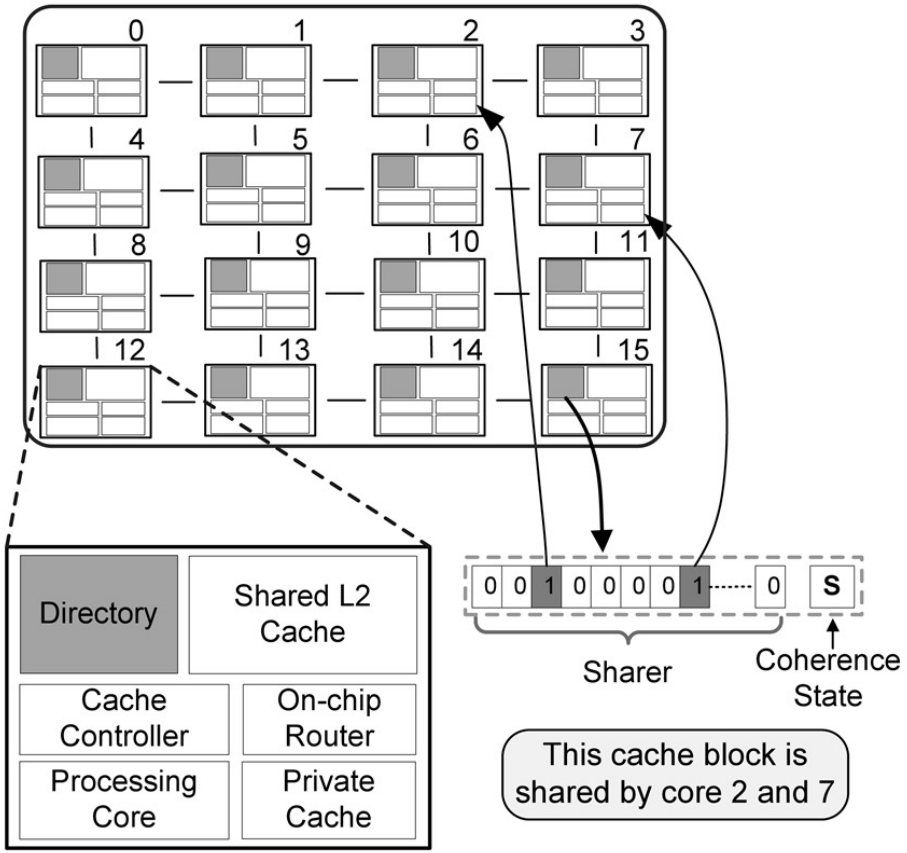
Example Directory Protocol (Wr to shared)



Example Directory Protocol (Wr to Ex)



SGI Origin 2000 Directory



(a)

L2 States	Description
Unowned (Invalid)	No core has a valid copy of this block
Shared	The block is cached at one or more nodes in the read-only state
Exclusive	The block is cached in the read-write state at exactly one node

(b)

L1 States	R/W Persimission, Status
M (Modified)	Read/Write, dirty
E (Exclusive)	Read/Write, clean
S (Shared)	Read
I (Invalid)	No permission

(c)

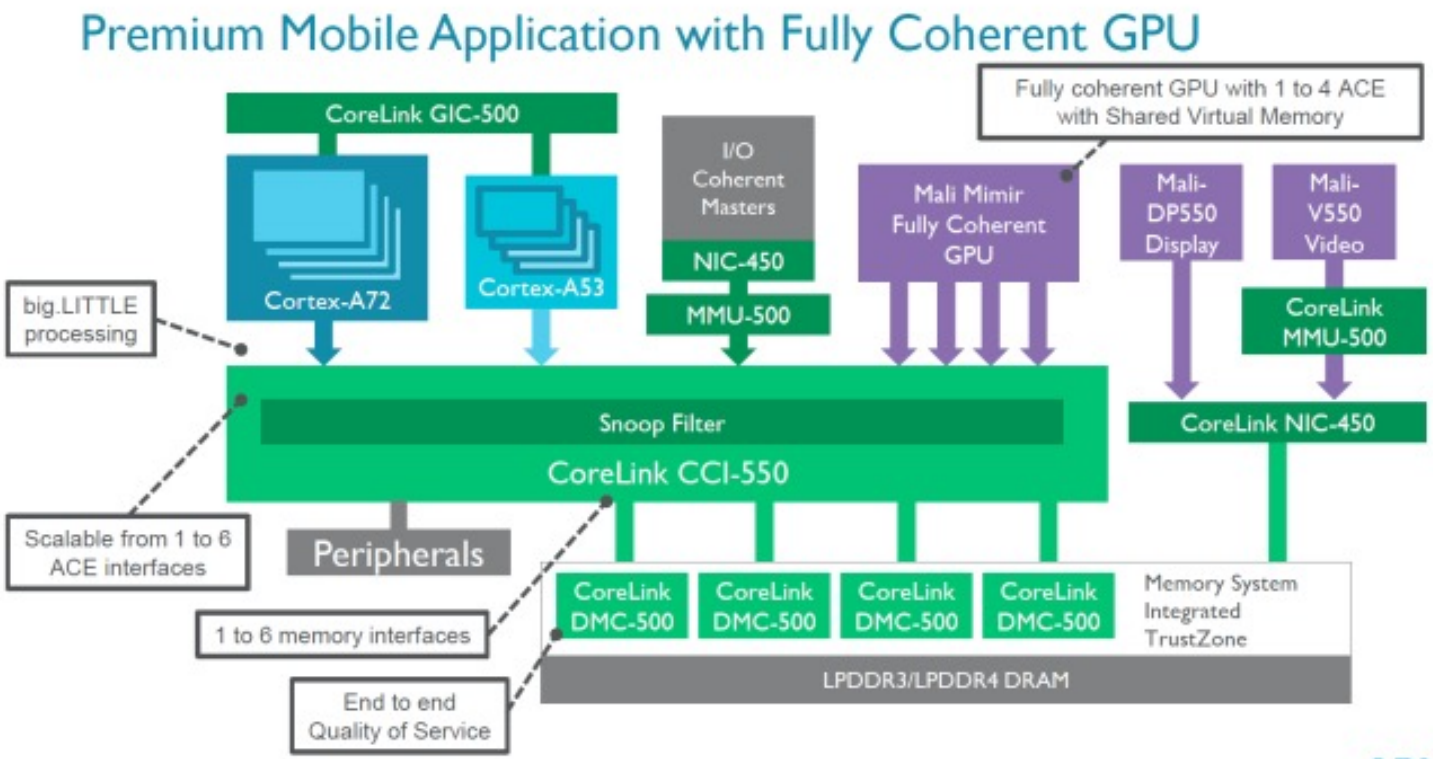
A Popular Middle Ground

- Two-level “hierarchy”
- Individual nodes are multiprocessors, connected non-hierarchically
  - mesh of SMP, or multicore chips
- Coherence across nodes is directory-based
  - Directory keeps track of nodes, not individual processors
- Coherence within nodes is snooping or directory
  - Orthogonal, but needs a good interface
- SMP on a chip: directory + snoop

Watch the writes!

- Frequently written cache lines exhibit a small number of sharers; so small number of invalidations
- Widely shared data are written infrequently; so large number of invalidations, but rare
- Synchronization variables are notorious: heavily contended locks are widely shared and written in quick succession generating a burst of invalidations; require special solutions such as queue locks or tree barriers

Going beyond CPU:
ARM CPU-GPU coherent system



Summary

- Cache coherence is compulsory for CC-UMA and CC-NUMA machines
- Invalidation-based snooping cache coherence protocols (e.g., MESI) are prevalent in multi-cores with bus
- Directory-coherence is important for many-core architectures where bus is replaced by NoC
- Directory is costly --- current research is in eliminating or compressing the directory information