# CS 152 Computer Architecture and Engineering

# Lecture 19: Directory-Based Cache Protocols

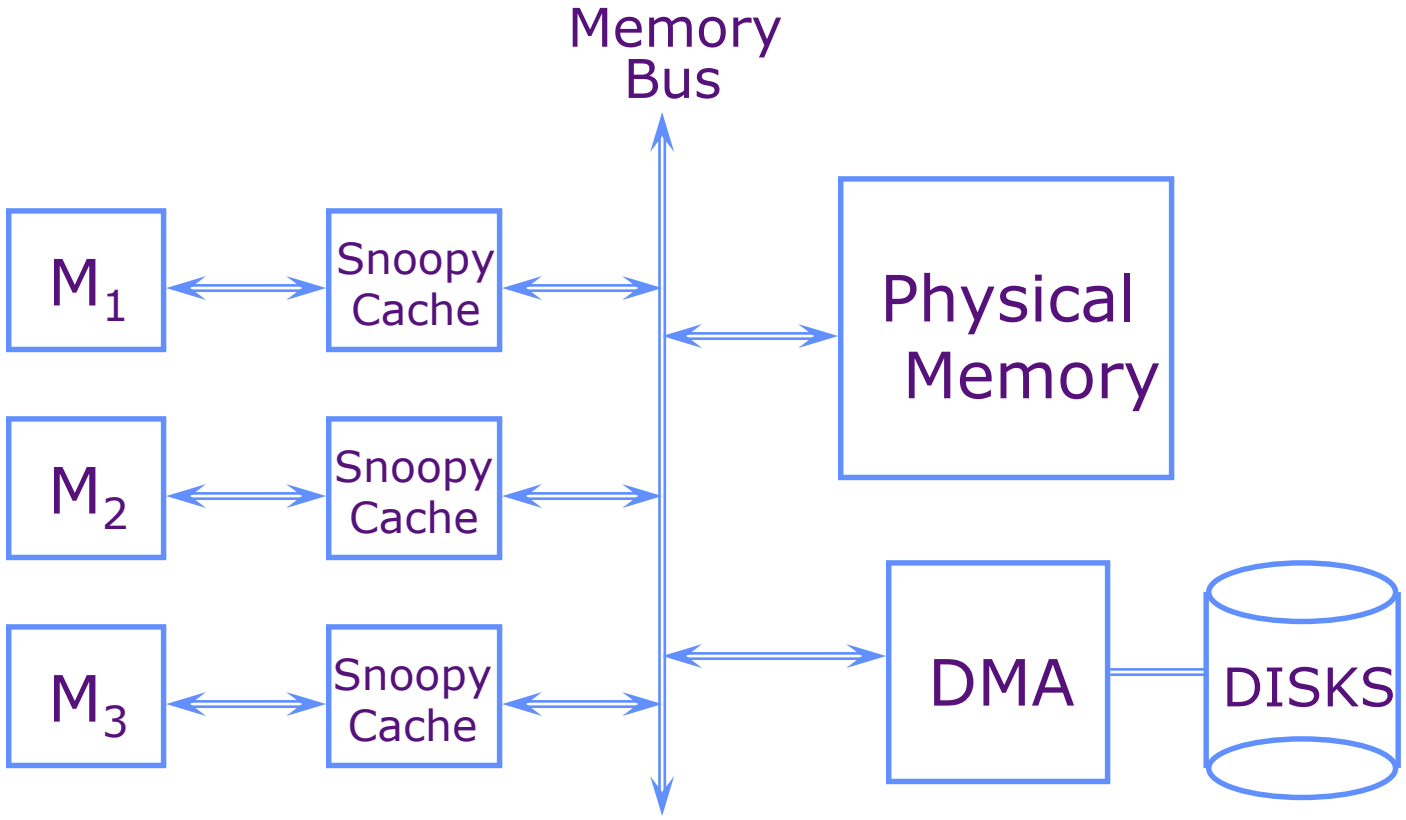
Dr. George Michelogiannakis
EECS, University of California at Berkeley
CRD, Lawrence Berkeley National Laboratory

**http://inst.eecs.berkeley.edu/~cs152**

# Administrivia

- PS 5 due on Wednesday
- Wednesday lecture on data centers
- Quiz 5 on Wednesday next week
- Please show up on Monday April 21st (last lecture)
  - Neuromorphic, quantum
  - Parting thoughts that have nothing to do with architecture
  - Class evaluation

# Recap: Snoopy Cache Protocols

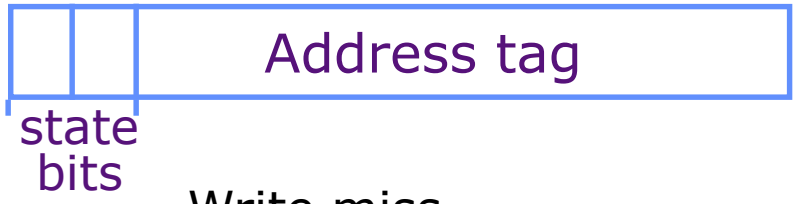



Use snoopy mechanism to keep all processors' view of memory coherent

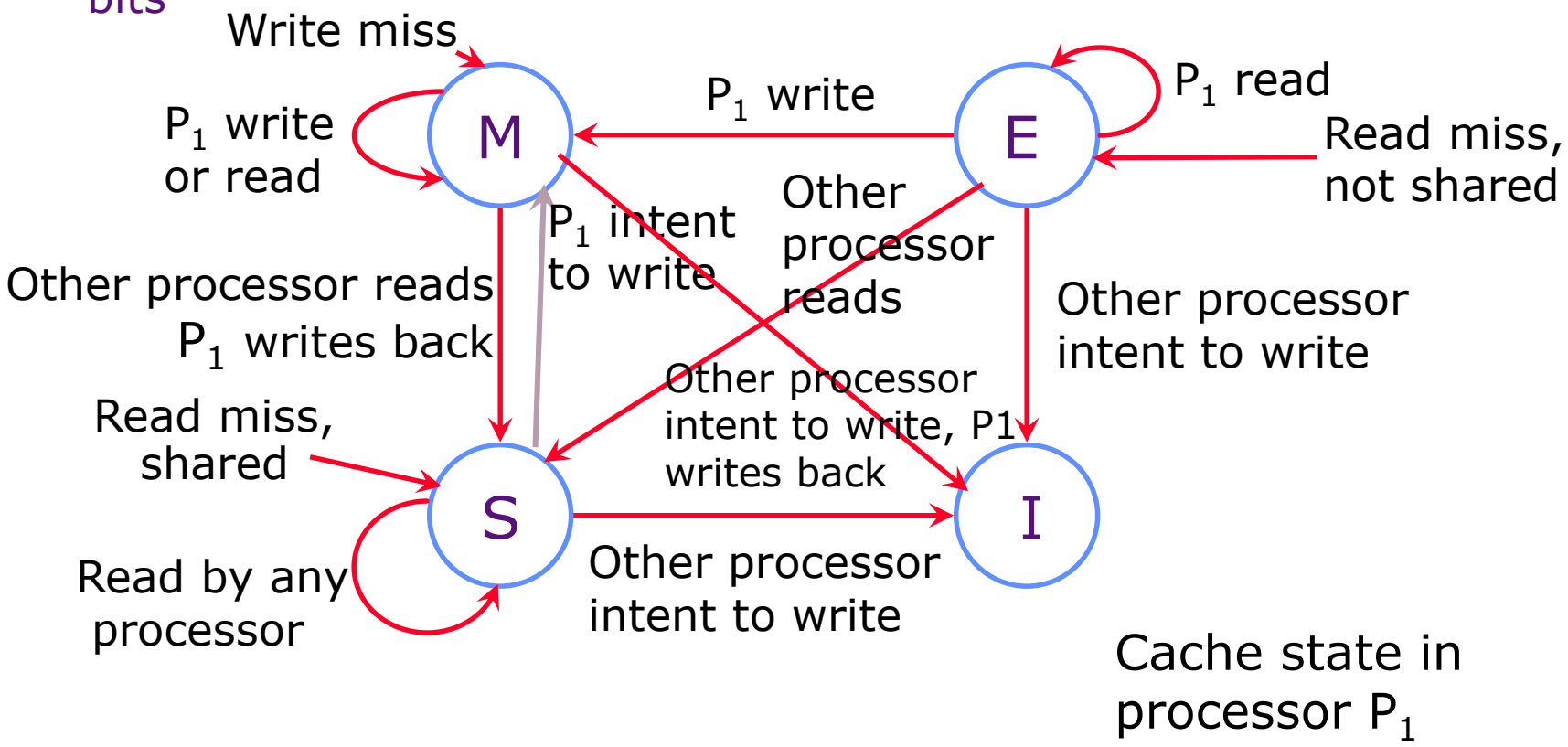
# Recap: MESI: An Enhanced MSI protocol

## increased performance for private data

*Each* cache line has a tag




M: Modified Exclusive
E: Exclusive but unmodified
S: Shared
I: Invalid



# Performance of Symmetric Shared-Memory Multiprocessors

Cache performance is combination of:

1. Uniprocessor cache miss traffic
2. Miss traffic caused by communication
   - Results in invalidations and subsequent cache misses

- *Coherence misses*
  - Sometimes called a *Communication* miss
  - A cache miss which is a result of a remote core
    - Read miss: remote core wrote
    - Write miss: remote core wrote or read
  - 4th *C* of cache misses along with *C*ompulsory, *C*apacity, & *C*onflict.

# Coherence Misses

1. 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
2. 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

| state | line addr | data0 | data1 | ... | dataN |
|---|---|---|---|---|---|

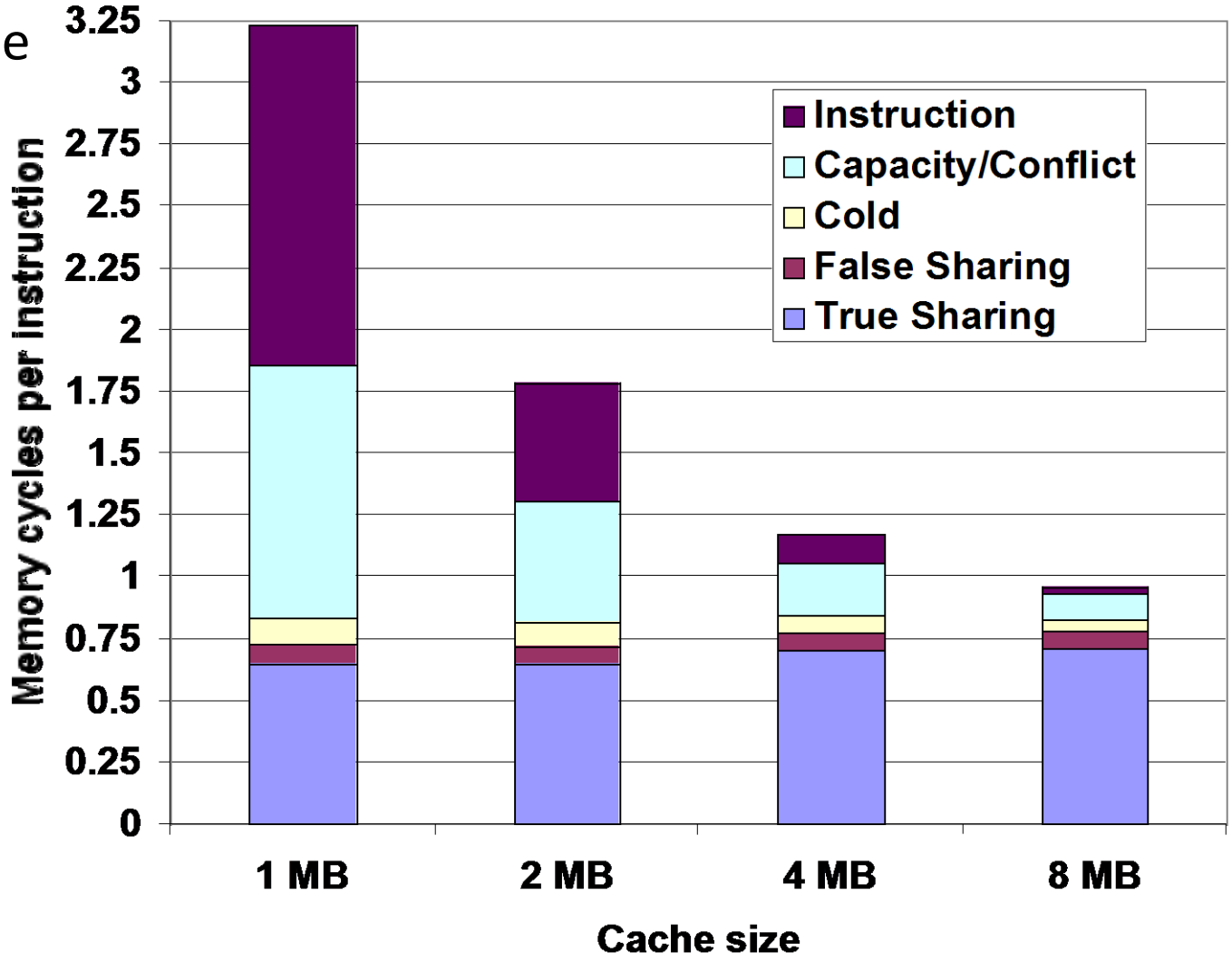
# Example: True v. False Sharing v. Hit?

- 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 |

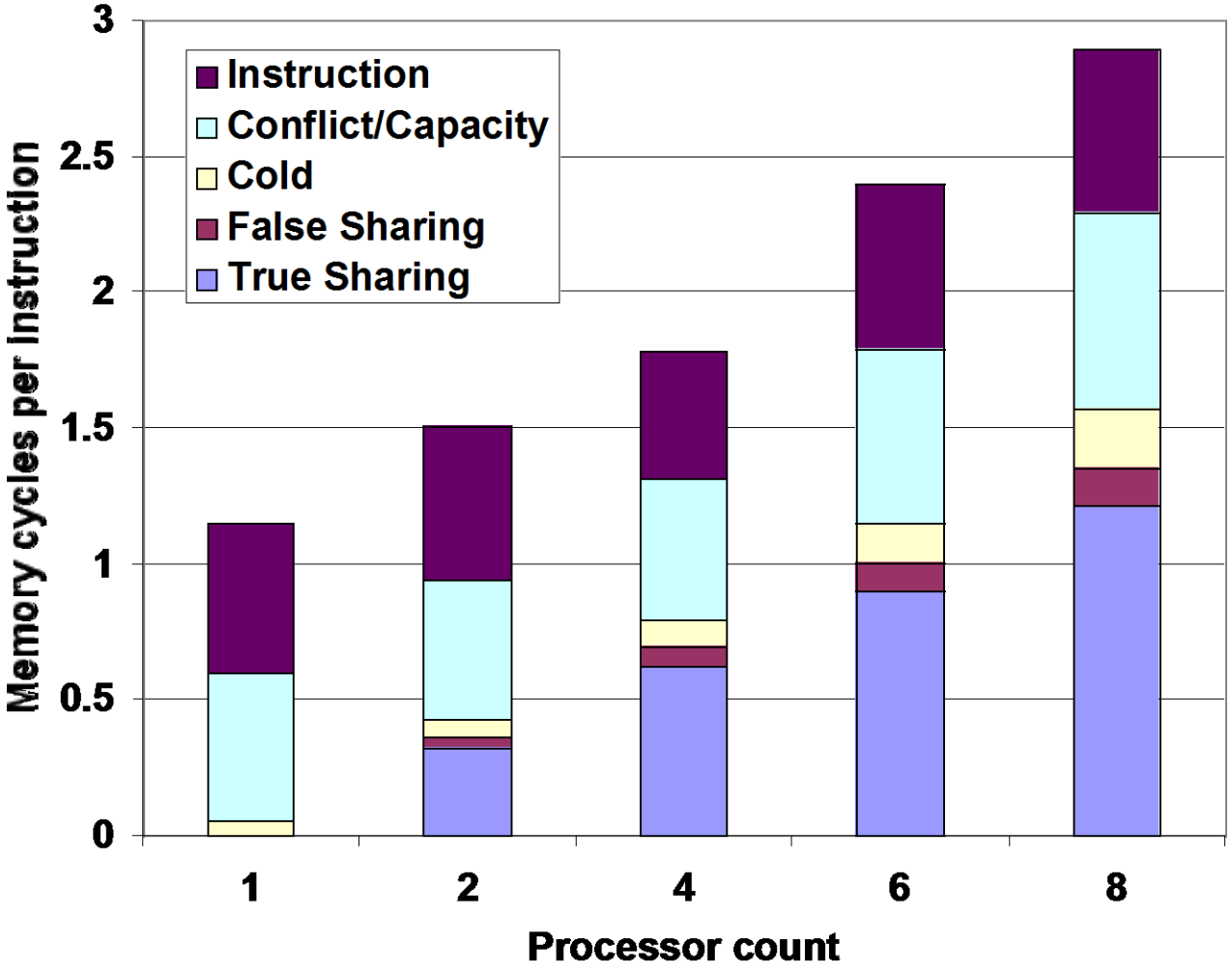
# MP Performance 4 Processor Commercial Workload: OLTP, Decision Support (Database), Search Engine

- 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)

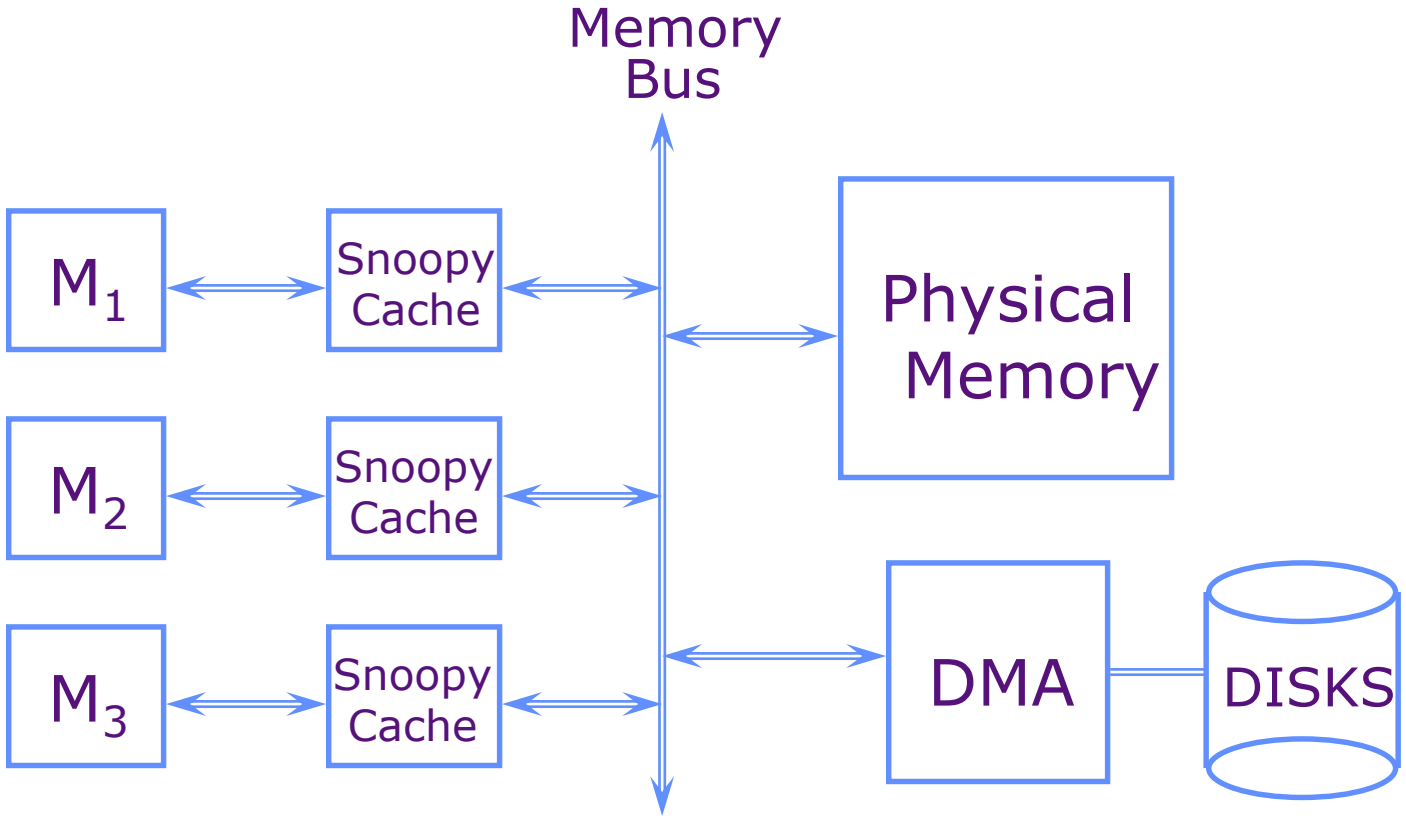


# MP Performance 2MB Cache Commercial Workload: OLTP, Decision Support (Database), Search Engine

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

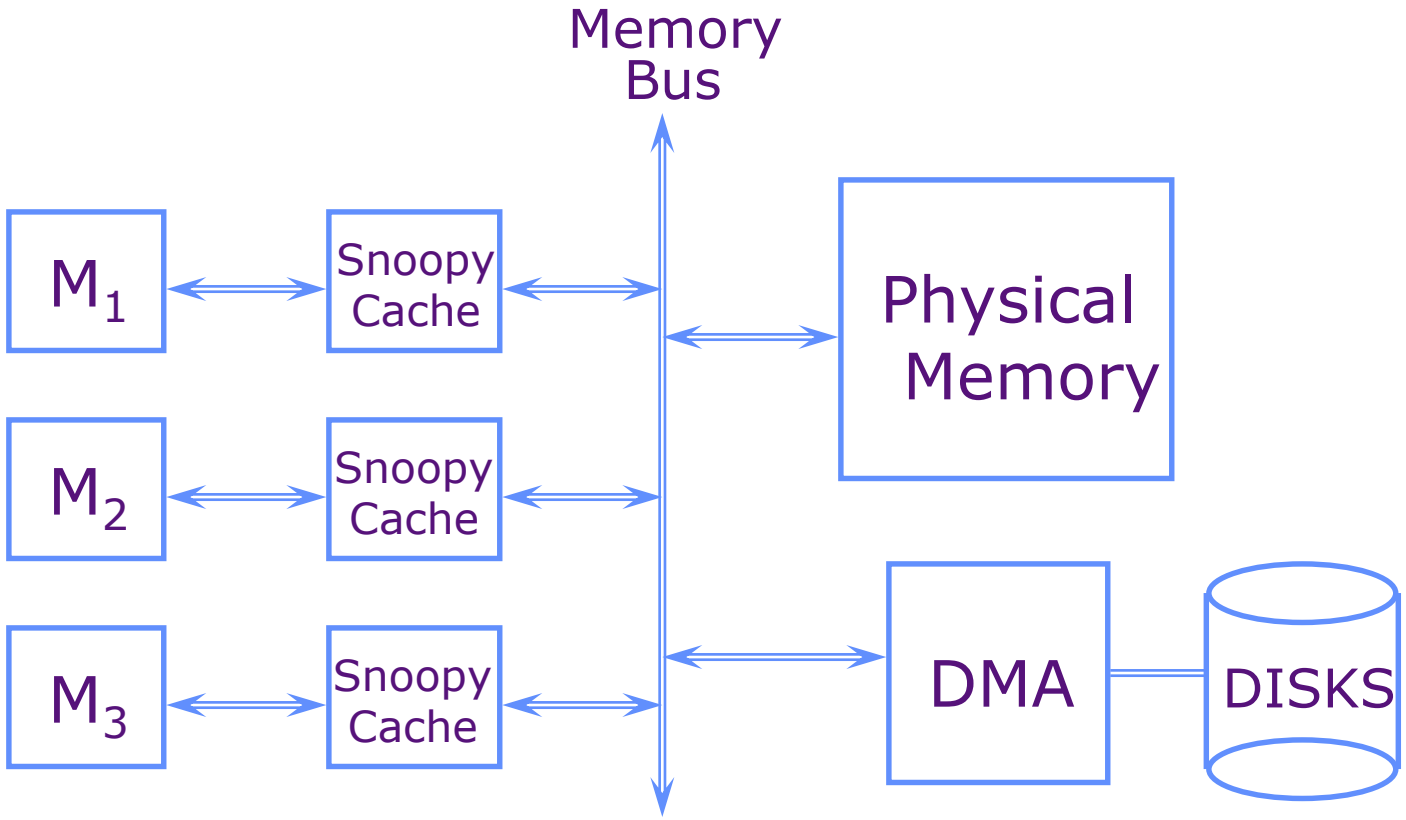


# What if We Had 128 Cores?



# Bus Is a Synchronization Point

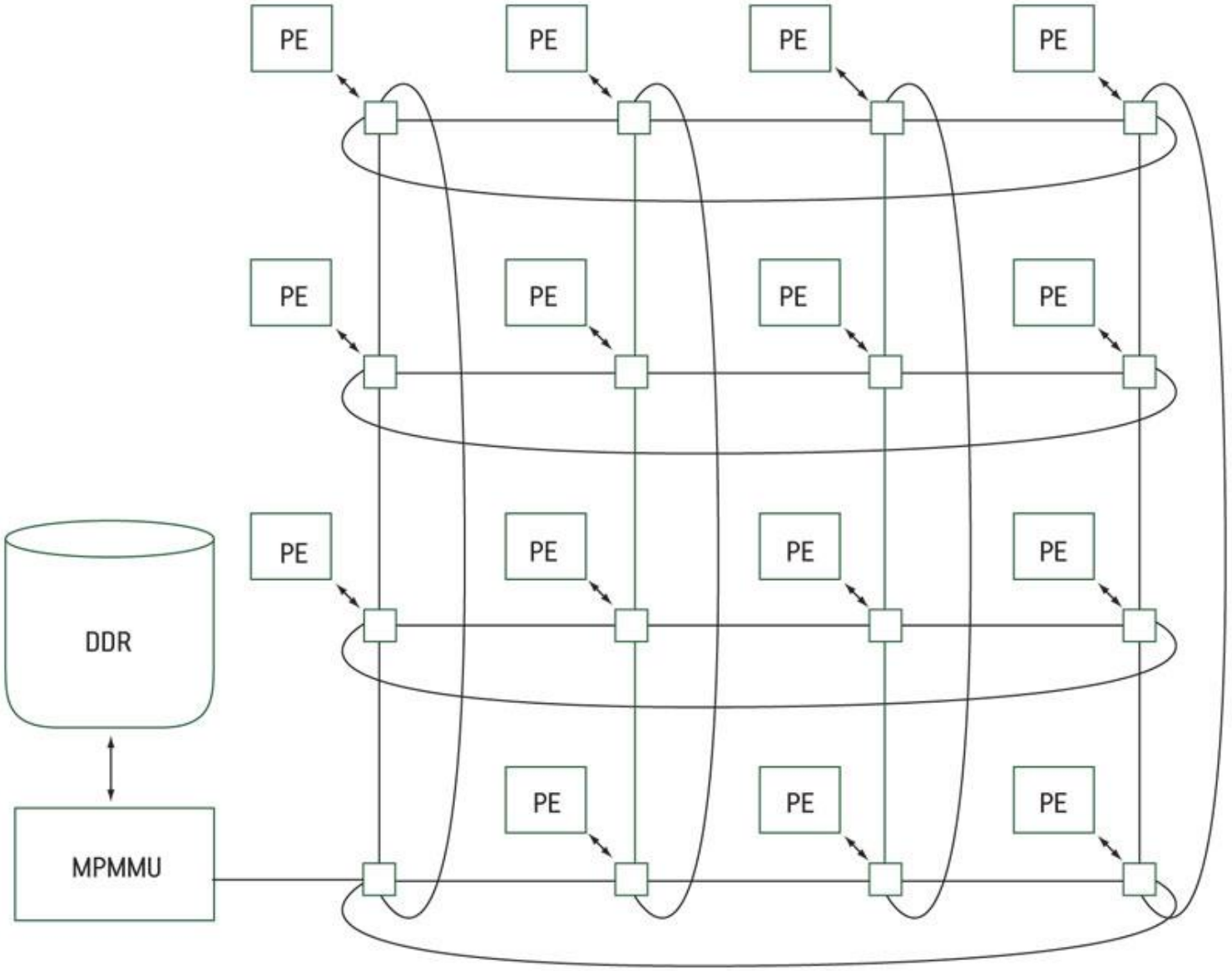
- So far, any message that enters the bus reaches all cores and gets replies before another message can enter the bus (instantaneous actions)
  - Therefore, the bus *forces ordering*



# Scaling Snoopy/Broadcast Coherence

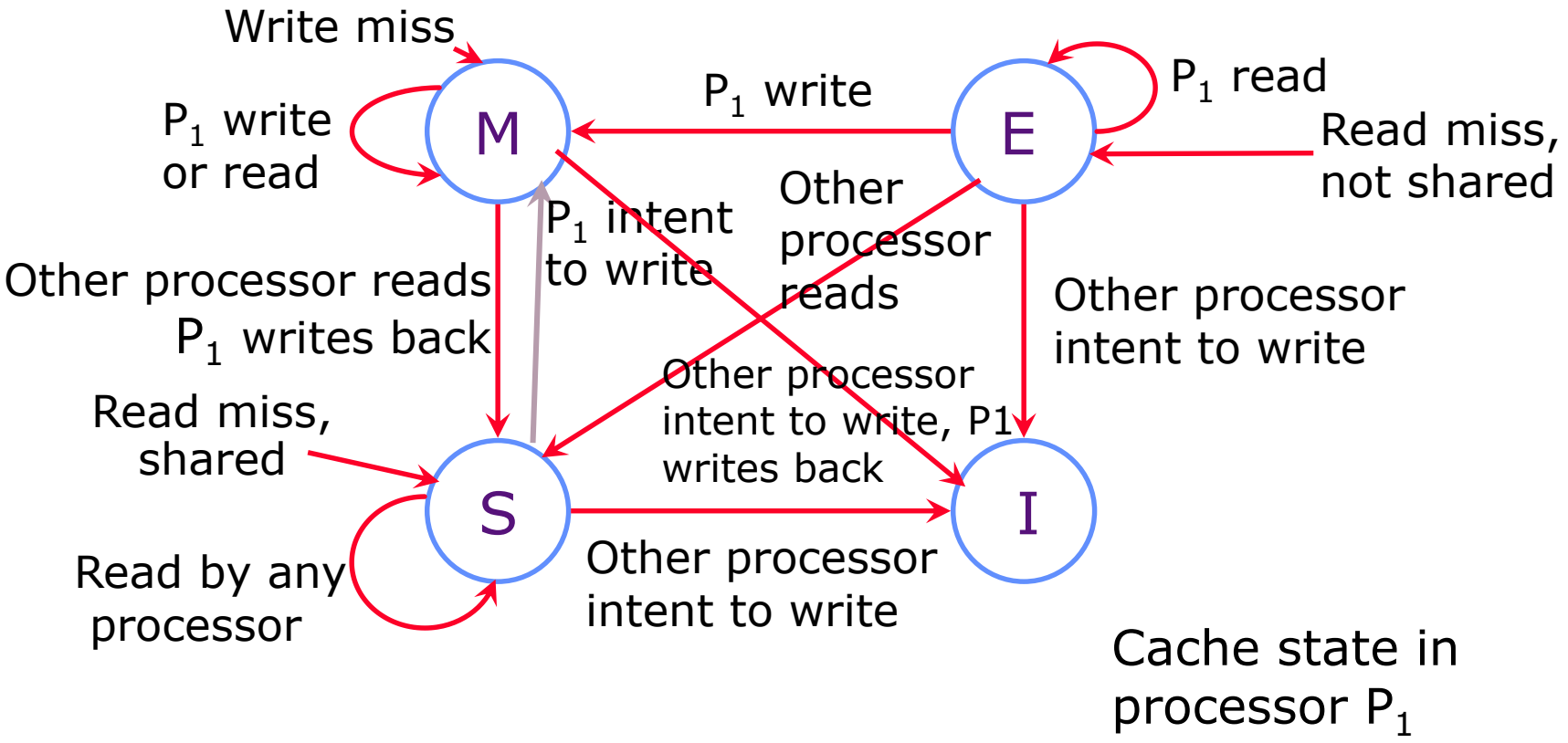
- When any processor gets a miss, must probe every other cache
- Scaling up to more processors limited by:
  - Communication bandwidth over bus
  - Snoop bandwidth into tags
- Can improve bandwidth by using multiple interleaved buses with interleaved tag banks
  - E.g, two bits of address pick which 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

# This Scales Better



# What if Bus Does not Synchronize?

- What if a cache broadcasts invalidations to transition to modified (M), and before that completes it receives an invalidation from another core's transition to modified?



# What if Bus Does not Synchronize?

- Time view:

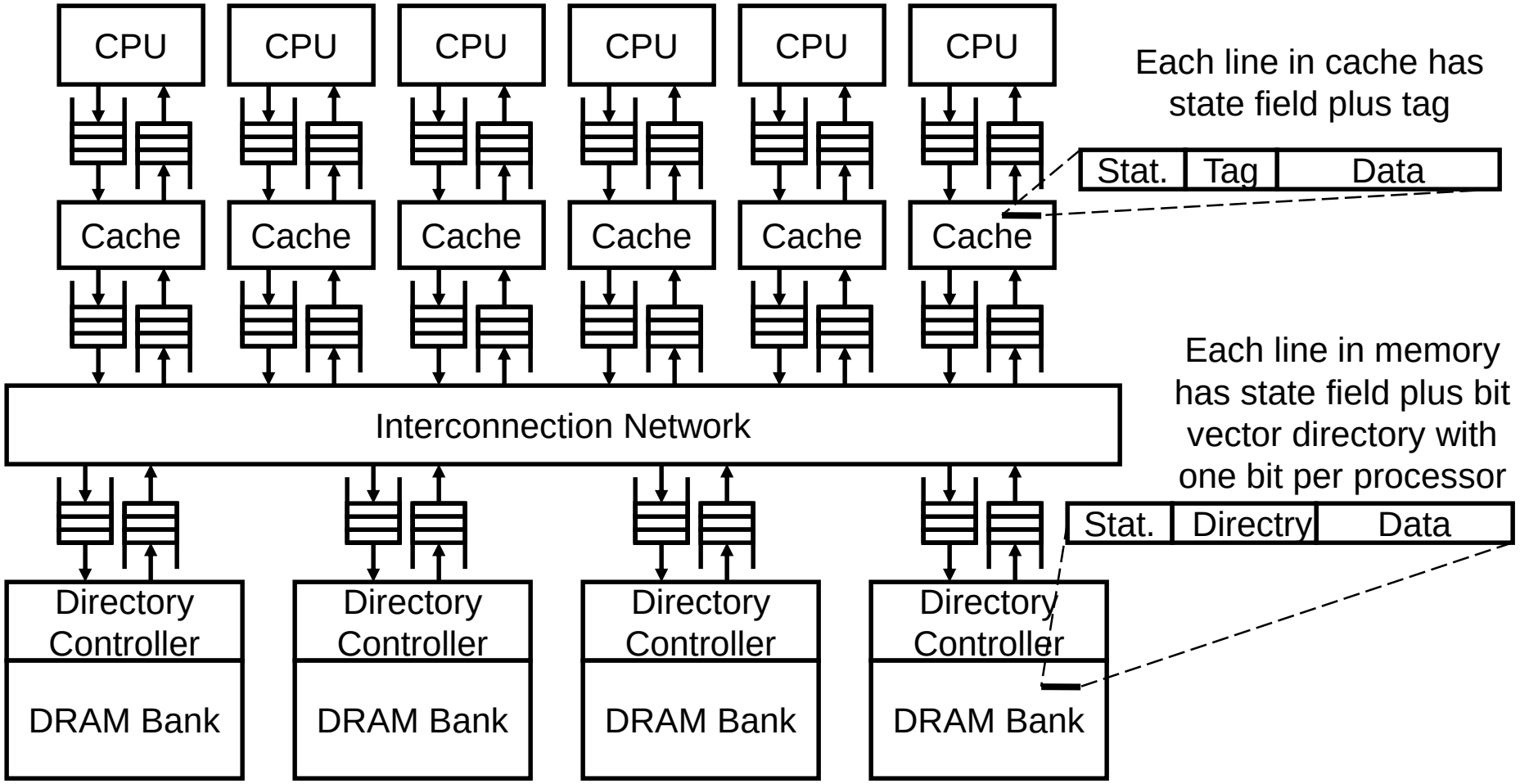
$P_1$ $P_2$

# Scalable Approach: Directories

- 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!

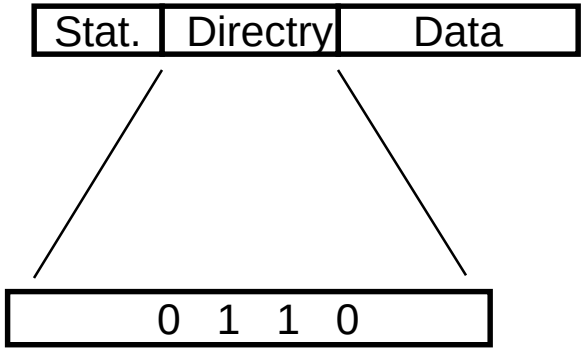
- Every memory line has associated directory information
  - keeps track of copies of cached lines and their states
  - on a miss, find directory entry, look it up, and communicate only with the nodes that have copies if necessary
  - in scalable networks, communication with directory and copies is through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

# Directory Cache Protocol (Lab 5 Handout)




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

# Vector Directory Bit Mask



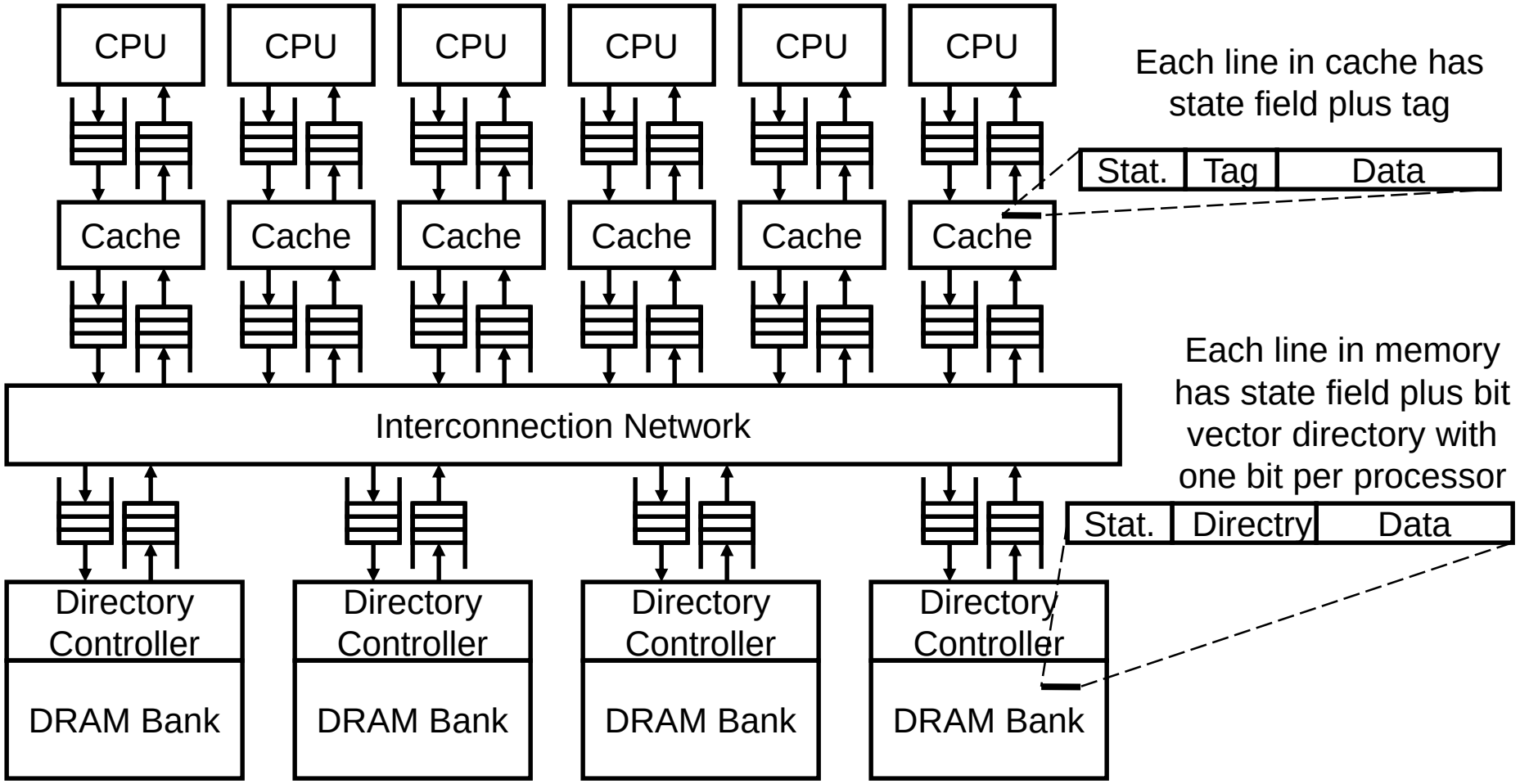

- With four cores, means that cores 2 and 3 have that line in their local cache
- Can also have a list of core IDs
- *With MESI, how do we know what state each cache has the line in*?
  - Think of the case with one sharer

# Cache States

For each cache line, there are 4 possible states (based on MSI):

- 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).

# Network Has No Ordering Guarantees

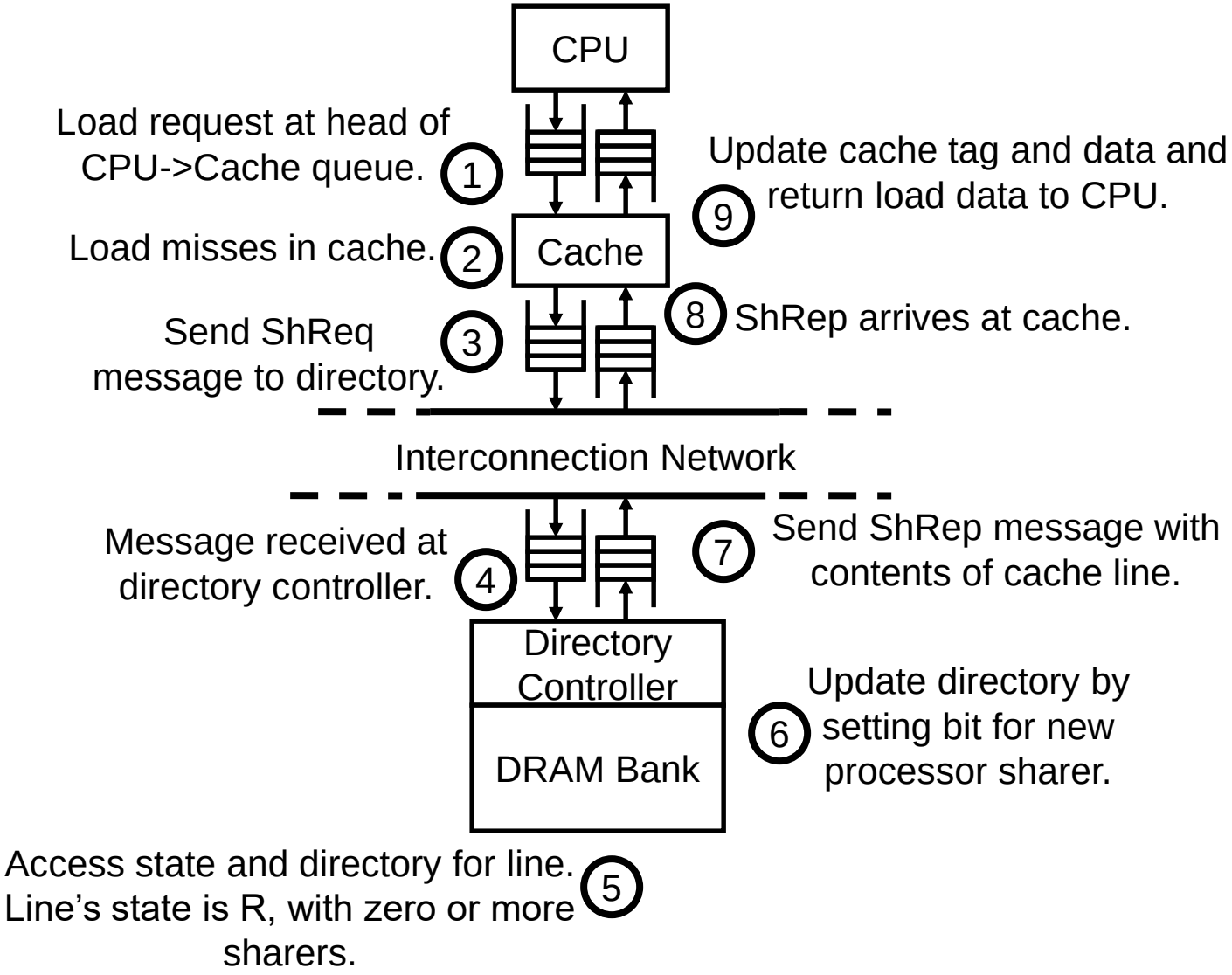


# Home Directory States

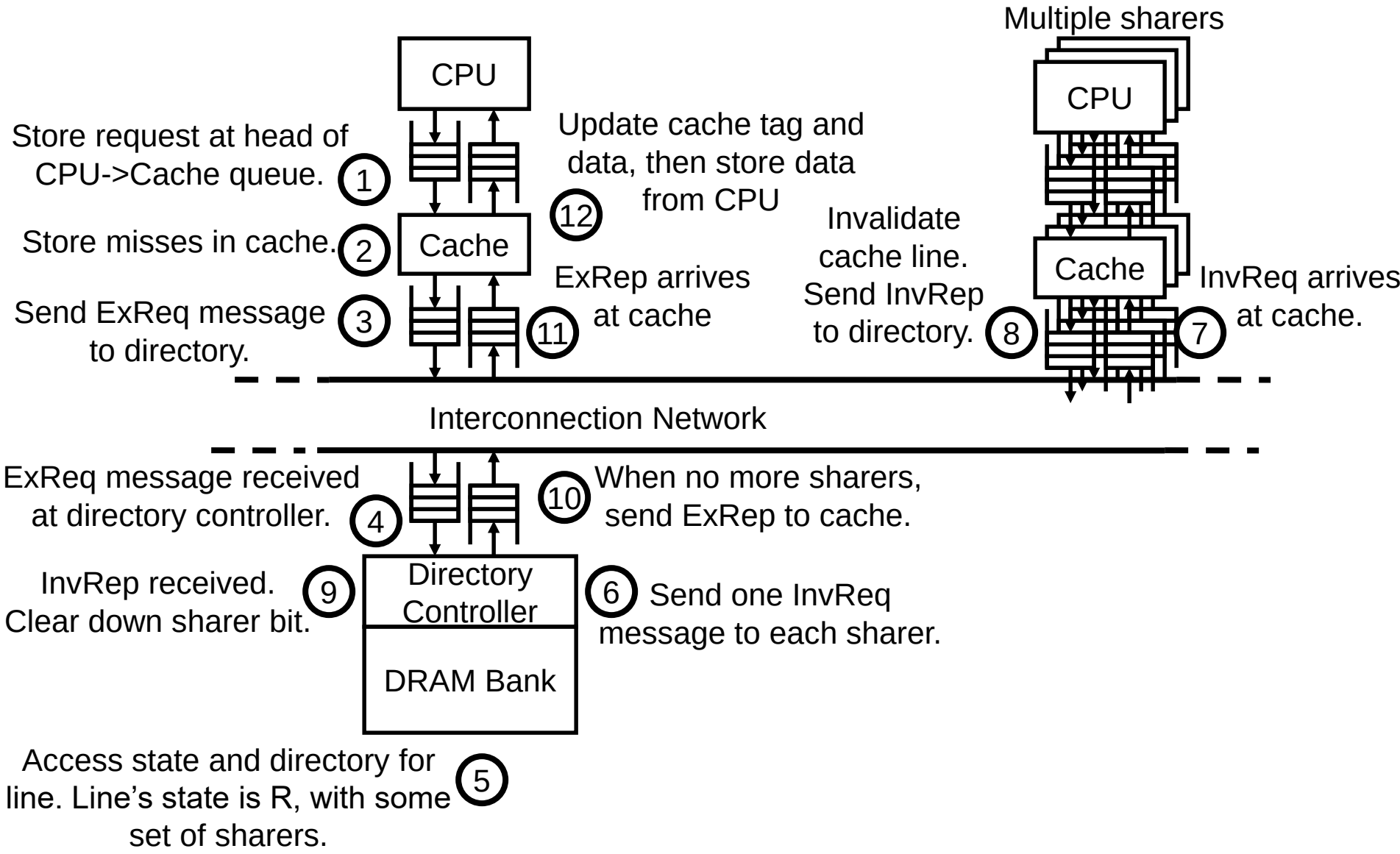
- For each memory line, there are 4 possible states:
  - R(dir): The memory line is shared by the sites specified in dir (dir is a set of sites). The data in memory is valid in this state. If dir is empty (i.e., dir = ε), the memory line is not cached by any site.
  - W(id): The memory line is exclusively 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.

- *Different states in directory than caches*

# Read miss, to uncached or shared line



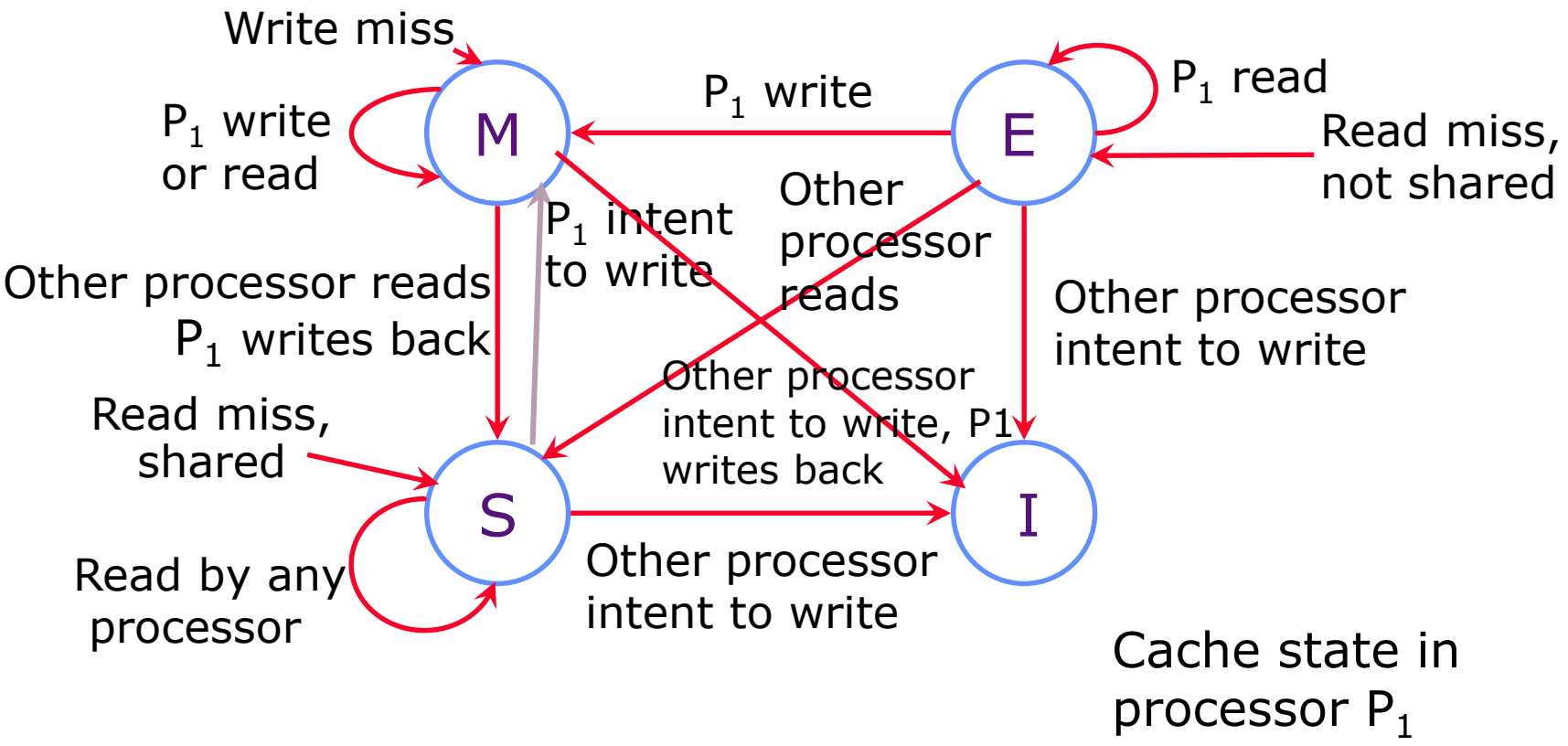
# Write miss, to read shared line



# Concurrency Management

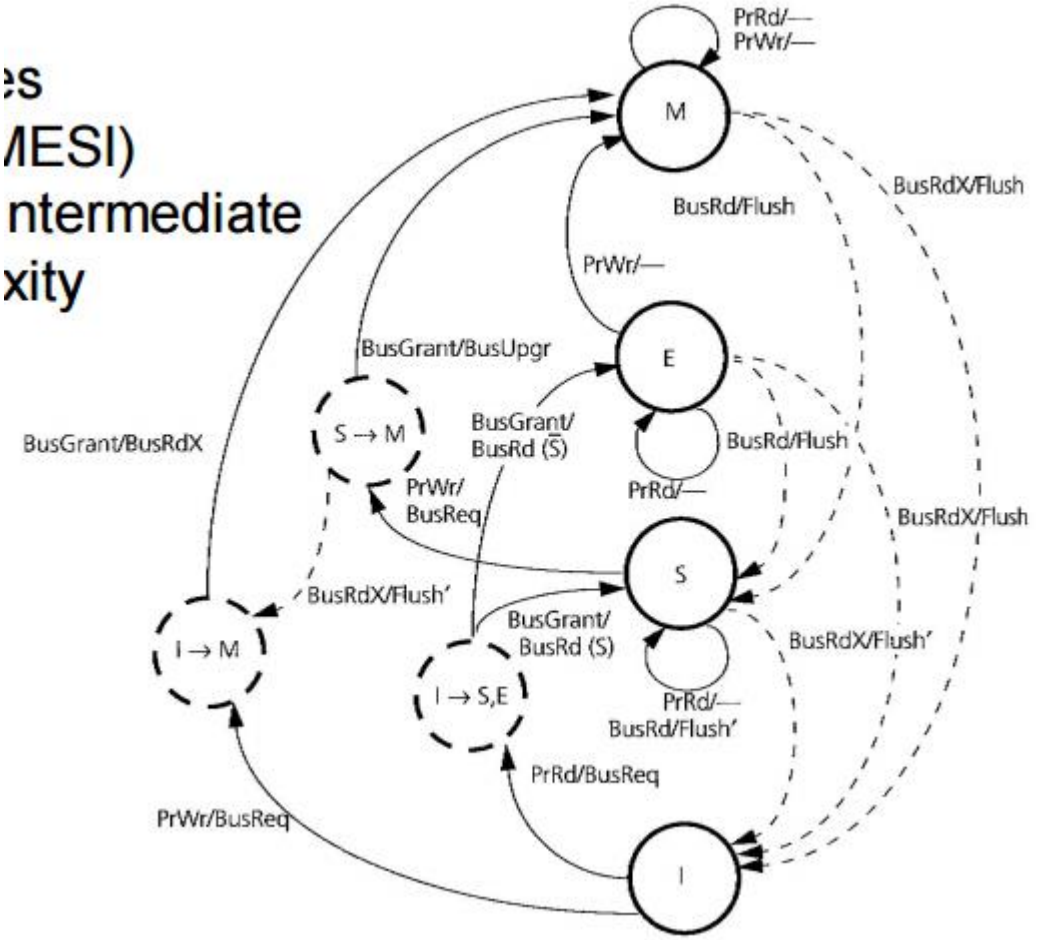
- Protocol would be easy to design if only one transaction in flight across entire system
- But, want greater throughput and don't want to have to coordinate across entire system
- Great complexity in managing multiple outstanding concurrent transactions to cache lines
  - Can have multiple requests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

# This is The Standard MESI



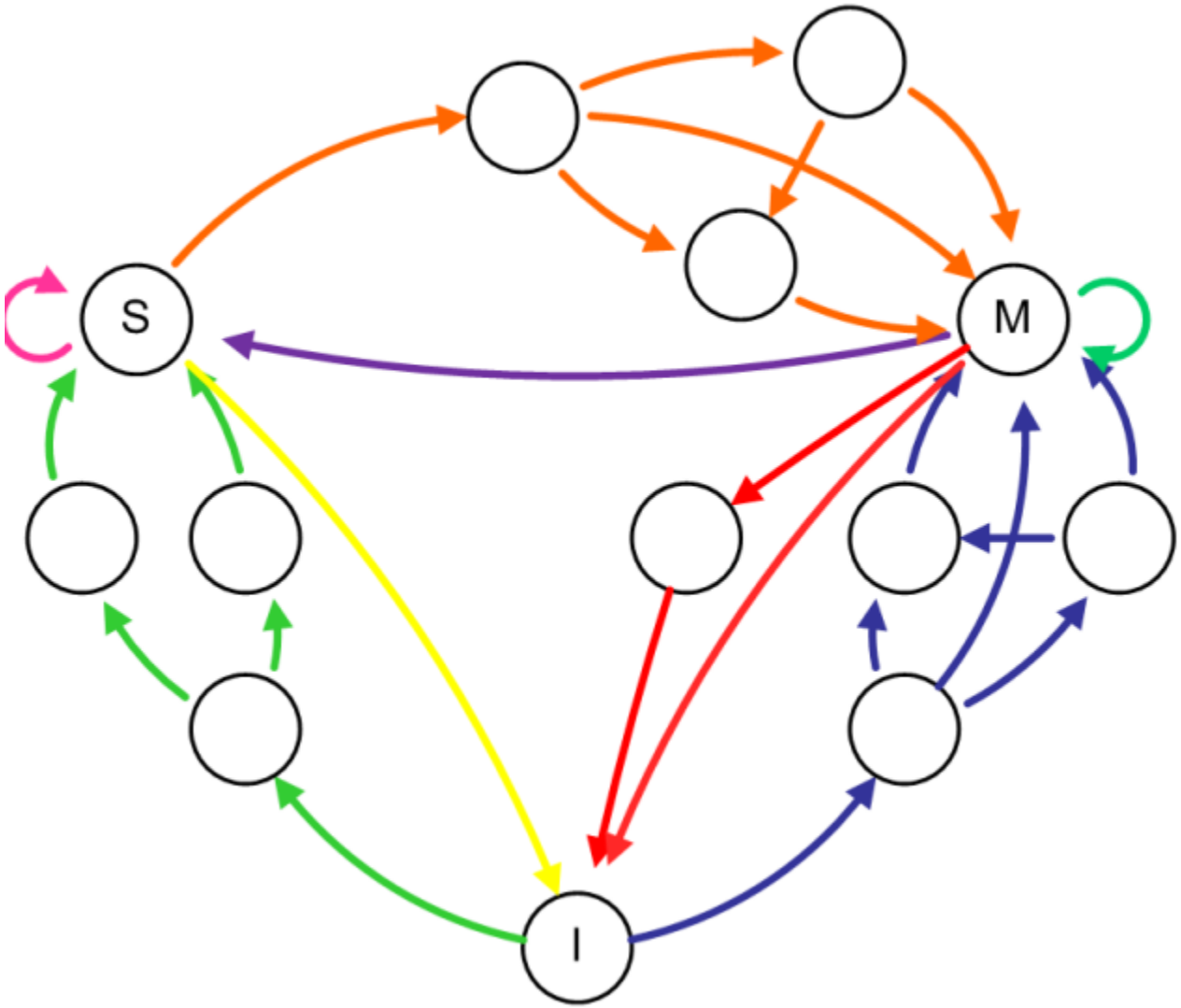
# With Transients (Cache)

Based on MESI

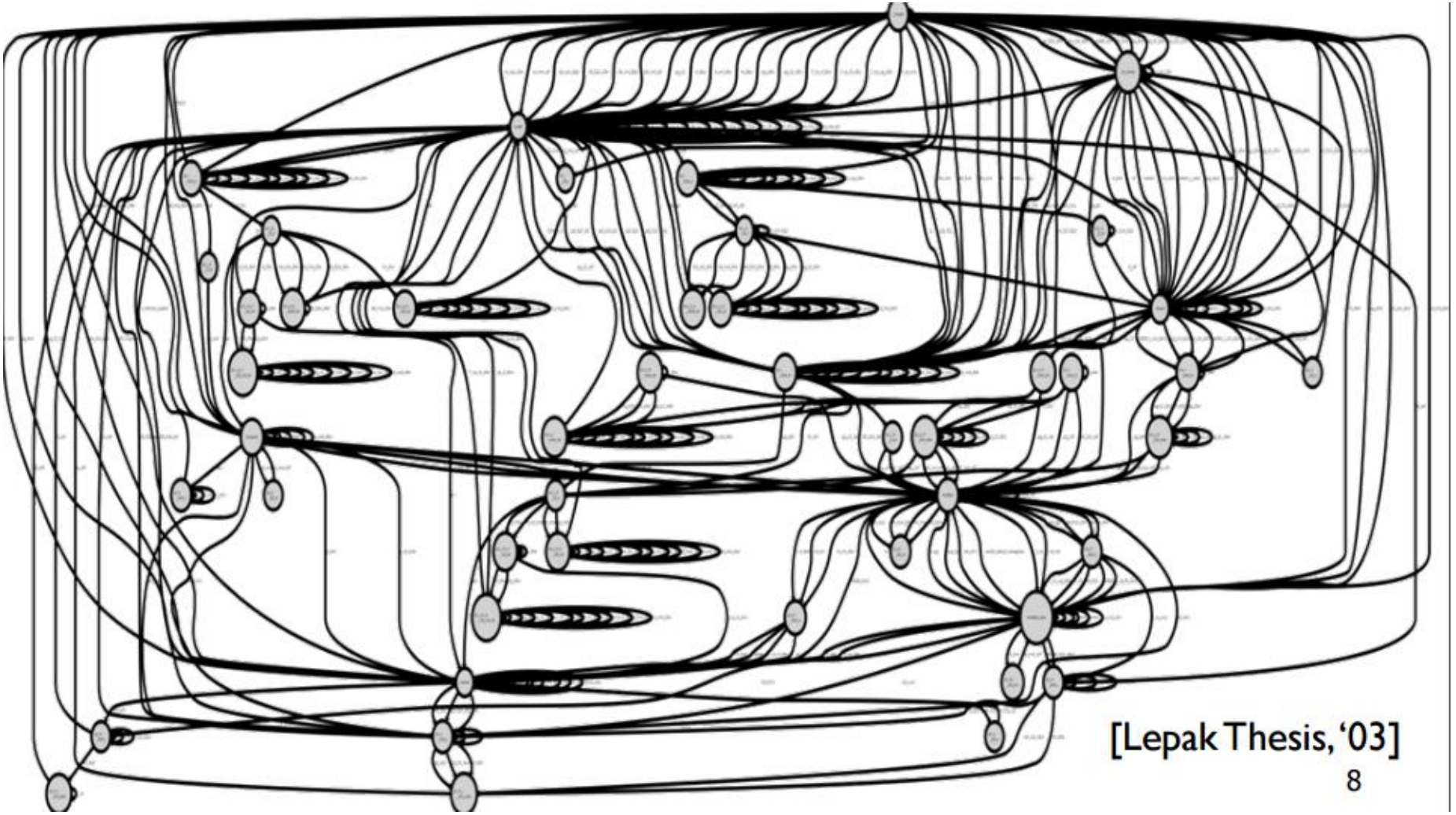


# With Transient (Directory)

- 17 states with MESI!
  - If you are curious: http://www.m5sim.org/MESI_Two_Level
- Figure based on MSI: 13 states



# More Complex Coherence Protocols



# Protocol Messages (MESI)

There are 10 different protocol messages:

| Category | Messages |
|---|---|
| Cache to Memory Requests | `ShReq, ExReq` |
| Memory to Cache Requests | `WbReq, InvReq, FlushReq` |
| Cache to Memory Responses | `WbRep(v), InvRep, FlushRep(v)` |
| Memory to Cache Responses | `ShRep(v), ExRep(v)` |

# Cache State Transitions (from invalid state)

| No. | Current State | Handling Message | Next State | Dequeue Message? | Action |
|---|---|---|---|---|---|
| 1 | C-nothing | Load | C-pending | No | ShReq(id,Home,a) |
| 2 | C-nothing | Store | C-pending | No | ExReq(id,Home,a) |
| 3 | C-nothing | WbReq(a) | C-nothing | Yes | None |
| 4 | C-nothing | FlushReq(a) | C-nothing | Yes | None |
| 5 | C-nothing | InvReq(a) | C-nothing | Yes | None |
| 6 | C-nothing | ShRep (a) | C-shared | Yes | updates cache with prefetch data |
| 7 | C-nothing | ExRep (a) | C-exclusive | Yes | updates cache with data |

# Cache State Transitions (from shared state)

| No. | Current State | Handling Message | Next State | Dequeue Message? | Action |
|---|---|---|---|---|---|
| 8 | C-shared | Load | C-shared | Yes | Reads cache |
| 9 | C-shared | WbReq(a) | C-shared | Yes | None |
| 10 | C-shared | FlushReq(a) | C-nothing | Yes | InvRep(id, Home, a) |
| 11 | C-shared | InvReq(a) | C-nothing | Yes | InvRep(id, Home, a) |
| 12 | C-shared | ExRep(a) | C-exclusive | Yes | None |
| 13 | C-shared | (Voluntary Invalidate) | C-nothing | N/A | InvRep(id, Home, a) |

# Cache State Transitions (from exclusive state)

| No. | Current State | Handling Message | Next State | Dequeue Message? | Action |
|---|---|---|---|---|---|
| 14 | C-exclusive | Load | C-exclusive | Yes | reads cache |
| 15 | C-exclusive | Store | C-exclusive | Yes | writes cache |
| 16 | C-exclusive | WbReq(a) | C-shared | Yes | WbRep(id, Home, data(a)) |
| 17 | C-exclusive | FlushReq(a) | C-nothing | Yes | FlushRep(id, Home, data(a)) |
| 18 | C-exclusive | (Voluntary Writeback) | C-shared | N/A | WbRep(id, Home, data(a)) |
| 19 | C-exclusive | (Voluntary Flush) | C-nothing | N/A | FlushRep(id, Home, data(a)) |

# Cache Transitions (from pending)

| No. | Current State | Handling Message | Next State | Dequeue Message? | Action |
|---|---|---|---|---|---|
| 20 | C-pending | WbReq(a) | C-pending | Yes | None |
| 21 | C-pending | FlushReq(a) | C-pending | Yes | None |
| 22 | C-pending | InvReq(a) | C-pending | Yes | None |
| 23 | C-pending | ShRep(a) | C-shared | Yes | updates cache with data |
| 24 | C-pending | ExRep(a) | C-exclusive | Yes | update cache with data |

# Home Directory State Transitions

| No. | Current State | Message Received | Next State | Dequeue Message? | Action |
|---|---|---|---|---|---|
| 1 | R(dir) & (dir = ε) | ShReq(a) | R({id}) | Yes | ShRep(Home, id, data(a)) |
| 2 | R(dir) & (dir = ε) | ExReq(a) | W(id) | Yes | ExRep(Home, id, data(a)) |
| 3 | R(dir) & (dir = ε) | (Voluntary Prefetch) | R({id}) | N/A | ShRep(Home, id, data(a)) |
| 4 | R(dir) & (id ∉ dir) & (dir ≠ ε) | ShReq(a) | R(dir + {id}) | Yes | ShRep(Home, id, data(a)) |
| 5 | R(dir) & (id ∉ dir) & (dir ≠ ε) | ExReq(a) | Tr(dir) | No | InvReq(Home, dir, a) |
| 6 | R(dir) & (id ∉ dir) & (dir ≠ ε) | (Voluntary Prefetch) | R(dir + {id}) | N/A | ShRep(Home, id, data(a)) |

Messages sent from site *id*

# Home Directory State Transitions

| No. | Current State | Message Received | Next State | Dequeue Message? | Action |
|---|---|---|---|---|---|
| 7 | R(dir) & (dir = {id}) | ShReq(a) | R(dir) | Yes | None |
| 8 | R(dir) & (dir = {id}) | ExReq(a) | W(id) | Yes | ExRep(Home, id, data(a)) |
| 9 | R(dir) & (dir = {id}) | InvRep(a) | R(ε) | Yes | None |
| 10 | R(dir) & (id ∈ dir) & (dir ≠ {id}) | ShReq(a) | R(dir) | Yes | None |
| 11 | R(dir) & (id ∈ dir) & (dir ≠ {id}) | ExReq(a) | Tr(dir-{id}) | No | InvReq(Home, dir - {id}, a) |
| 12 | R(dir) & (id ∈ dir) & (dir ≠ {id}) | InvRep(a) | R(dir - {id}) | Yes | None |

Messages sent from site *id*

# Home Directory State Transitions

| No. | Current State | Message Received | Next State | Dequeue Message? | Action |
|---|---|---|---|---|---|
| 13 | W(id') | ShReq(a) | Tw(id') | No | WbReq(Home, id', a) |
| 14 | W(id') | ExReq(a) | Tw(id') | No | FlushReq(Home, id', a) |
| 15 | W(id) | ExReq(a) | W(id) | Yes | None |
| 16 | W(id) | WbRep(a) | R({id}) | Yes | data -> memory |
| 17 | W(id) | FlushRep(a) | R(ε) | Yes | data -> memory |

Messages sent from site *id*

# Home Directory State Transitions

| No. | Current State | Message Received | Next State | Dequeue Message? | Action |
|---|---|---|---|---|---|
| 18 | Tr(dir) & (id ∈ dir) | InvRep(a) | Tr(dir - {id}) | Yes | None |
| 19 | Tr(dir) & (id ∉ dir) | InvRep(a) | Tr(dir) | Yes | None |
| 20 | Tw(id) | WbRep(a) | R({id}) | Yes | data-> memory |
| 21 | Tw(id) | FlushRep(a) | R(ε) | Yes | data-> memory |

Messages sent from site *id*

# Acknowledgements


- These slides contain material developed and copyright by:
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)
  - Mark Hill (U. Wisconsin-Madison)
  - Dana Vantrase, Mikko Lipasti, Nathan Binkert (U. Wisconsin-Madison & HP)

- MIT material derived from course 6.823
- UCB material derived from course CS252