CS 152 Computer Architecture and Engineering
CS252 Graduate Computer Architecture

Lecture 18 Cache Coherence

Krste Asanovic
Electrical Engineering and Computer Sciences
University of California at Berkeley

http://www.eecs.berkeley.edu/~krste
http://inst.eecs.berkeley.edu/~cs152

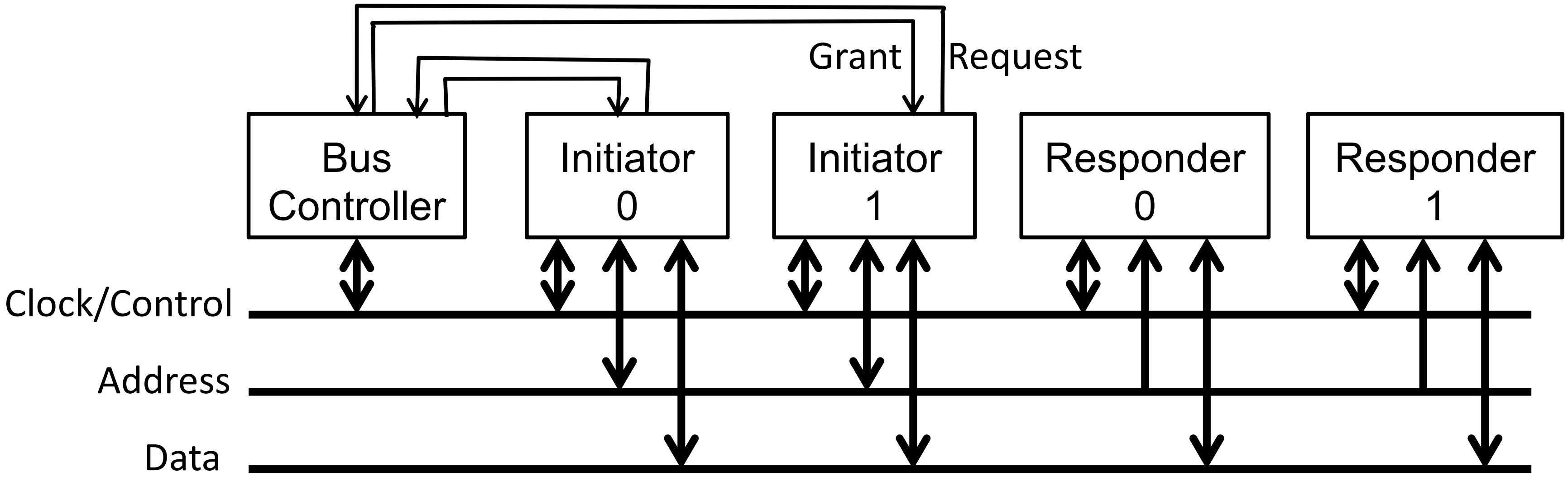
Last Time in Lecture 17

- RISC-V Standard Vectors

Adopting Alternative Terminology

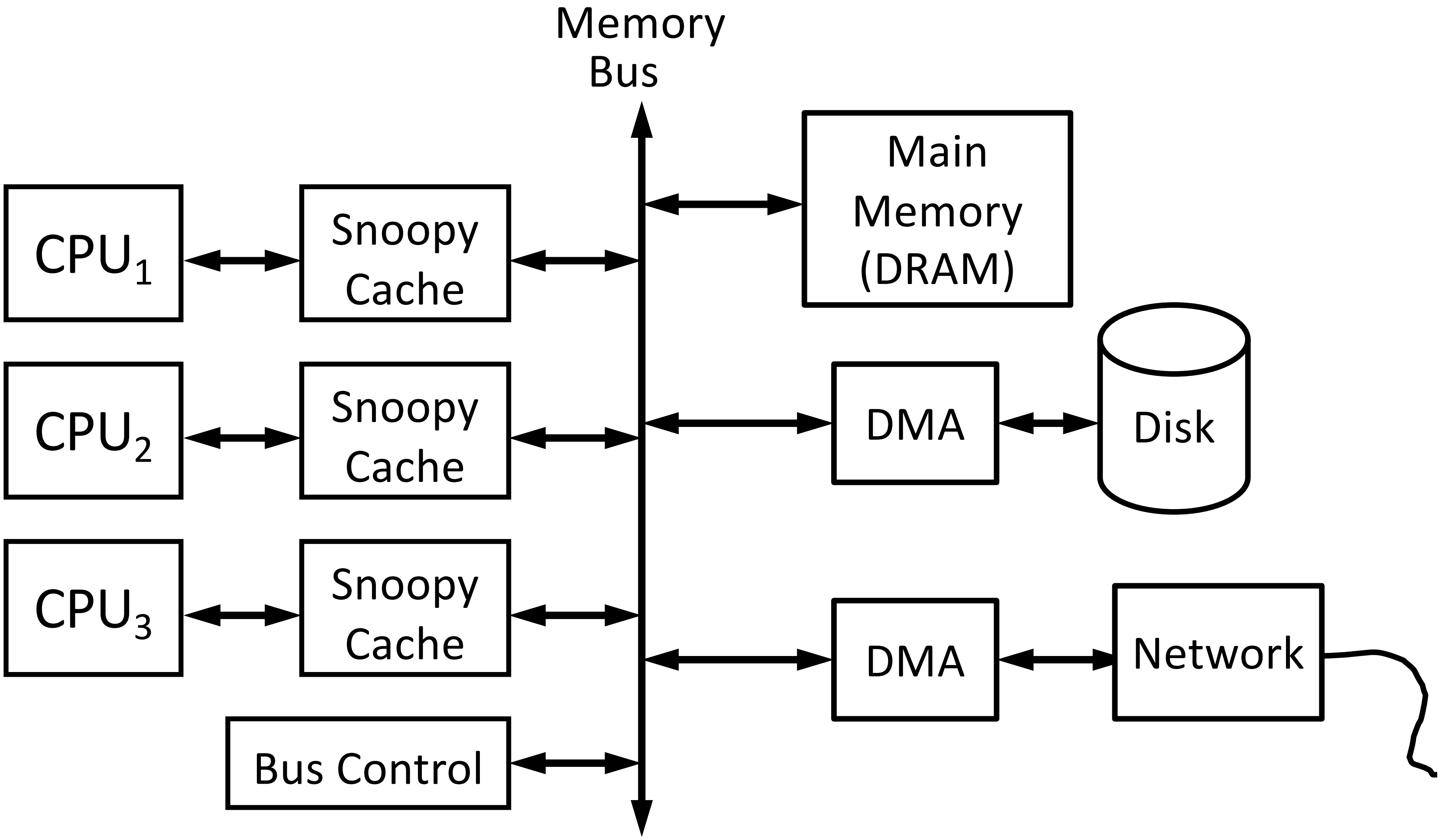
- Traditional terms “master” and “slave” replaced with “initiator” and “responder”
- Most extant documentation will refer to “bus masters” and “bus slaves”

Bus Management



- A "bus" is a collection of shared wires
  - Newer "busses" use point-point links
- At any instant, only one "initiator" can initiate a transaction by driving wires
  - Initiators arbitrate for access with requests to bus "controller"
  - Some busses only allow one initiator (in which case, it's also the controller)
- Multiple "responders" can observe and conditionally respond to the transaction on the wires
  - responders decode address on bus to see if they should respond (memory is most common responder)
  - some initiators can also act as responders

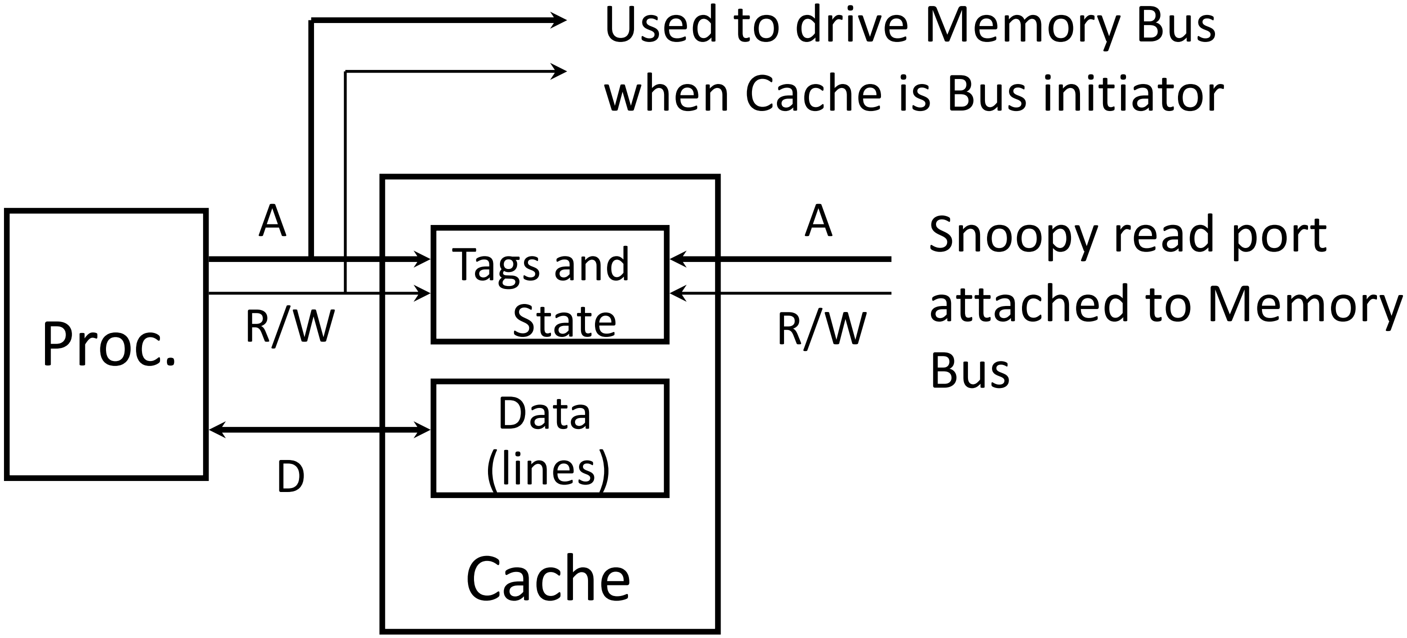
Shared-Memory Multiprocessor



Use snoopy mechanism to keep all processors' view of memory coherent

Snoopy Cache, *Goodman 1983*

- Idea: Have cache watch (or snoop upon) other memory transactions, and then "do the right thing"
- Snoopy cache tags are dual-ported



Snoopy Cache-Coherence Protocols

- Write miss:
  - the address is invalidated in all other caches before the write is performed

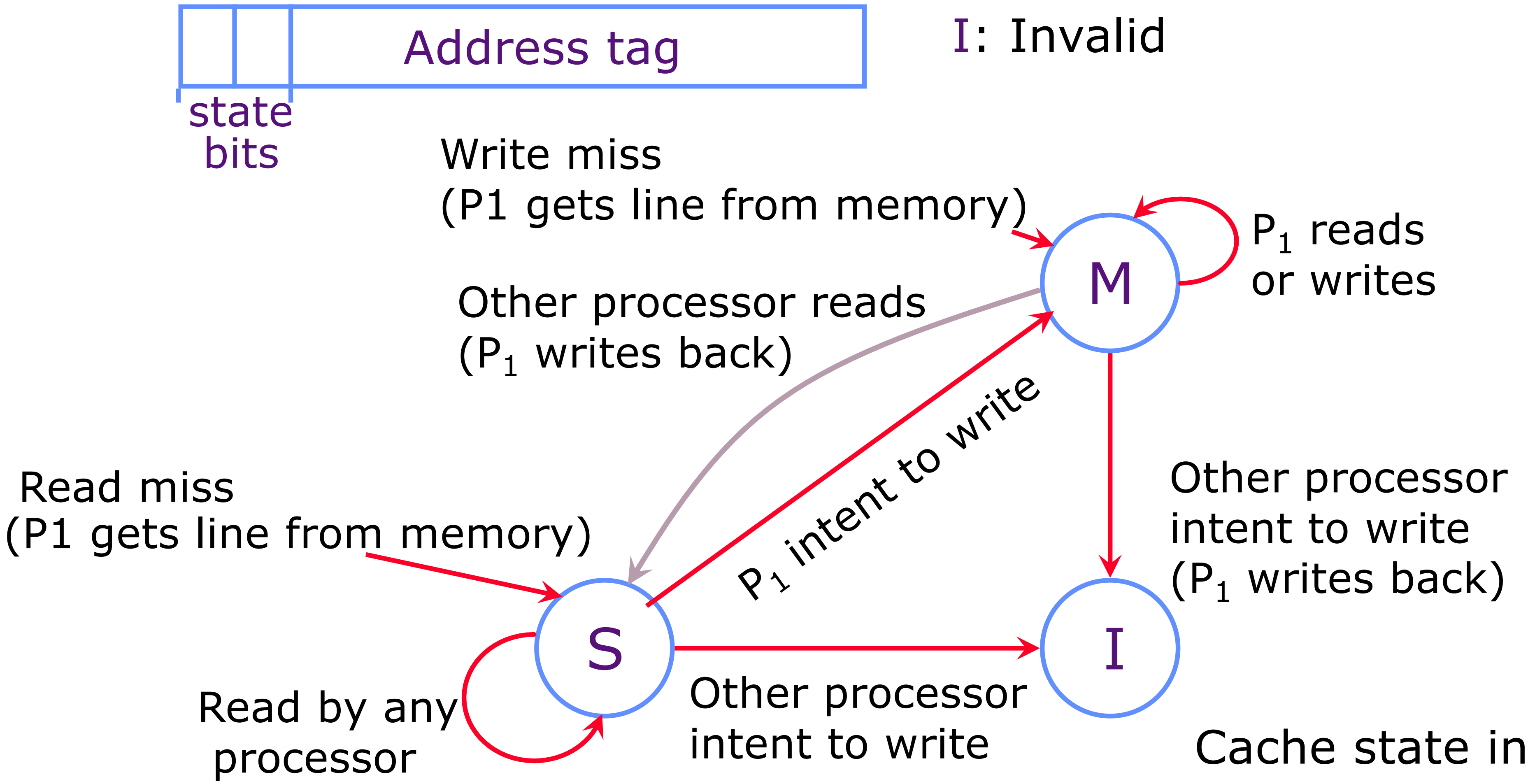
- Read miss:
  - if a dirty copy is found in some cache, a write-back is performed before the memory is read

Cache State-Transition Diagram

The MSI protocol

Each cache line has state bits

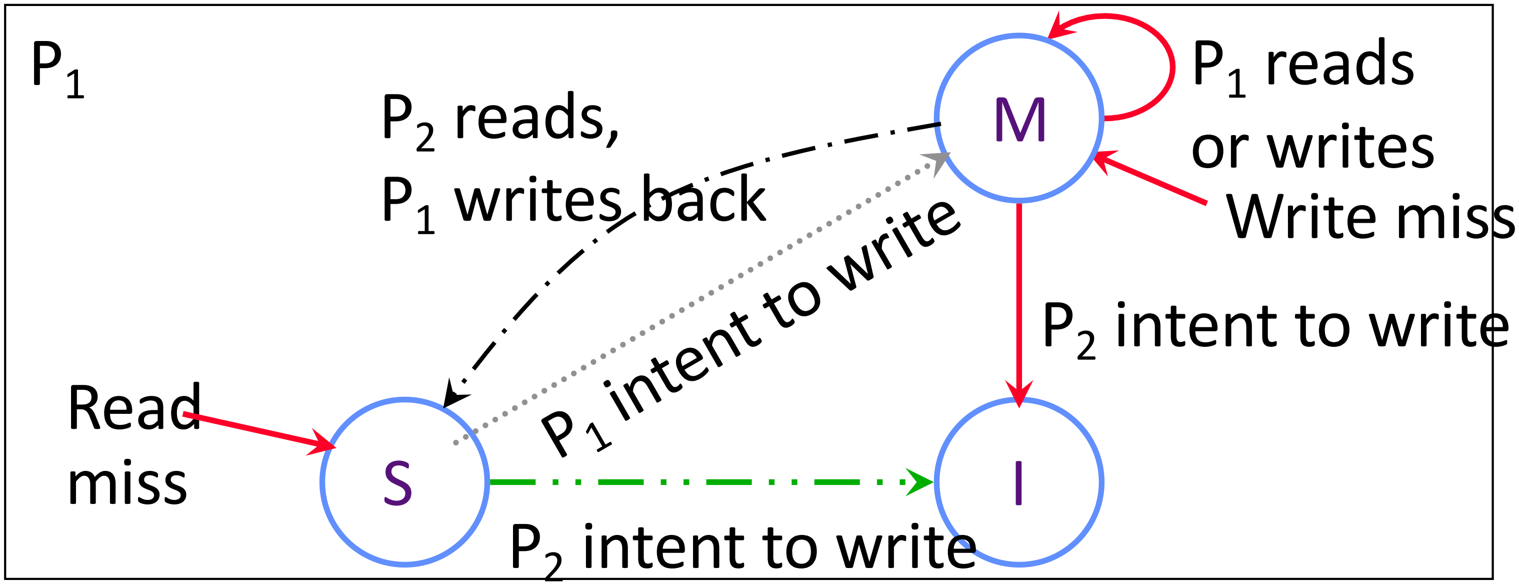
M: Modified
S: Shared
I: Invalid

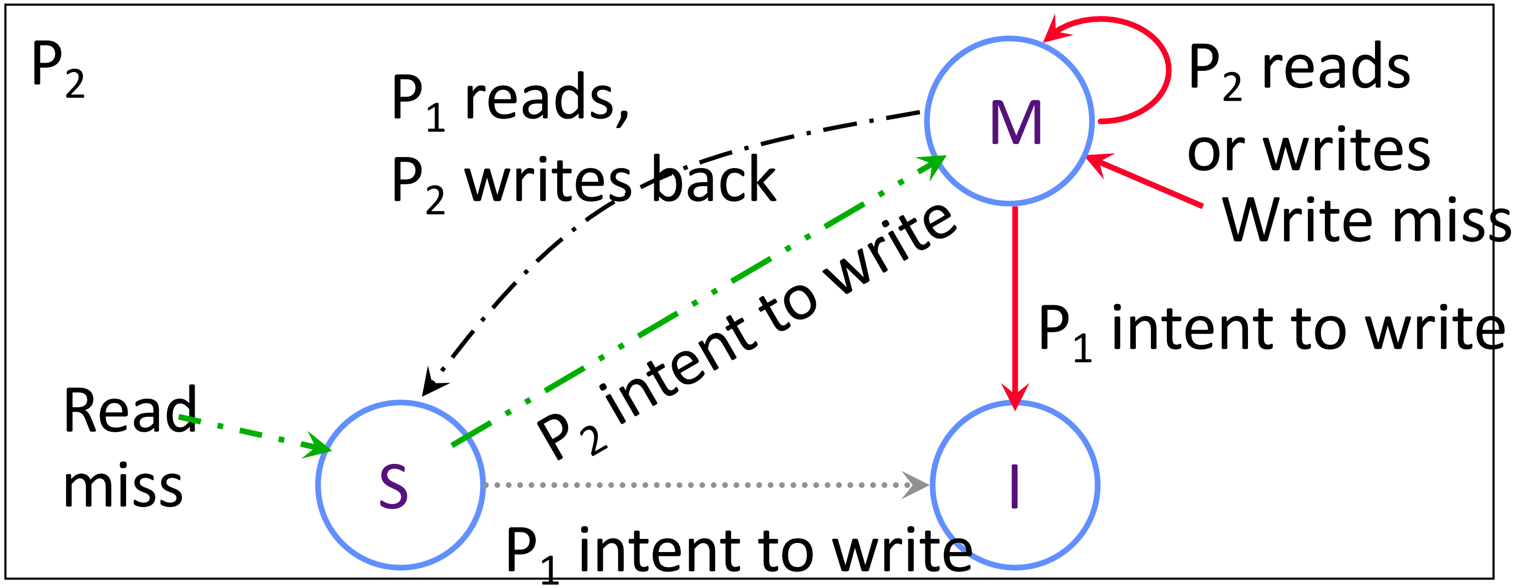

Address tag
state bits
Write miss (P1 gets line from memory)
M
P$_1$ reads or writes
Other processor reads (P$_1$ writes back)
P$_1$ intent to write
Other processor intent to write (P$_1$ writes back)
Read miss (P1 gets line from memory)
S
I
Read by any processor
Other processor intent to write

Cache state in processor P$_1$

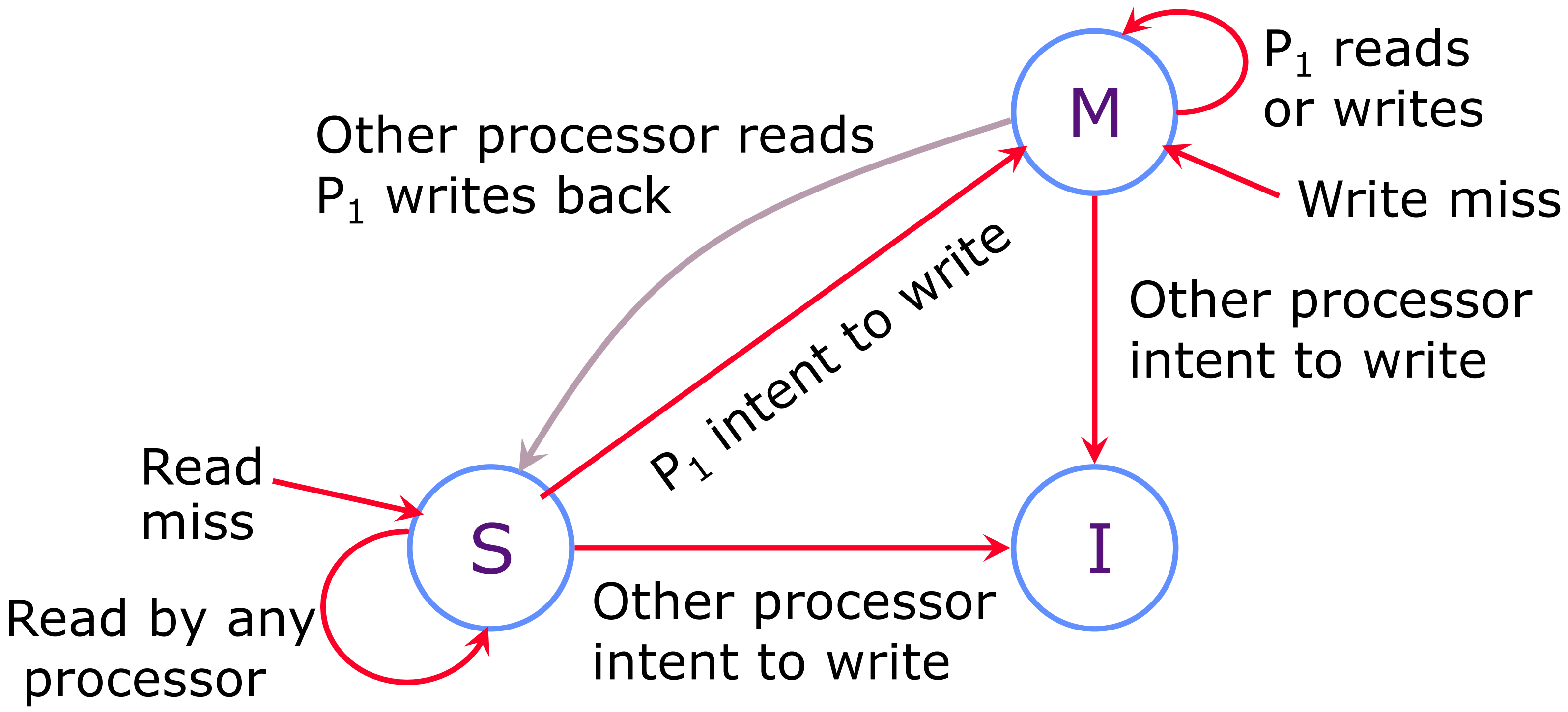
Two-Processor Example
(Reading and writing the same cache line)

P_1 reads
P_1 writes
P_2 reads
P_2 writes
P_1 reads
P_1 writes
P_2 writes
P_1 writes





Observation

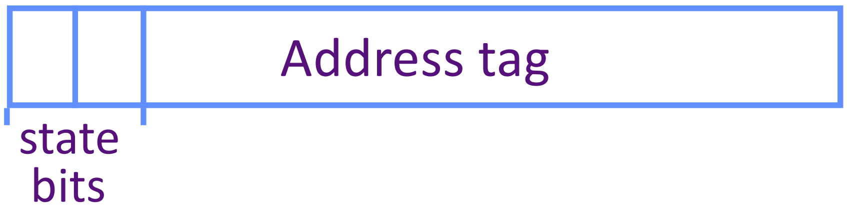

M
S
I
P1 reads or writes
Write miss
Other processor reads P1 writes back
P1 intent to write
Other processor intent to write
Read miss
Read by any processor
Other processor intent to write

- If a line is in the M state then no other cache can have a copy of the line!
- Memory stays coherent, multiple differing copies cannot exist

MESI: An Enhanced MSI protocol

increased performance for private data

Each cache line has a tag

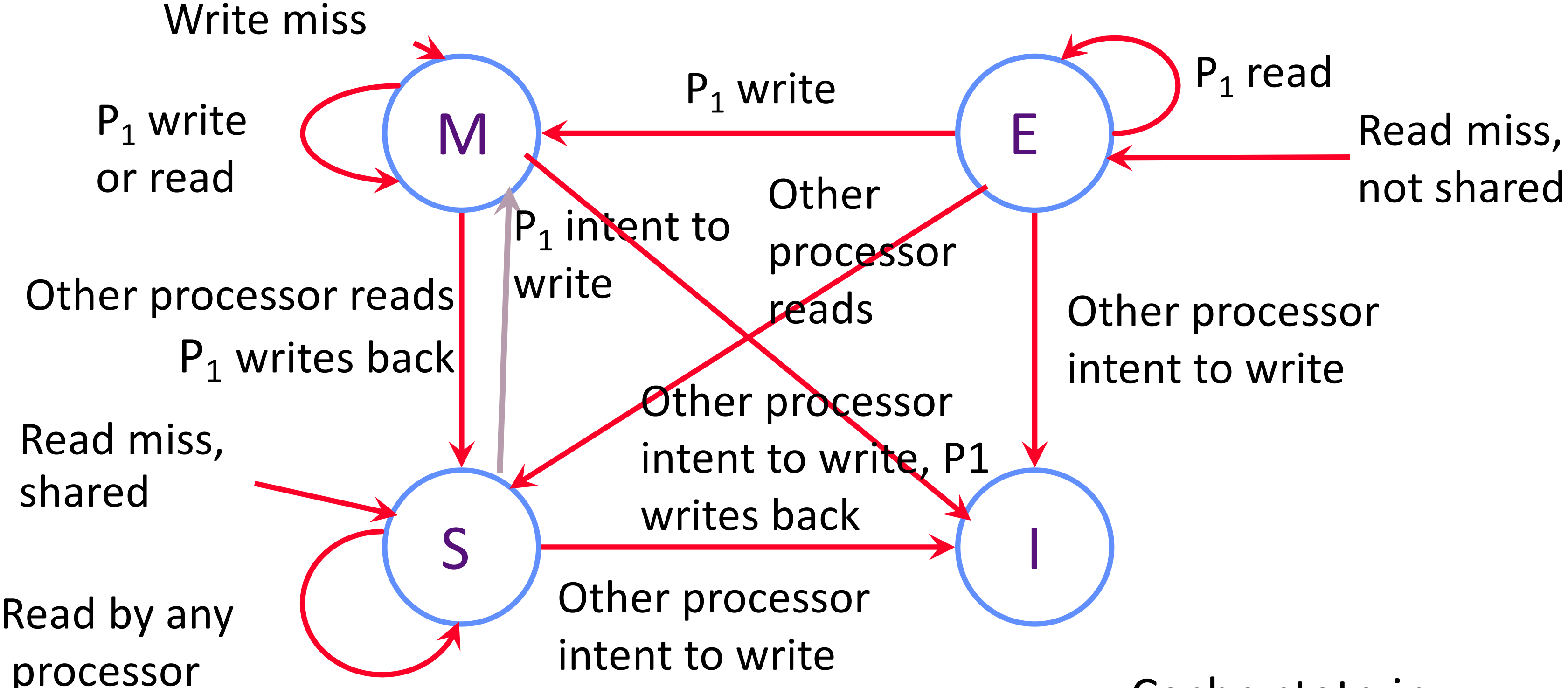


M: Modified Exclusive

E: Exclusive but unmodified

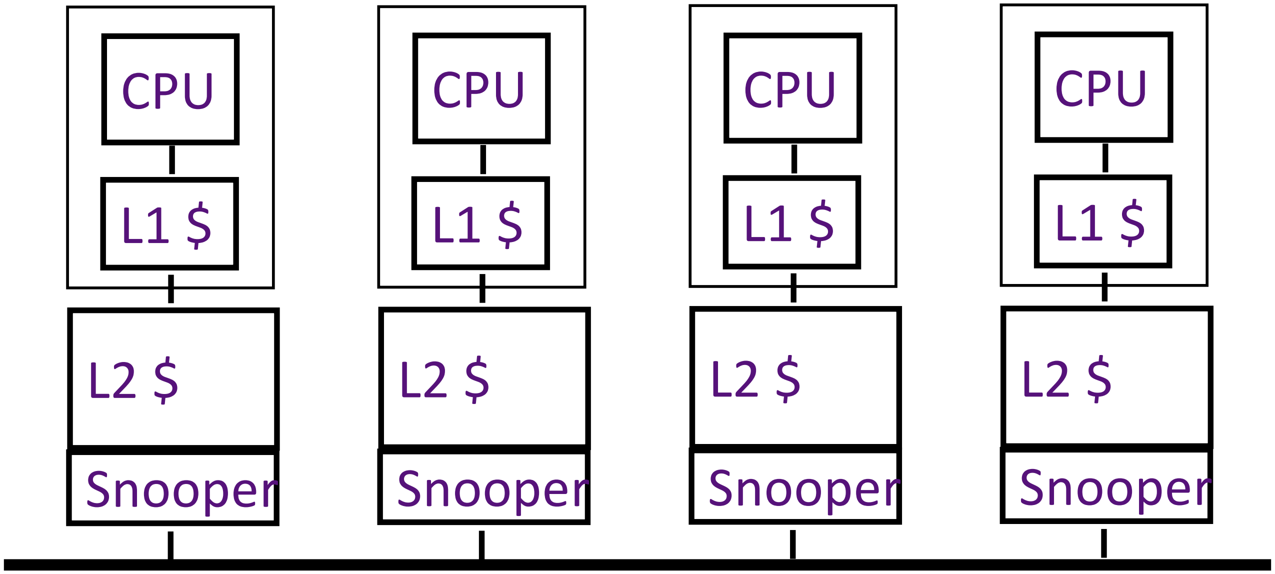
S: Shared

I: Invalid



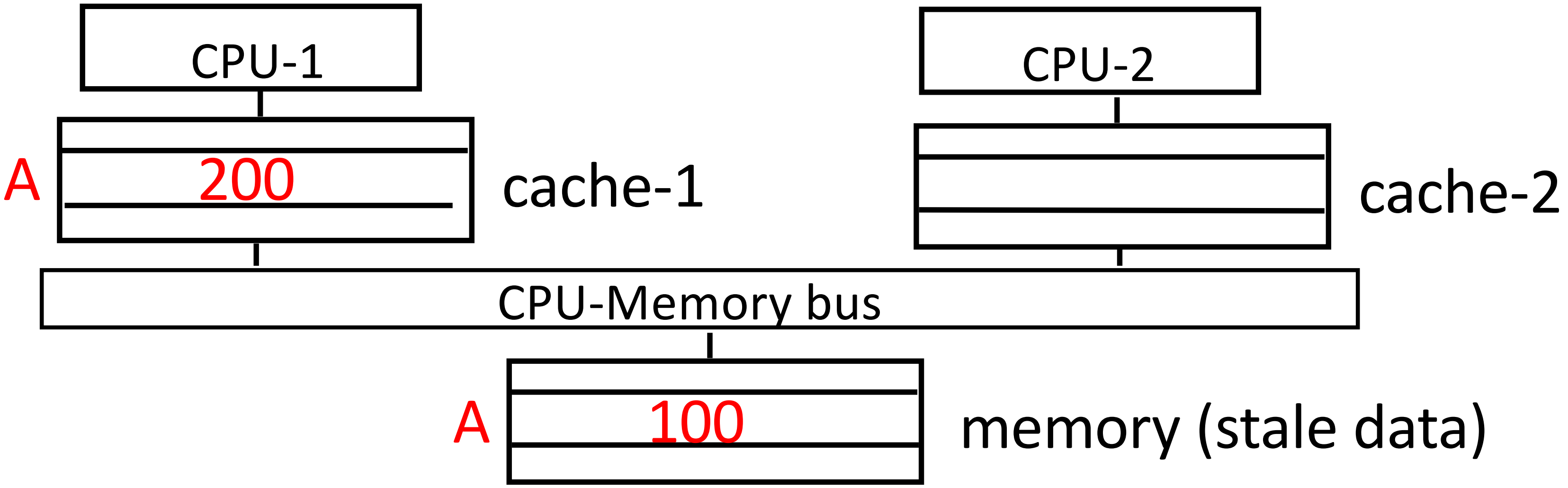
Cache state in processor P_1

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
  - Miss in L2 ⇒ Not present in L1
  - Only if invalidation hits in L2 ⇒ probe and invalidate in L1
- Snooping on L2 does not affect CPU-L1 bandwidth

Intervention



When a read-miss for A occurs in cache-2,

Cache-2 initiates a read request for A 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

state	line addr	data0	data1	…	dataN

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 $i \neq k$ but 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
    ⇒ miss would not occur if line size were 1 word

Example: True v. False Sharing v. Hit?

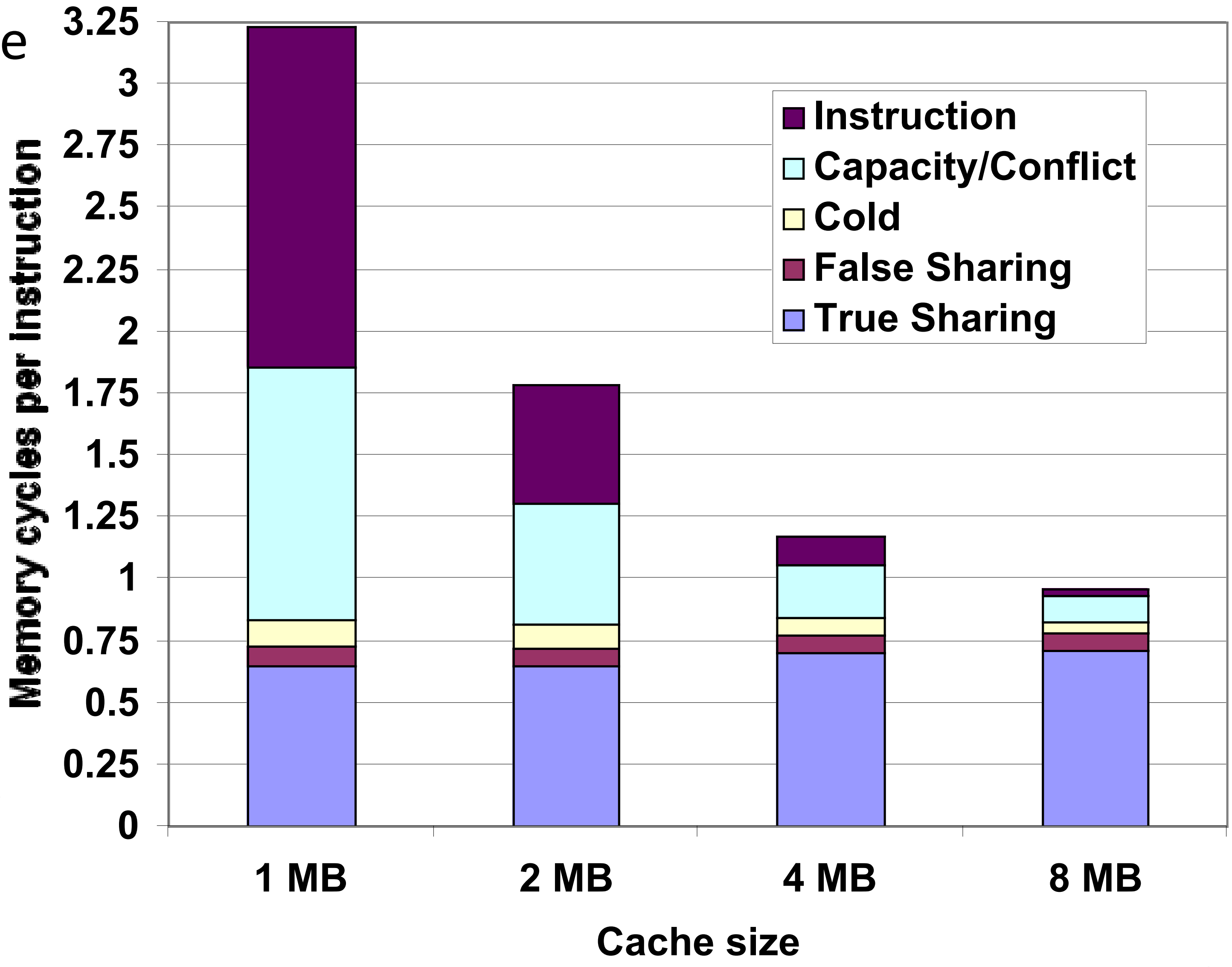
- MSI protocol
- Assume x1 and x2 in same cache line.
  P1 and P2 both read x1 and x2 before.

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; x2 invalid in P1**

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

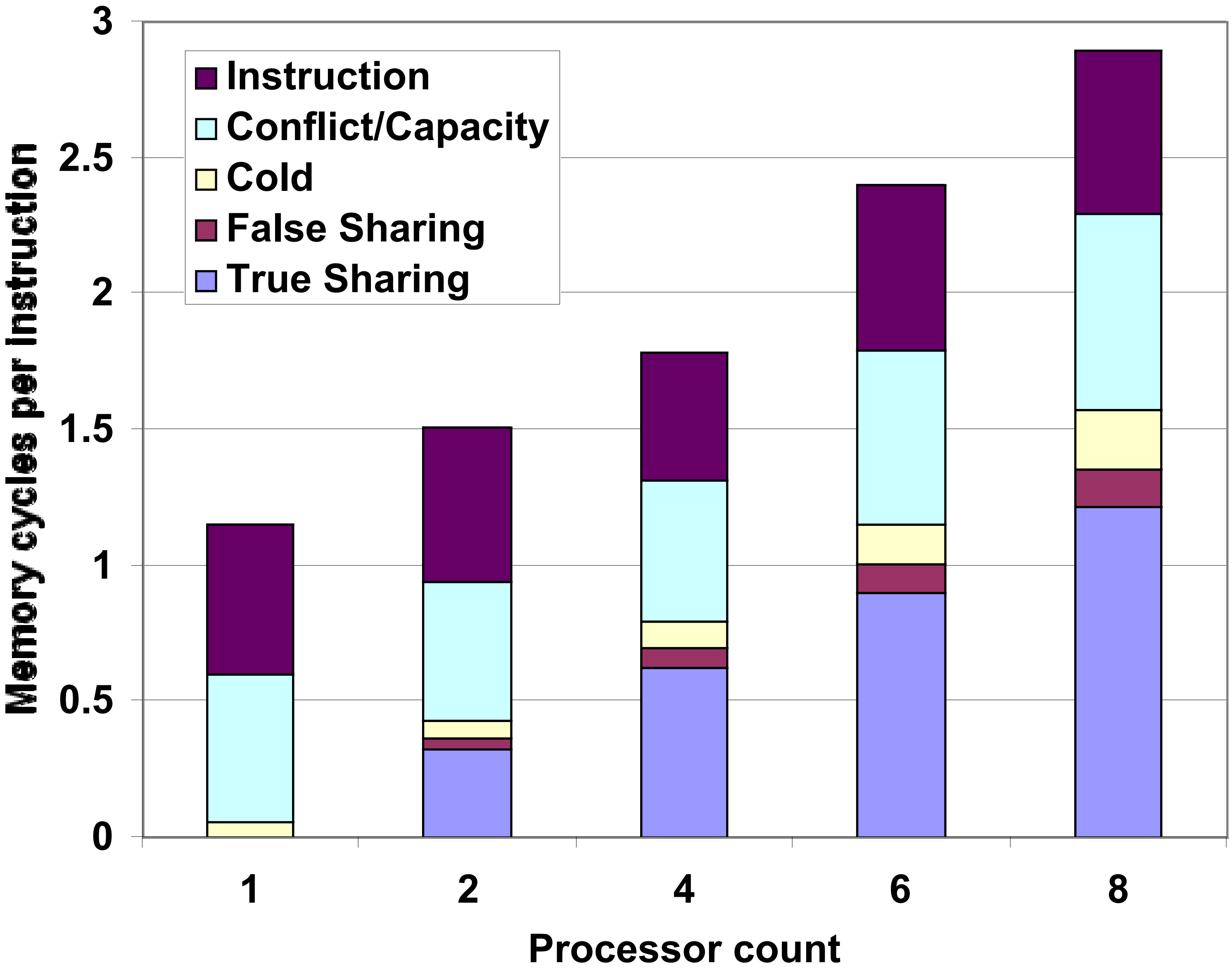
- Uniprocessor cache misses improve with cache size increase (Instruction, Capacity/Conflict, Compulsory)

- True sharing and false sharing unchanged going from 1 MiB to 8 MiB (L3 cache)



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

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



CS152 Administrivia

- PS4 and Lab 3 due today
- Lab 4 due Monday April 19
- Midterm 2 Wednesday April 14
  - covers lectures 10-17, plus associated problem sets, labs, and readings
  - updated Zoom proctoring procedure

CS252 Administrivia

- This week’s readings Cray-1 and VLIW machines
- Thursday April 15th Project Checkpoint
  - Schedule 10-minute individual group zoom calls during discussion period

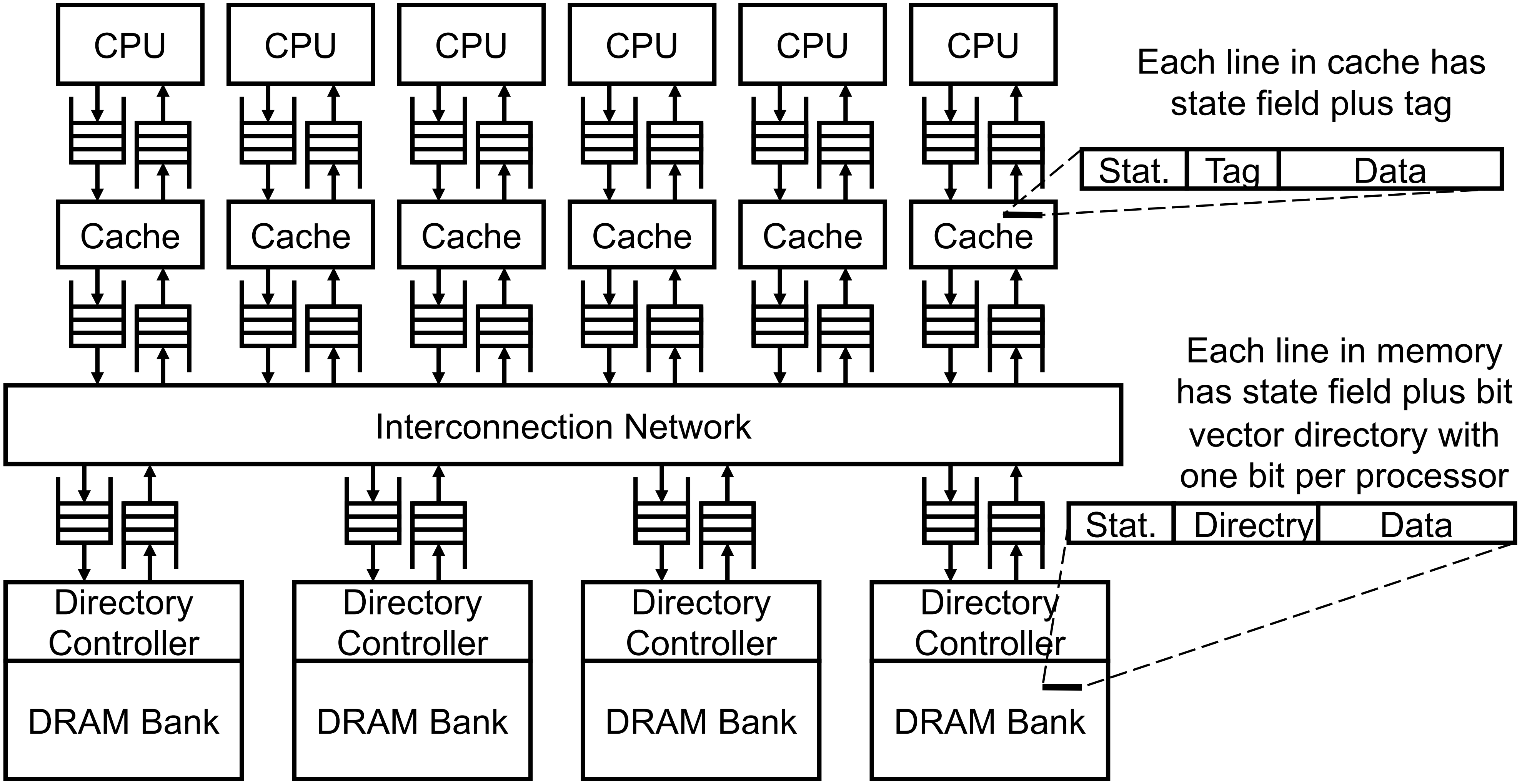
Scaling Snoopy/Broadcast Coherence

- 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

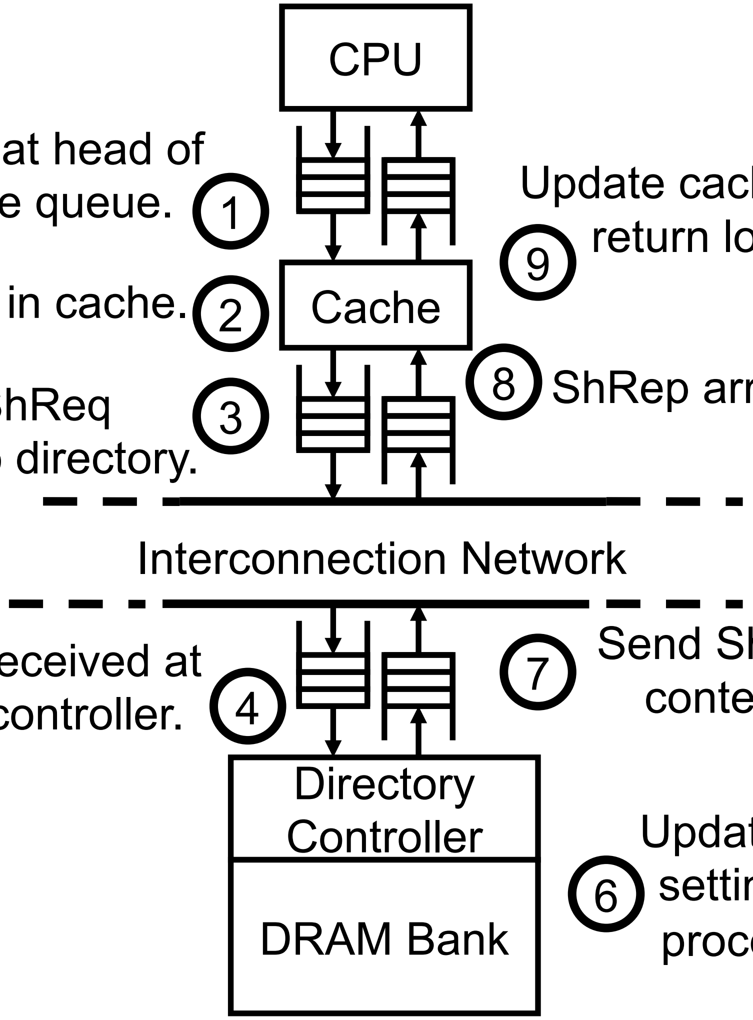
Cache States

- 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



1 Load request at head of CPU->Cache queue.

2 Load misses in cache.

3 Send ShReq message to directory.

4 Message received at directory controller.

5 Access state and directory for line. Line’s state is R, with zero or more sharers.

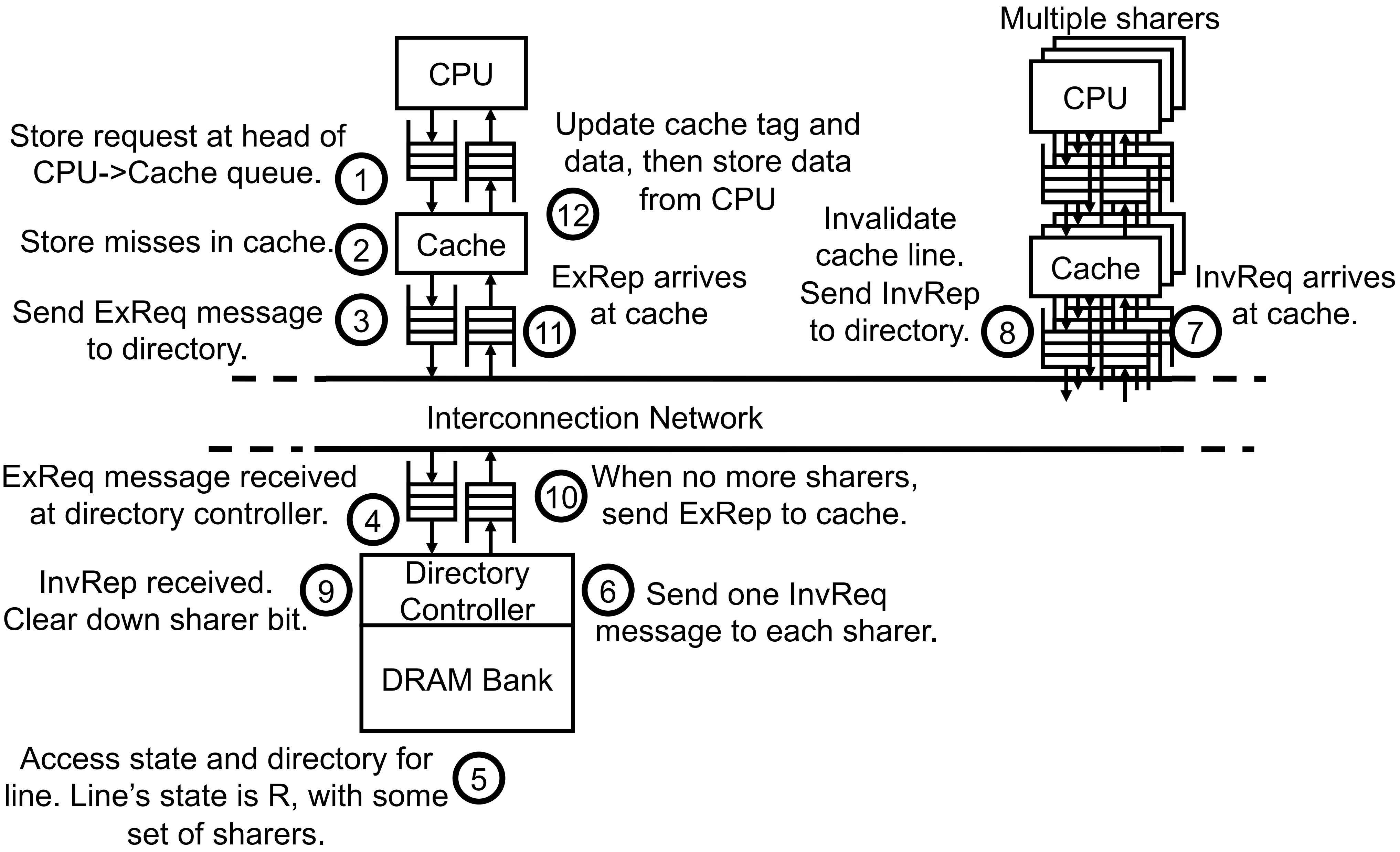
6 Update directory by setting bit for new processor sharer.

7 Send ShRep message with contents of cache line.

8 ShRep arrives at cache.

9 Update cache tag and data and return load data to CPU.

Write miss, to read shared line



1 Store request at head of CPU->Cache queue.

2 Store misses in cache.

3 Send ExReq message to directory.

4 ExReq message received at directory controller.

5 Access state and directory for line. Line's state is R, with some set of sharers.

6 Send one InvReq message to each sharer.

7 InvReq arrives at cache.

8 Invalidate cache line. Send InvRep to directory.

9 InvRep received. Clear down sharer bit.

10 When no more sharers, send ExRep to cache.

11 ExRep arrives at cache

12 Update cache tag and data, then store data from CPU

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!

Acknowledgements

- This course is partly inspired by previous MIT 6.823 and Berkeley CS252 computer architecture courses created by my collaborators and colleagues:
  - Arvind (MIT)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)