Computer Architecture

Cache Coherence

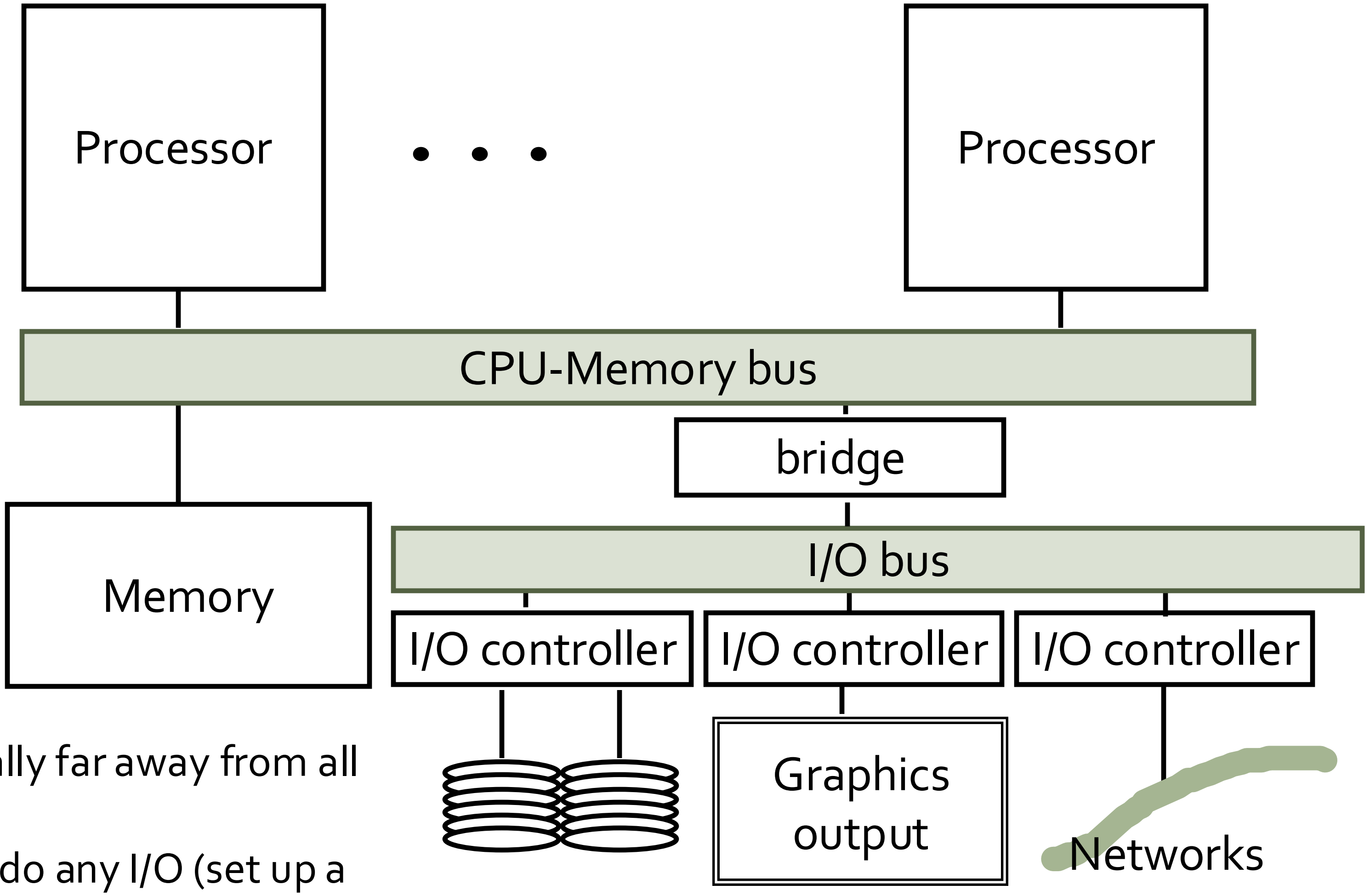
Ting-Jung Chang

NYCU CS

Agenda

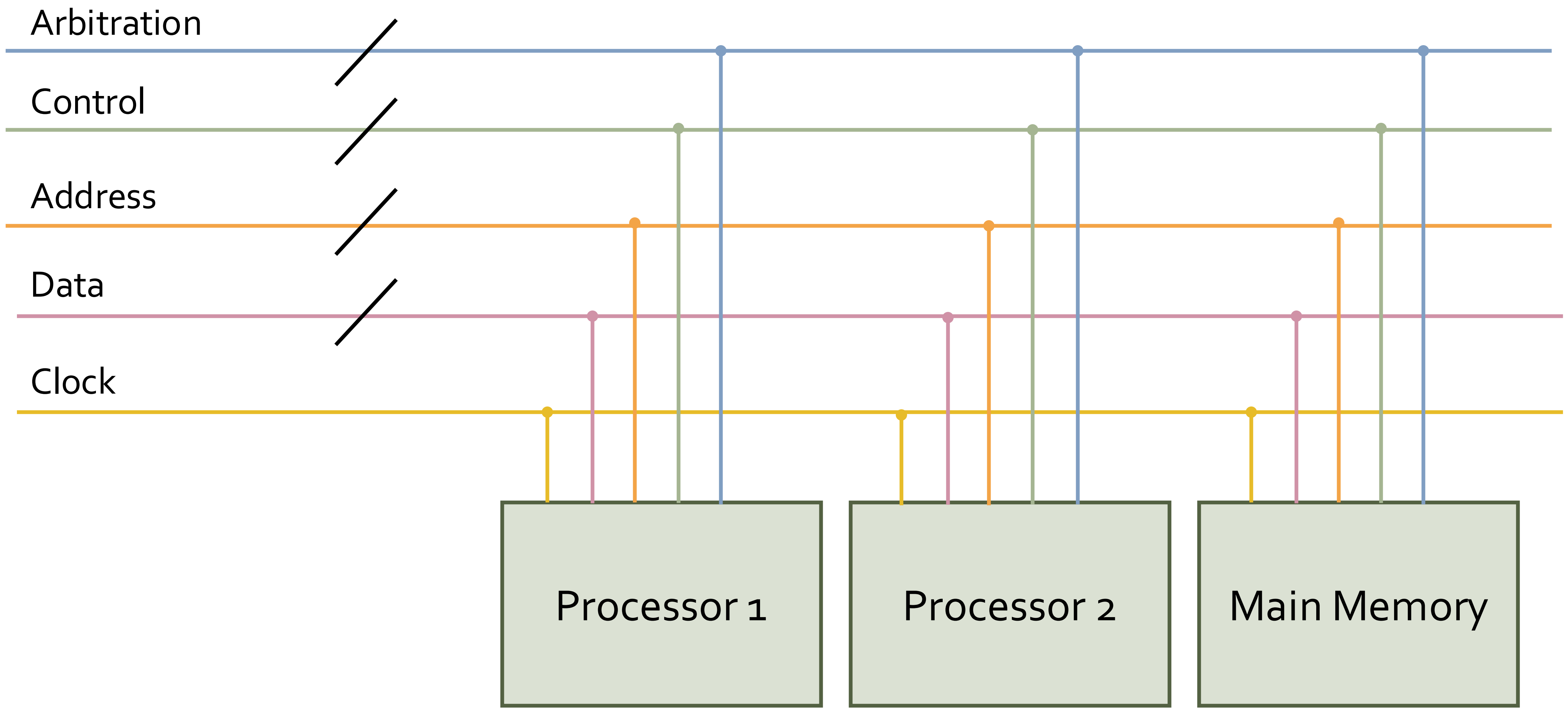
- Cache Coherence

Symmetric Multiprocessors

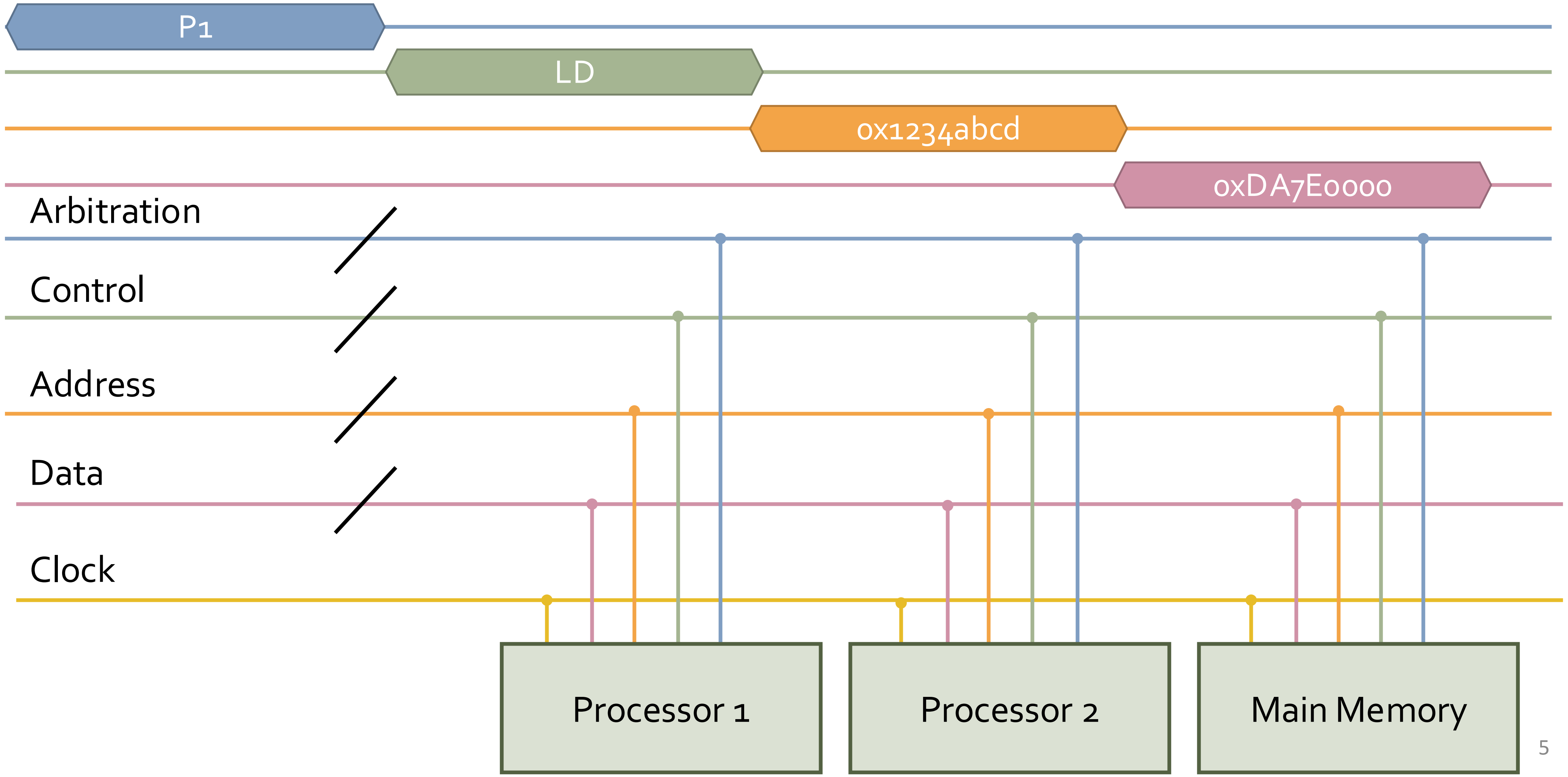


- All memory is equally far away from all processors
- Any processor can do any I/O (set up a DMA transfer)

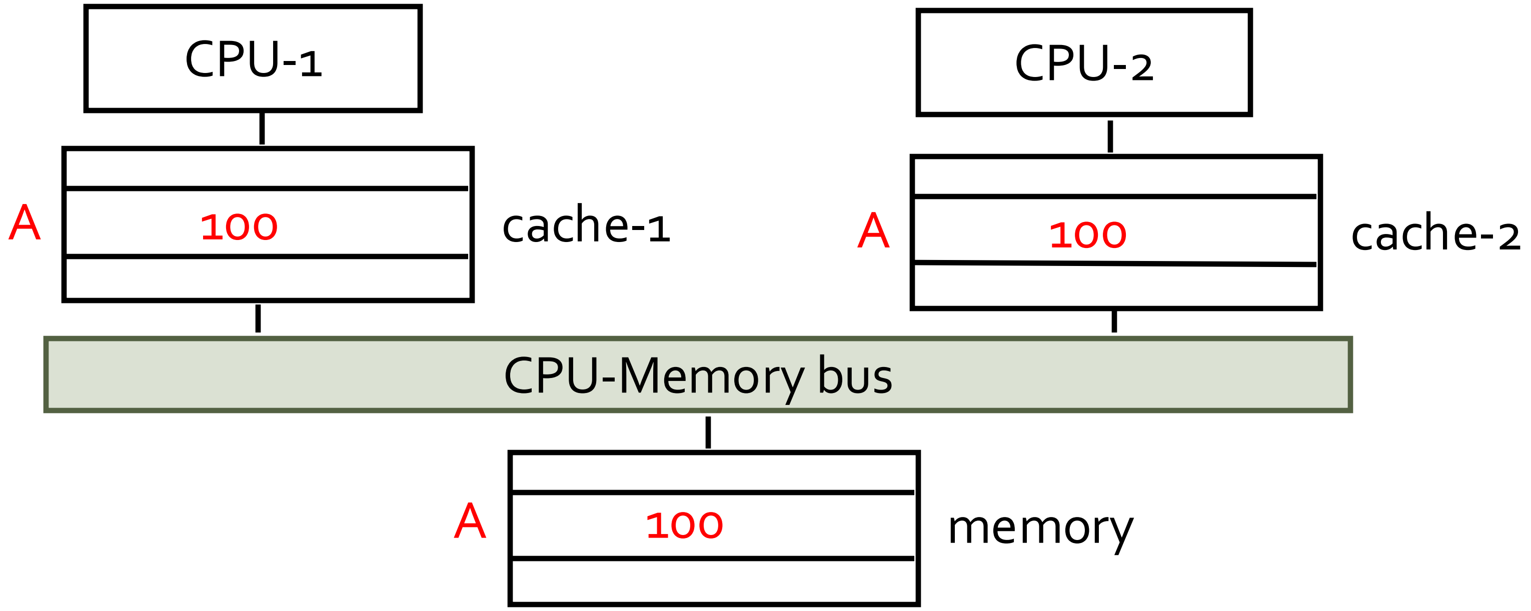
Multidrop Memory Bus



Pipelined Memory Bus



Memory Coherence in SMPs



- Suppose CPU-1 updates A to 200.
    - write-back: memory and cache-2 have stale values
    - write-through: cache-2 has a stale value

- Do these stale values matter?
- What is the view of shared memory for programming?

Recap: Sequential Consistency Example

Sequential concurrent tasks: T1, T2
Shared variables: X, Y (initially X = 0, Y = 10)

```
T1:                           T2:
    Store 1,  (X) (X =  1)        Load  R1, (Y)
    Store 11, (Y) (Y = 11)        Store R1, (Y') (Y'= Y)
                                  Load  R2, (X)
                                  Store R2, (X') (X'= X)
```

what are the legitimate answers for X' and Y' ?

X'	Y'	SC
1	11	Y
0	10	Y
1	10	Y
0	11	N

If Y is 11 then X cannot be 0

Write-back Caches & SC

prog T1
```
ST 1, X
ST 11, Y
```

prog T2
```
LD Y, R1
ST R1, Y'
LD X, R2
ST R2, X'
```

	cache-1	memory	cache-2

• T1 is executed	X= 1
	Y=11
		X = 0
		Y =10
		X'=
		Y'=
			Y =
			Y'=
			X =
			X'=

• cache-1 writes back Y	X= 1
	Y=11
		X = 0
		Y =11
		X'=
		Y'=
			Y =
			Y'=
			X =
			X'=

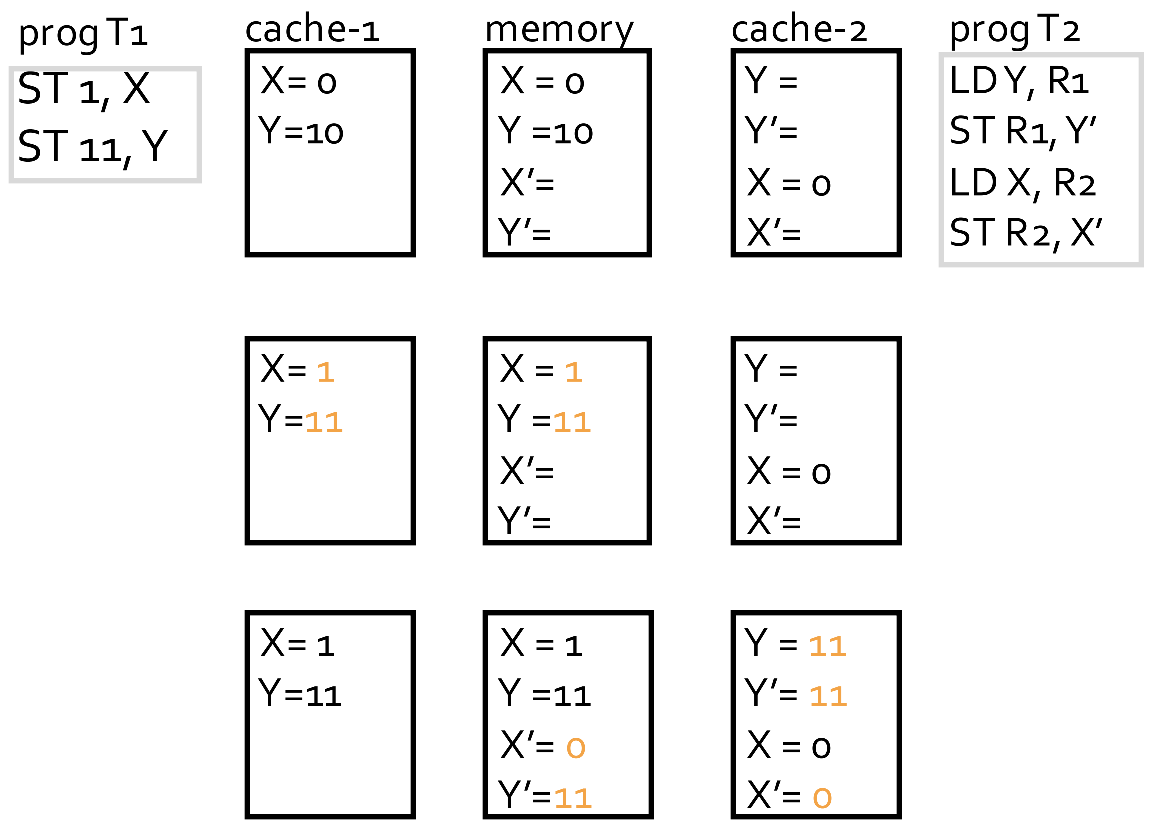
• T2 executed	X= 1
	Y=11
		X = 0
		Y =11
		X'=
		Y'=
			Y = 11
			Y'= 11
			X = 0
			X'= 0

• cache-1 writes back X	X= 1
	Y=11
		X = 1
		Y =11
		X'=
		Y'=
			Y = 11
			Y'= 11
			X = 0
			X'= 0

• cache-2 writes back X' & Y'	X= 1
	Y=11
		X = 1
		Y =11
		X'= 0
		Y'=11
			Y =11
			Y'=11
			X = 0
			X'= 0

inconsistent

Write-through Caches & SC



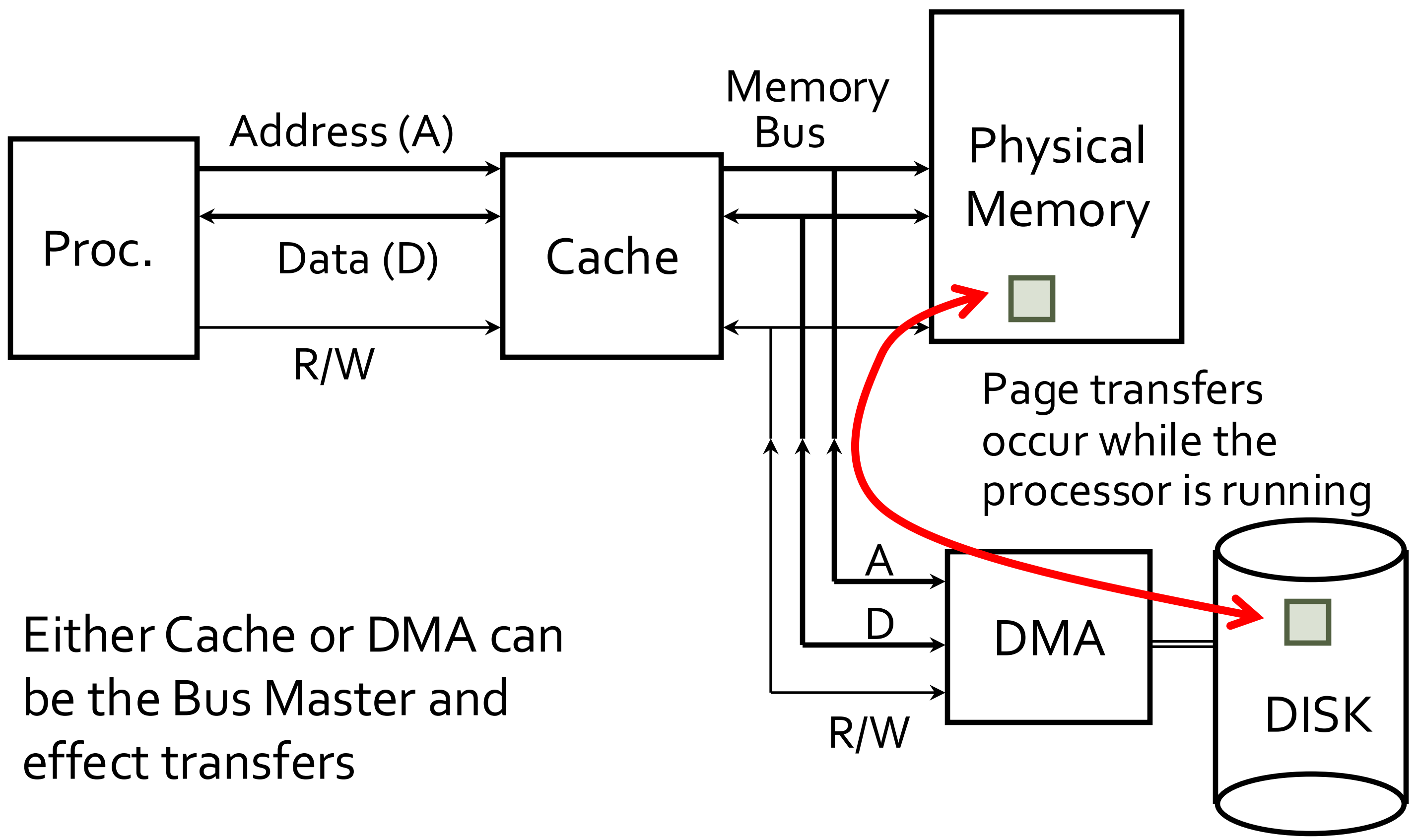
- T1 executed
- T2 executed

Write-through caches don't preserve sequential consistency either

Cache Coherence vs. Memory Consistency

- A cache coherence protocol ensures that all writes by one processor are eventually visible to other processors, for one memory address
  - i.e., updates are not lost
- A memory consistency model gives the rules on when a write by one processor can be observed by a read on another, across different addresses
  - Equivalently, what values can be seen by a load
- A cache coherence protocol is not enough to ensure sequential consistency
  - But if sequentially consistent, then caches must be coherent
- Combination of cache coherence protocol plus processor memory reorder buffer implements a given machine's memory consistency model

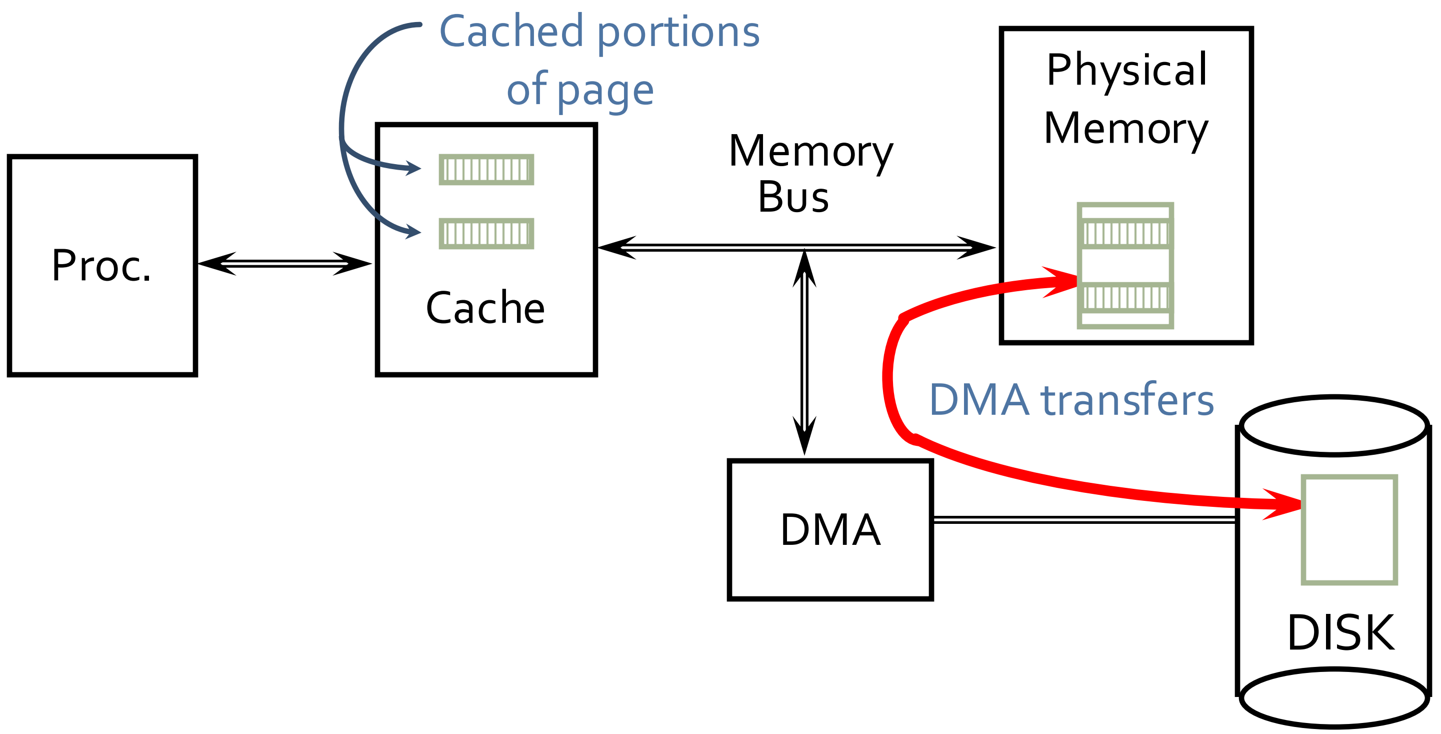
Warmup: Parallel I/O



Either Cache or DMA can be the Bus Master and effect transfers

(DMA stands for "Direct Memory Access", means the I/O device can read/write memory autonomous from the CPU)

Problems with Parallel I/O



Memory ⟶ Disk: Physical memory may be stale if cache copy is dirty
Disk ⟶ Memory: Cache may hold stale data and not see memory writes

Implementing Cache Coherence

- Coherence protocols must enforce two rules:
  - Write propagation: Writes eventually become visible to all processors
  - Write serialization: Writes to the same location are serialized (all processors see them in the same order)
- How to ensure write propagation?
  - Write-invalidate protocols: Invalidate all other cached copies before performing the write
  - Write-update protocols: Update all other cached copies after performing the write
- How to track sharing state of cached data and serialize requests to the same address?
  - Snooping-based protocols: All caches observe each other's actions through a shared bus (bus is the serialization point)
  - Directory-based protocols: A coherence directory tracks contents of private caches and serializes requests (directory is the serialization point)

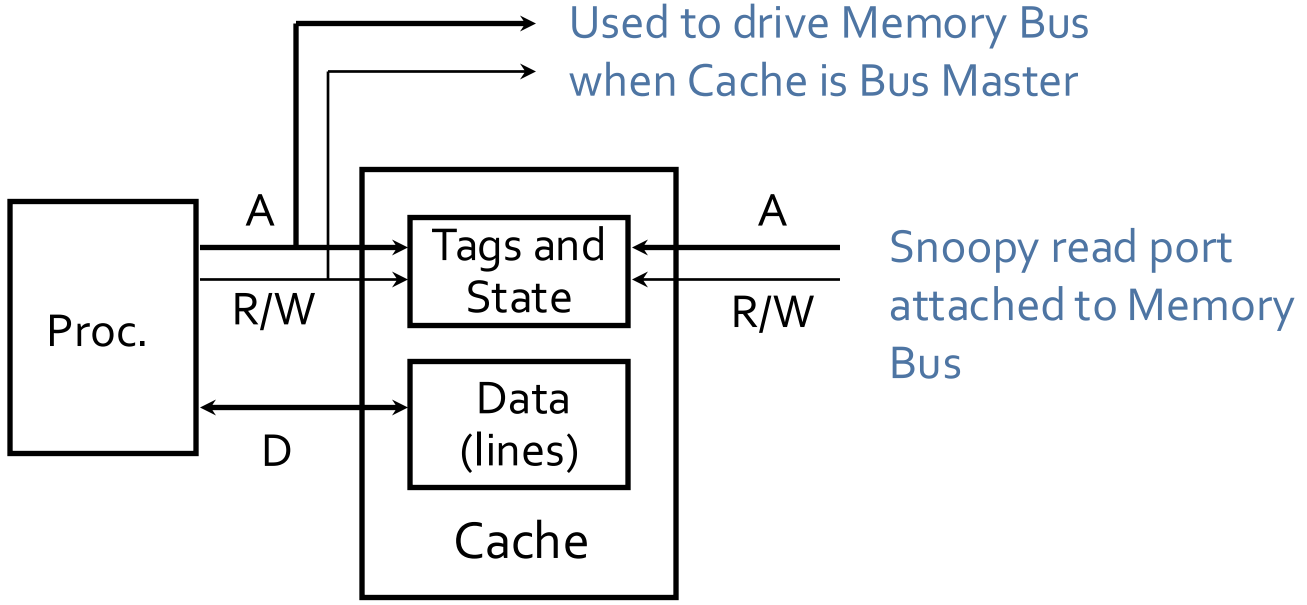
Agenda

- Cache Coherence
    - Snoopy coherence

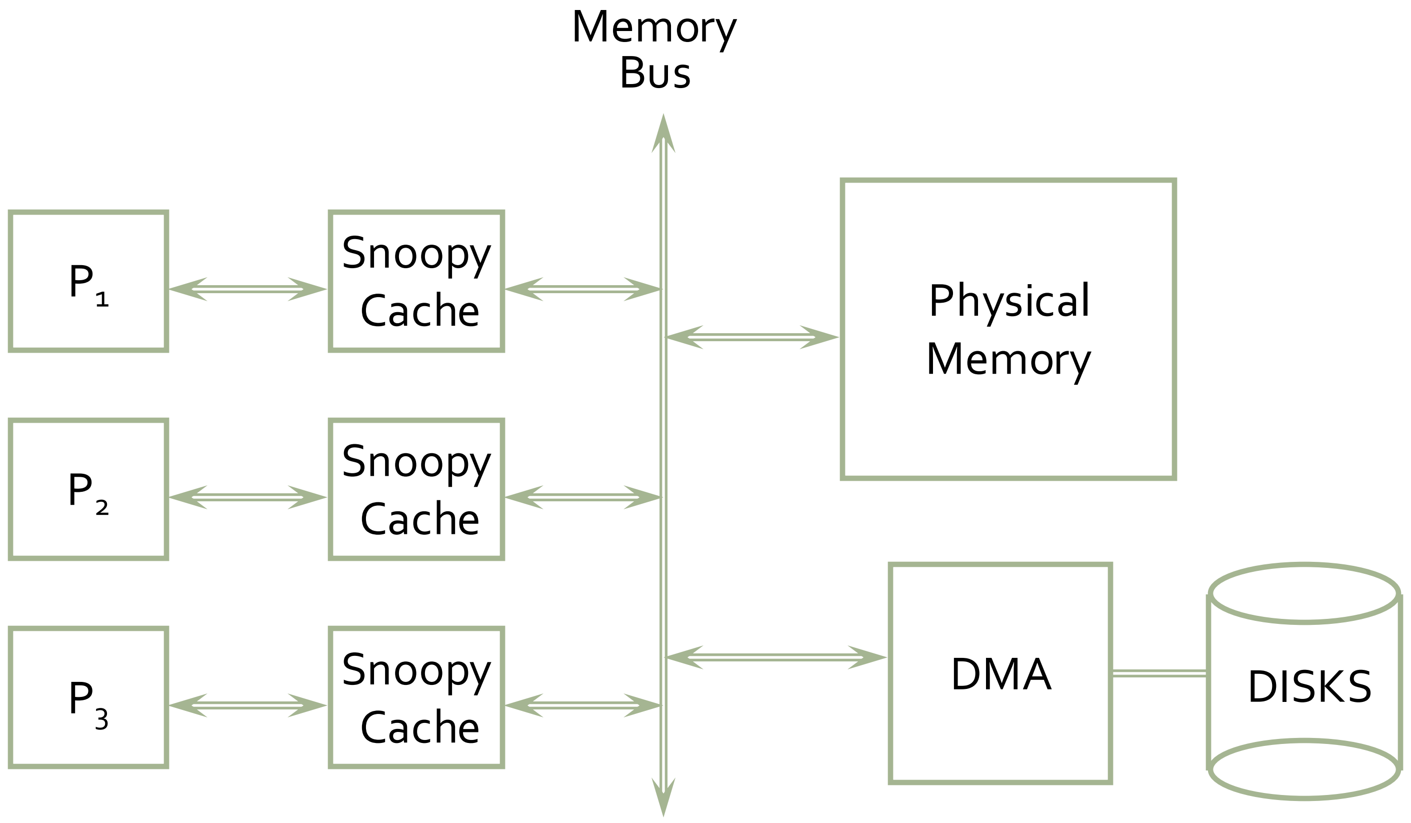
Snoopy Cache

Goodman & Ravishankar 1983

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


Used to drive Memory Bus when Cache is Bus Master
A
R/W
Proc.
D
Tags and State
Data (lines)
Cache
A
R/W
Snoopy read port attached to Memory Bus

Shared Memory Multiprocessor



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

Snooping-Based Coherence

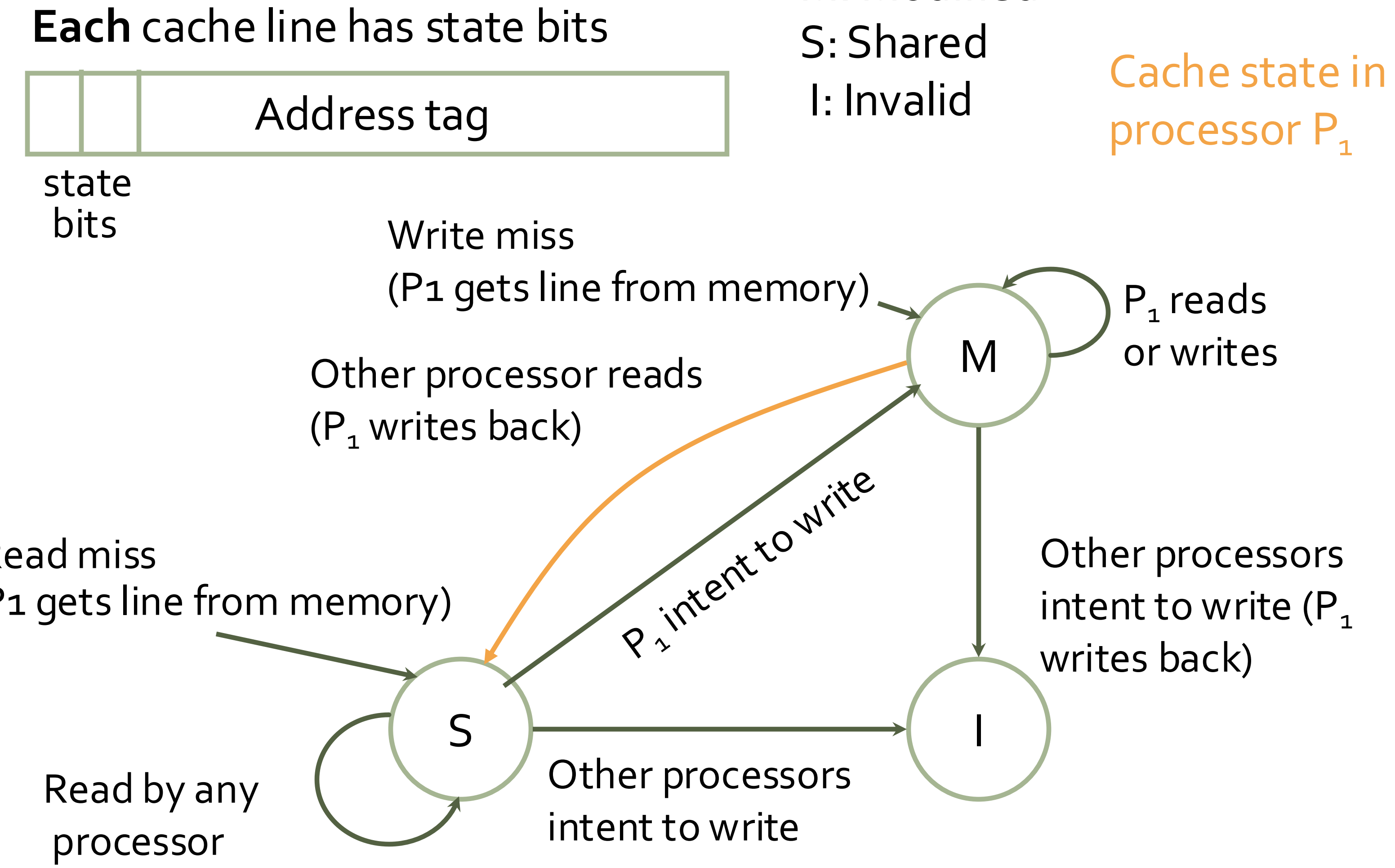
- Bus provides serialization point
  - Broadcast, totally ordered
- Controller
  - One cache controller for each core “snoops” all bus transactions
  - Controller
    - Responds to requests from core and the bus
    - changes state of the selected cache block
    - generates bus transactions to access data or invalidate
- Snoopy protocol (FSM)
  - State-transition diagram
  - Actions
- Handling writes:
  - Write-invalidate
  - Write-update

Update (Broadcast) vs. Invalidate Snoopy Cache Coherence Protocols

- Write Update (Broadcast)
  - Writes are broadcast and update all other cache copies
  - Write miss:
    - Broadcast on bus, other processors update copies (in place)
  - Read miss:
    - Memory is always up to date
- Write Invalidate
  - Writes invalidate all other cache copies
  - 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

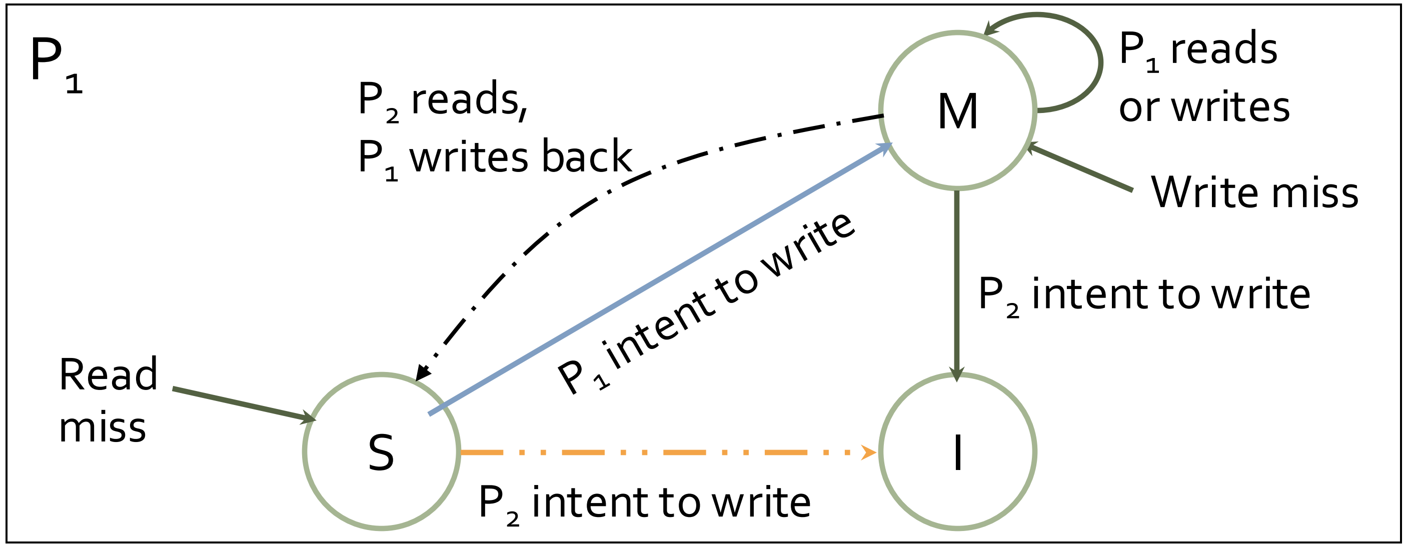


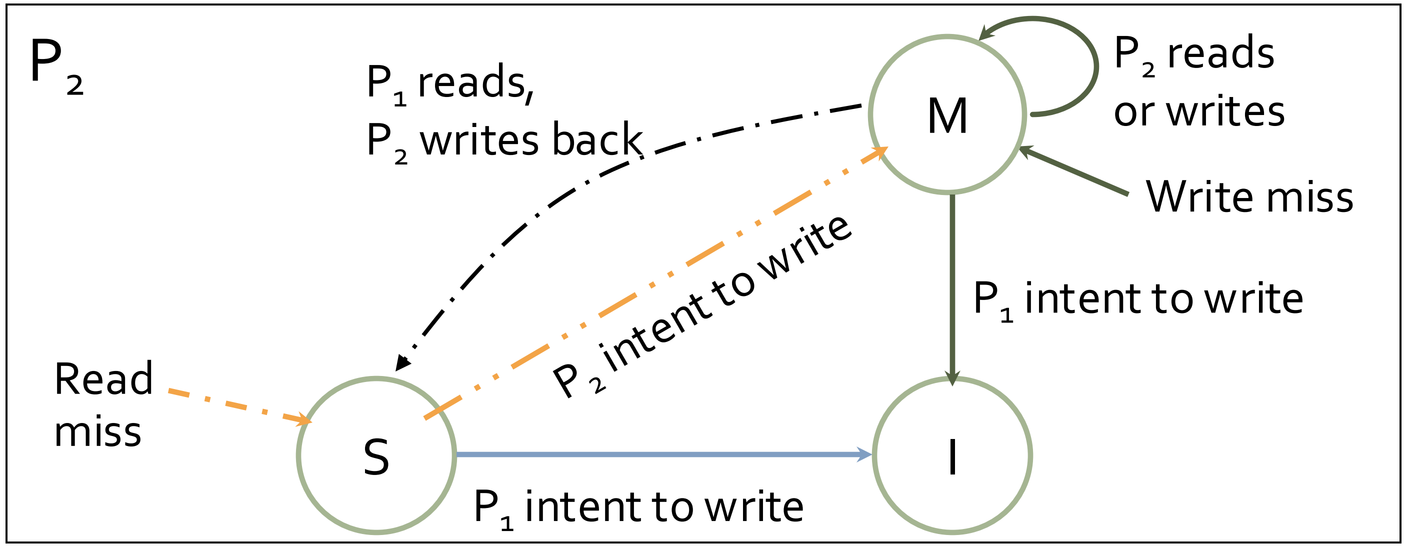
Each cache line has state bits
Address tag
state bits
M: Modified
S: Shared
I: Invalid
Cache state in processor P_1
Write miss (P1 gets line from memory)
M
P_1 reads or writes
Other processor reads (P_1 writes back)
P_1 intent to write
Other processors intent to write (P_1 writes back)
Read miss (P1 gets line from memory)
S
I
Read by any processor
Other processors intent to write

Two Processor Example

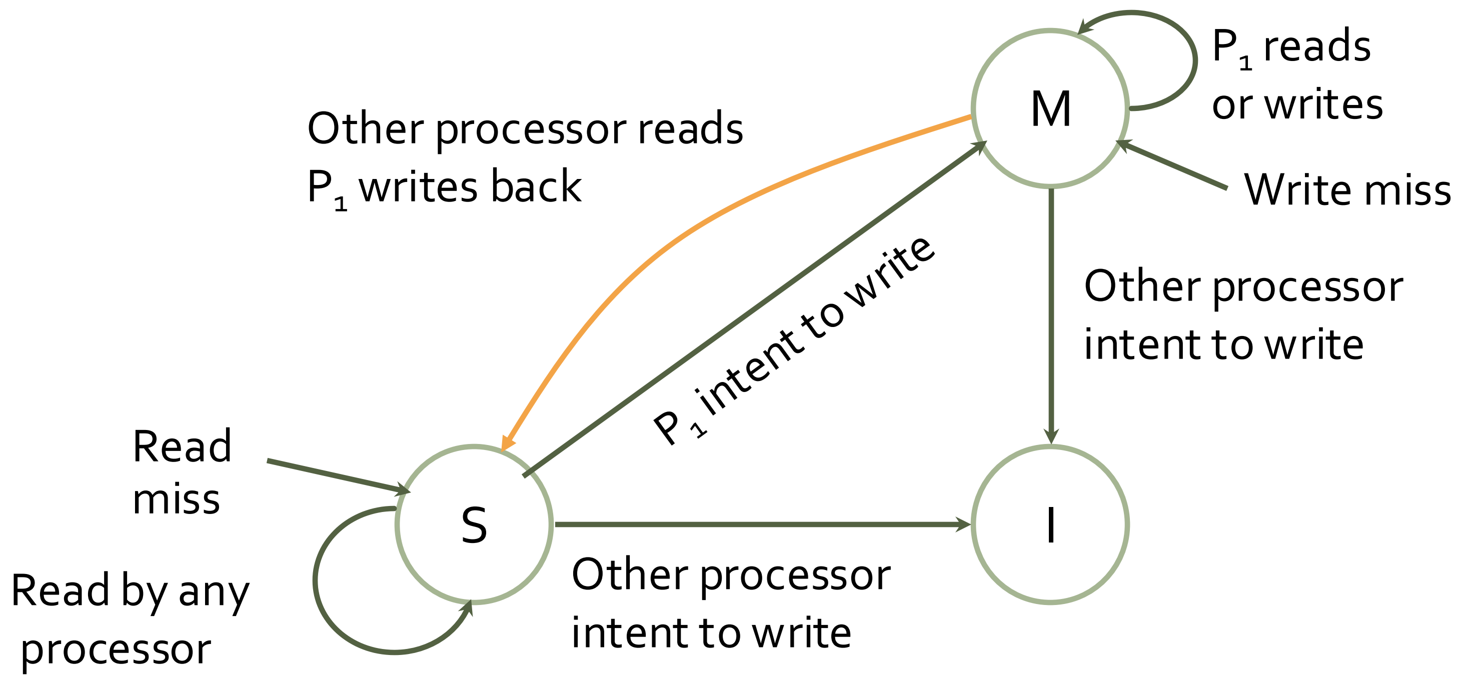
(Reading and writing the same cache line)

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

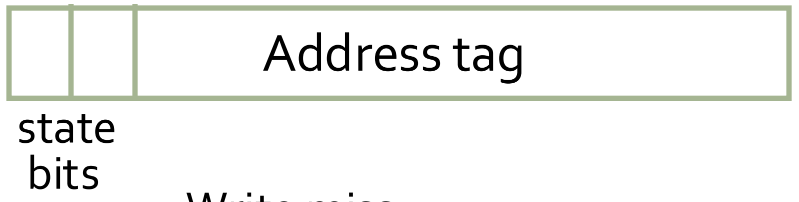
MSI Optimizations: Exclusive State

- Observation: Doing read-modify-write sequences on private data is common
  - What's the problem with MSI?
- Solution: E state (exclusive, clean)
  - If no other sharers, a read acquires line in E instead of S
  - Writes silently cause E $\rightarrow$ M (exclusive, dirty)

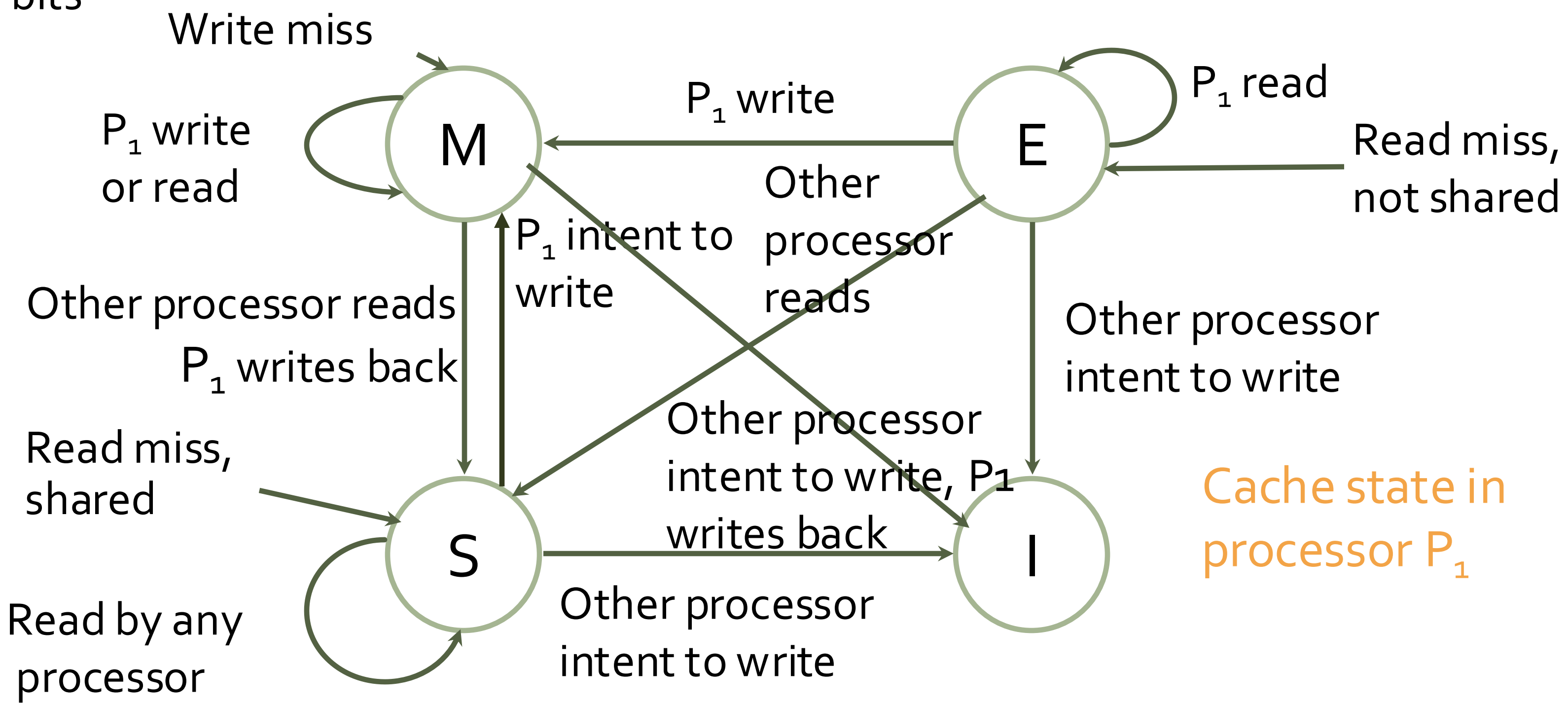
MESI: An Enhanced MSI protocol

increased performance for private data (Illinois Protocol)

Each cache line has a tag

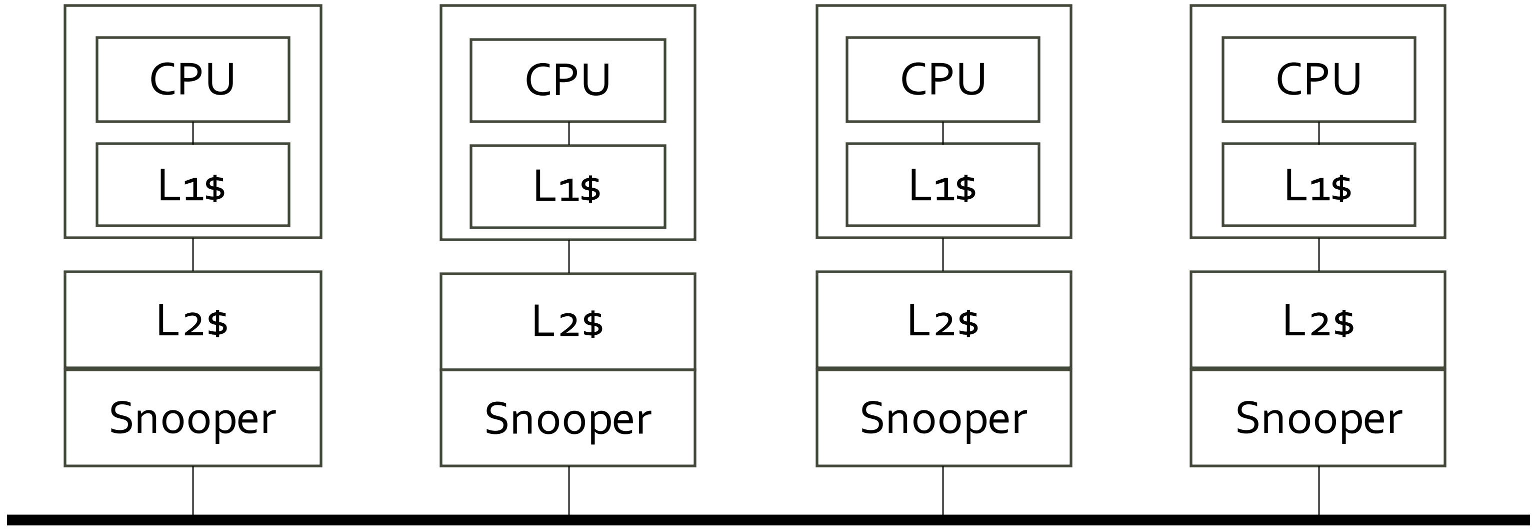


M: Modified Exclusive
E: Exclusive but unmodified
S: Shared
I: Invalid



Cache state in processor P_1

Optimized Snoop with L2 Caches



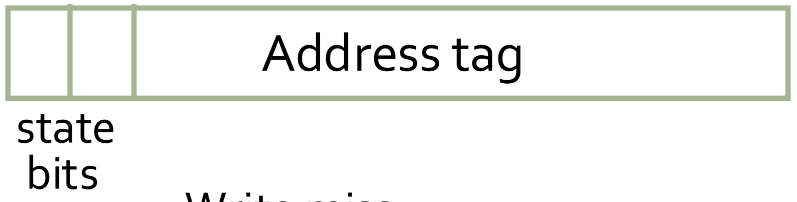
- Processors often have two-level caches
  - small L1, large L2 (usually both on chip now)
- Inclusion property: entries in L1 must be in L2
  - Miss in L2 ⇒ Not present in L1
  - Only if invalidation hits in L2 ⇒ probe and invalidate in L1
- Snooping on L2 does not affect CPU-L1 bandwidth

MSI Optimizations: Owner State

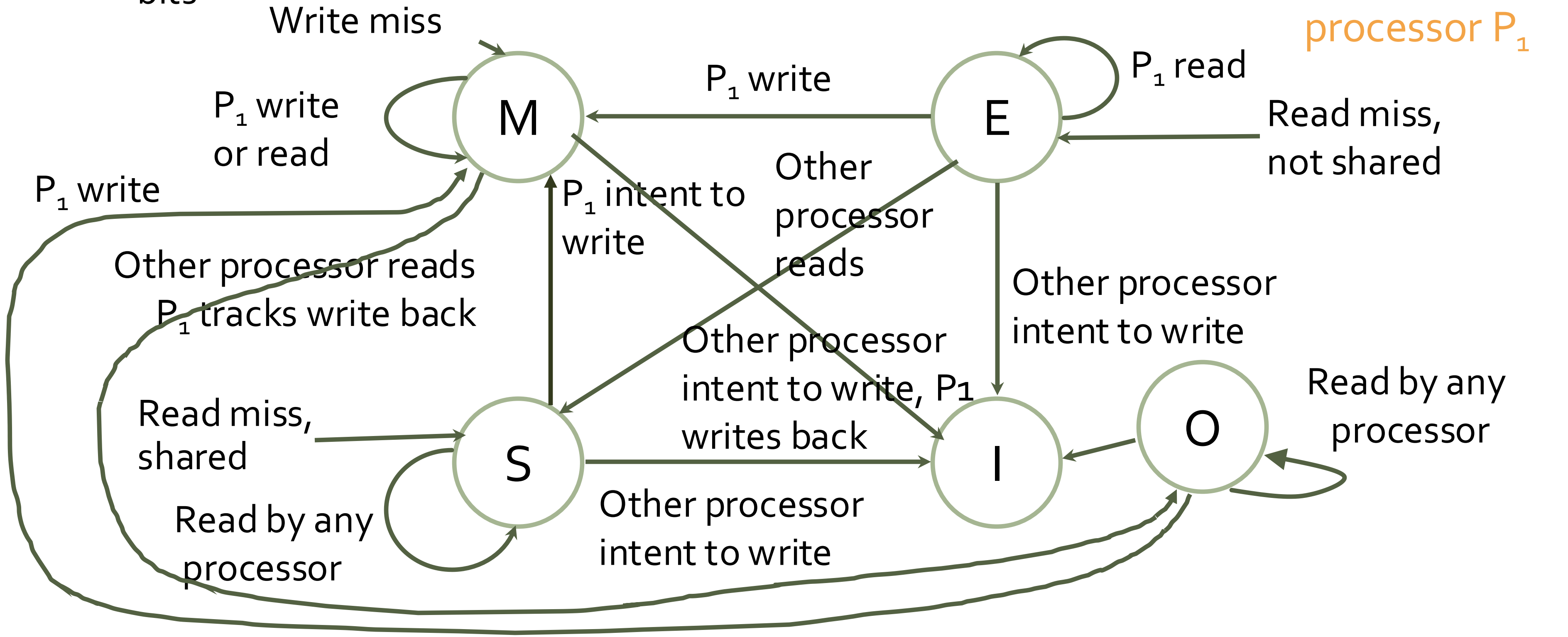
- Observation: On M→S transitions, must write back line!
- What happens with frequent read-write sharing?
- Can we defer the write after S?
- Solution: O state (Owner)
    - O = S + responsibility to write back
    - On M→S transition, one sharer (typically the one who had the line in M) retains the line in O instead of S
    - On eviction, O writes back line (or another sharer does S → O)
- MSI, MESI, MOSI, MOESI…
    - Typically E if private read-write >> shared read-only (common)
    - Typically O only if writebacks are expensive (main mem vs L3)

MOESI (Used in AMD Opteron)

Each cache line has a tag

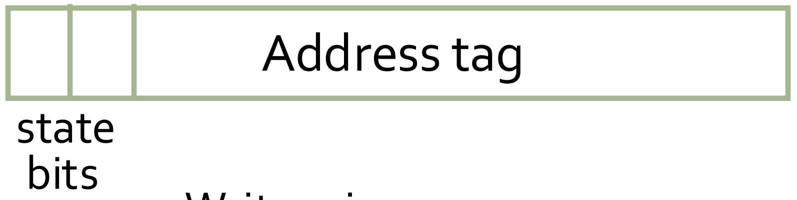


M: Modified Exclusive
O: Owned
E: Exclusive but unmodified
S: Shared
I: Invalid

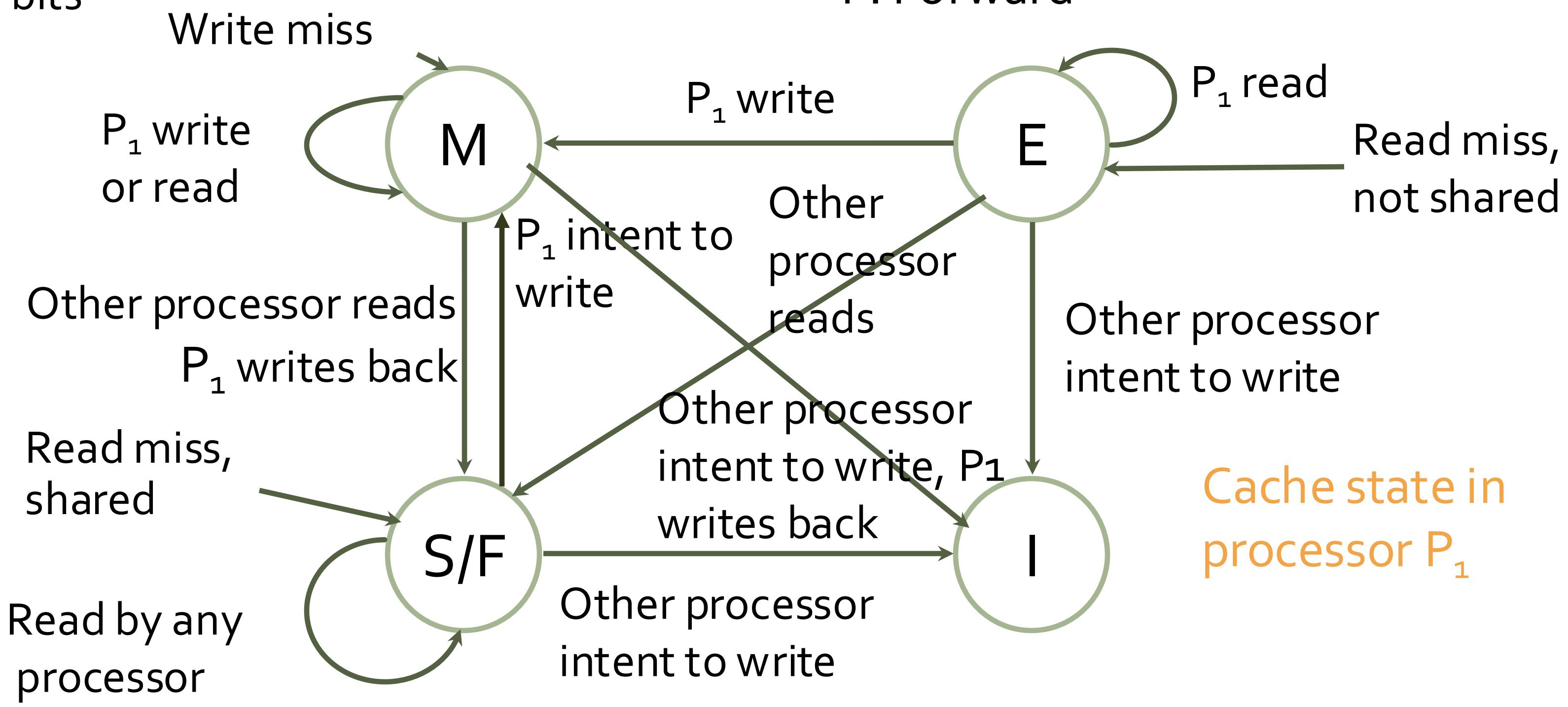


MESIF (Used by Intel Core i7)

Each cache line has a tag



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

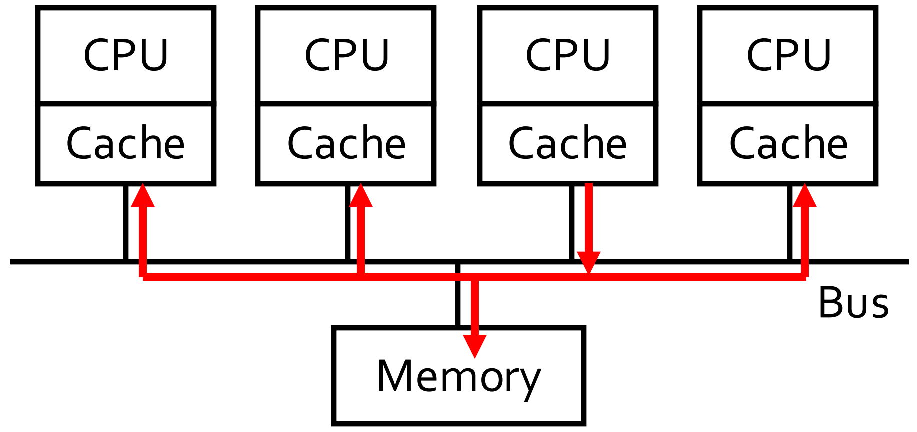


Cache state in processor P_1

Scalability Limitations of Snooping

- Caches
  - Bandwidth into caches
  - Tags need to be dual ported or steal cycles for snoops
  - Need to invalidate all the way to L1 cache
- Bus
  - Bandwidth
  - Occupancy (As number of cores grows, atomically utilizing bus becomes a challenge)

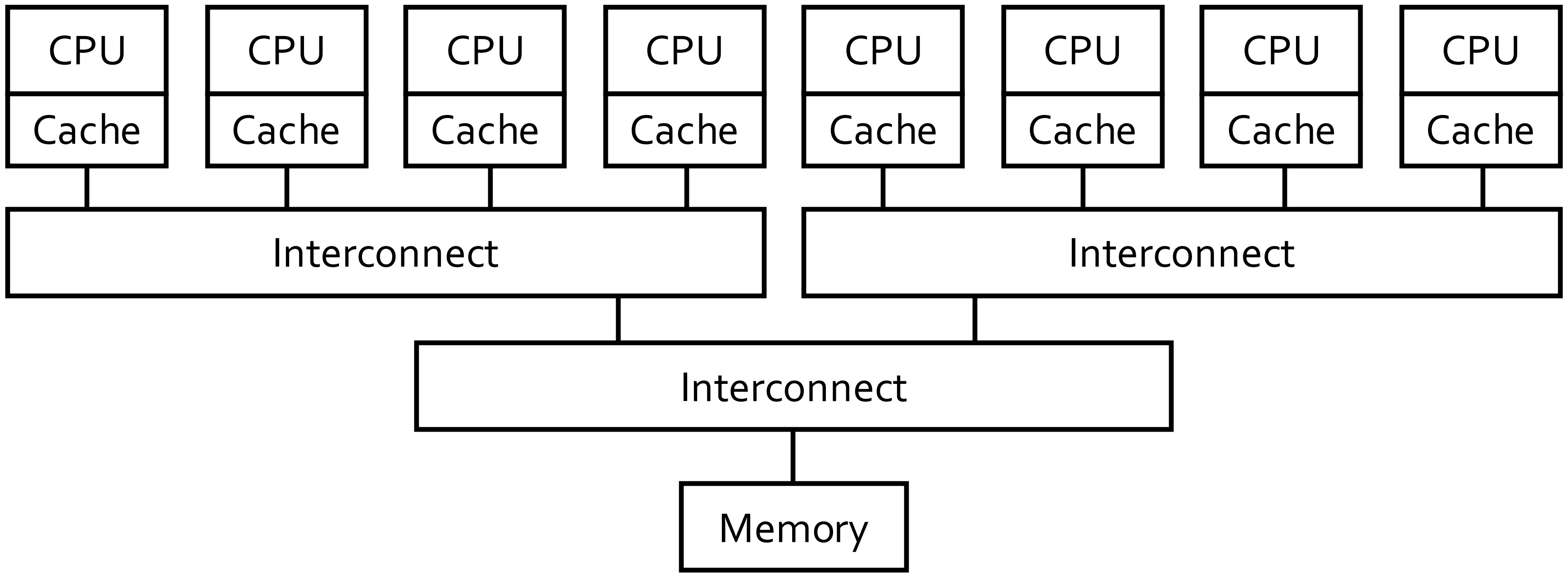
Every time a cache miss occurred, the triggering cache communicated with all other caches!



Fundamentally, all-to-all broadcast will not scale!

Scalable Approach #1: Hierarchical Snooping

- Simplest way to build large-scale cache-coherent MPs is to use a hierarchy of buses and use snoopy coherence at each level.



Scalable Approach #1: Hierarchical Snooping

- Advantages:
  - Relatively simple to build (already must deal with similar issues due to multi-level caches)
- Problems:
  - Higher latency: multiple levels, and snoop/lookup at every level
  - The root of the network can become bandwidth bottleneck
  - Larger latencies than direct communication
  - Does not apply to more general network topologies (meshes, cubes)
- Not popular today

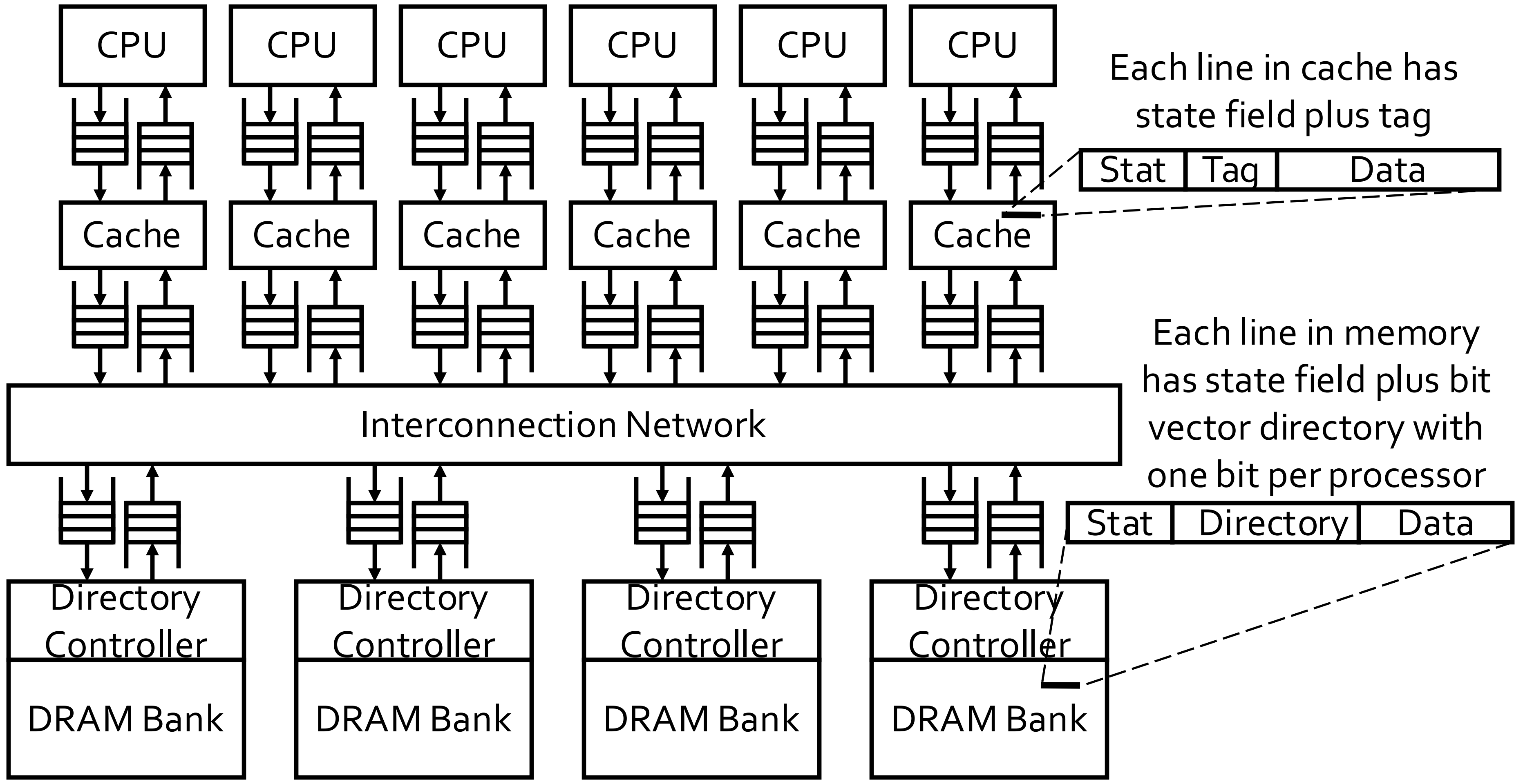
Agenda

- Cache Coherence
    - Snoopy coherence
    - Directory coherence

Scalable Approach #2: Directories

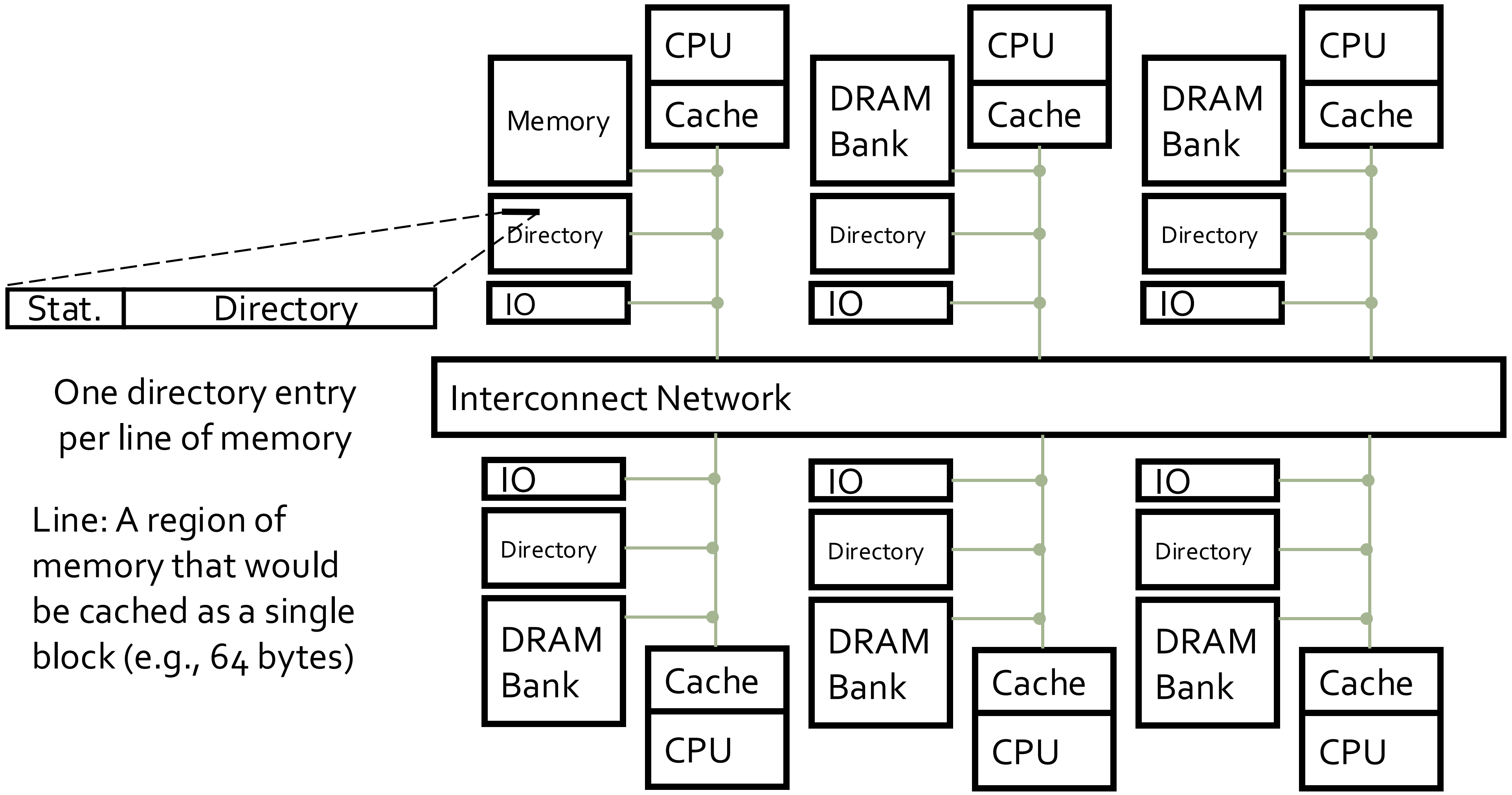
- Snoopy protocols require every cache miss to broadcast
  - Requires large bus bandwidth
  - Requires large cache snooping bandwidth aggregate
  - **Insight**: Most snoops fail to find a match!
- Alternative idea: avoid broadcast by storing information about the status of the line in one place: a "**directory**"
  - The directory entry for a cache line contains information about the state of the cache line in all caches.
  - Caches look up information from the directory as necessary
  - Cache coherence is maintained by point-to-point messages between the caches on a "need to know" basis (not by broadcast mechanisms)

Directory Cache Coherence



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

Distributed Shared Memory



One directory entry per line of memory

Line: A region of memory that would be cached as a single block (e.g., 64 bytes)

“Home node” of a line: node with memory holding the corresponding data for the line

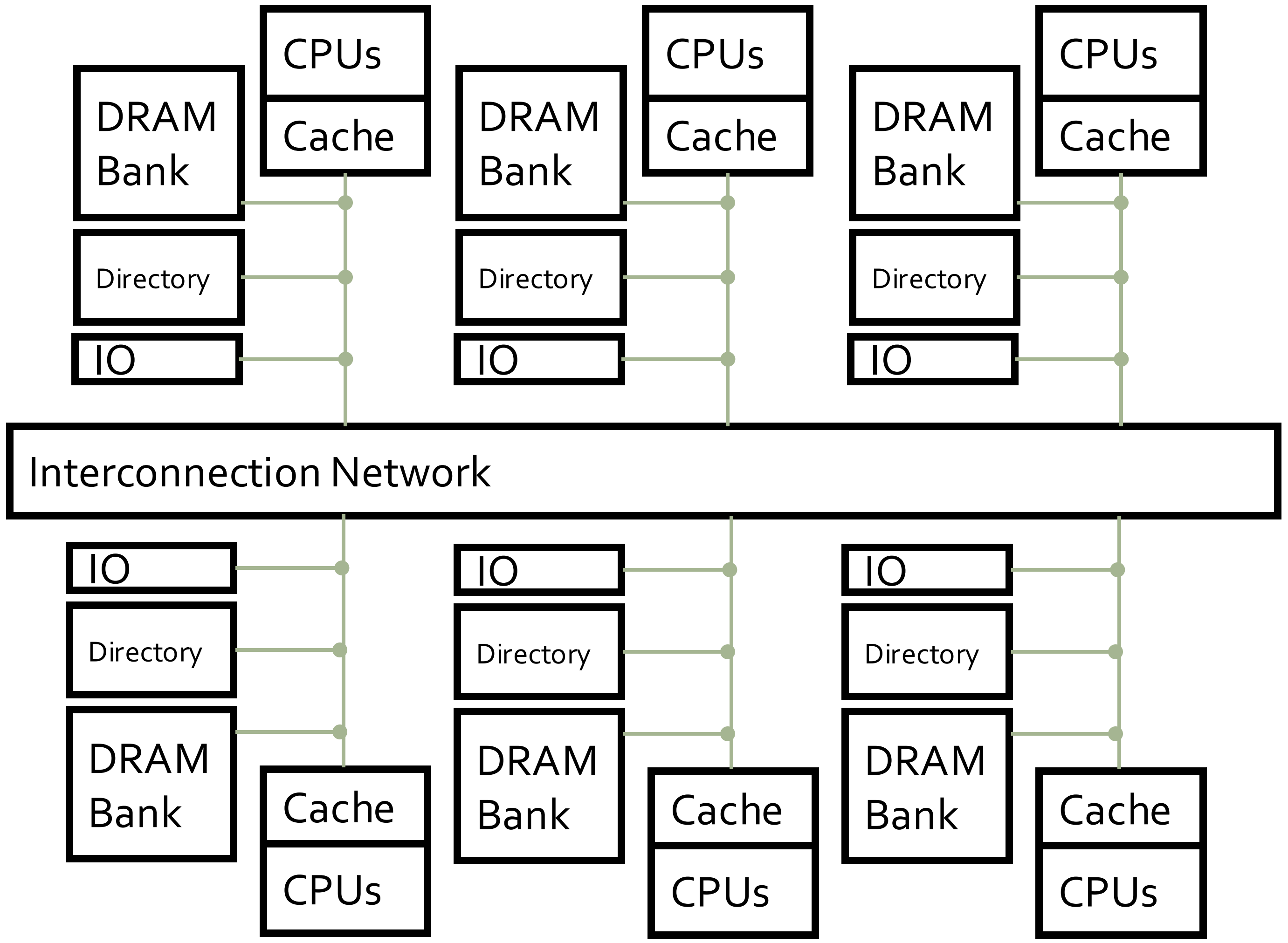
A Popular Middle Ground

- Two-level “hierarchy”
- Individual nodes are multiprocessors, connected non-hierarchically
  - e.g. mesh of SMPs
- 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 of functionality
- Early examples:
  - Convex Exemplar: directory-directory
  - Sequent, Data General, HAL: directory-snoopy

Non-Uniform Memory Access (NUMA)

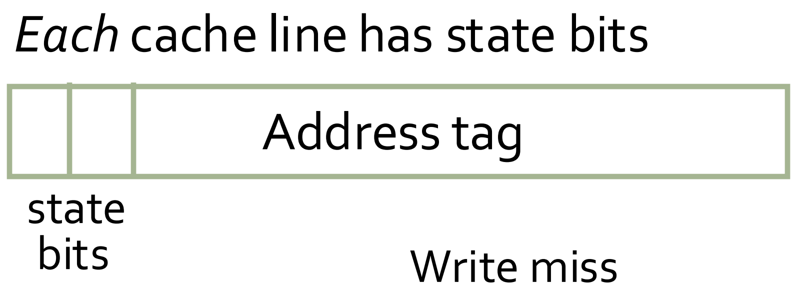
- Latency to access memory is different depending on node accessing it and address being accessed
- NUMA does not necessarily imply ccNUMA

Multicore Chips in a Multi-chip System



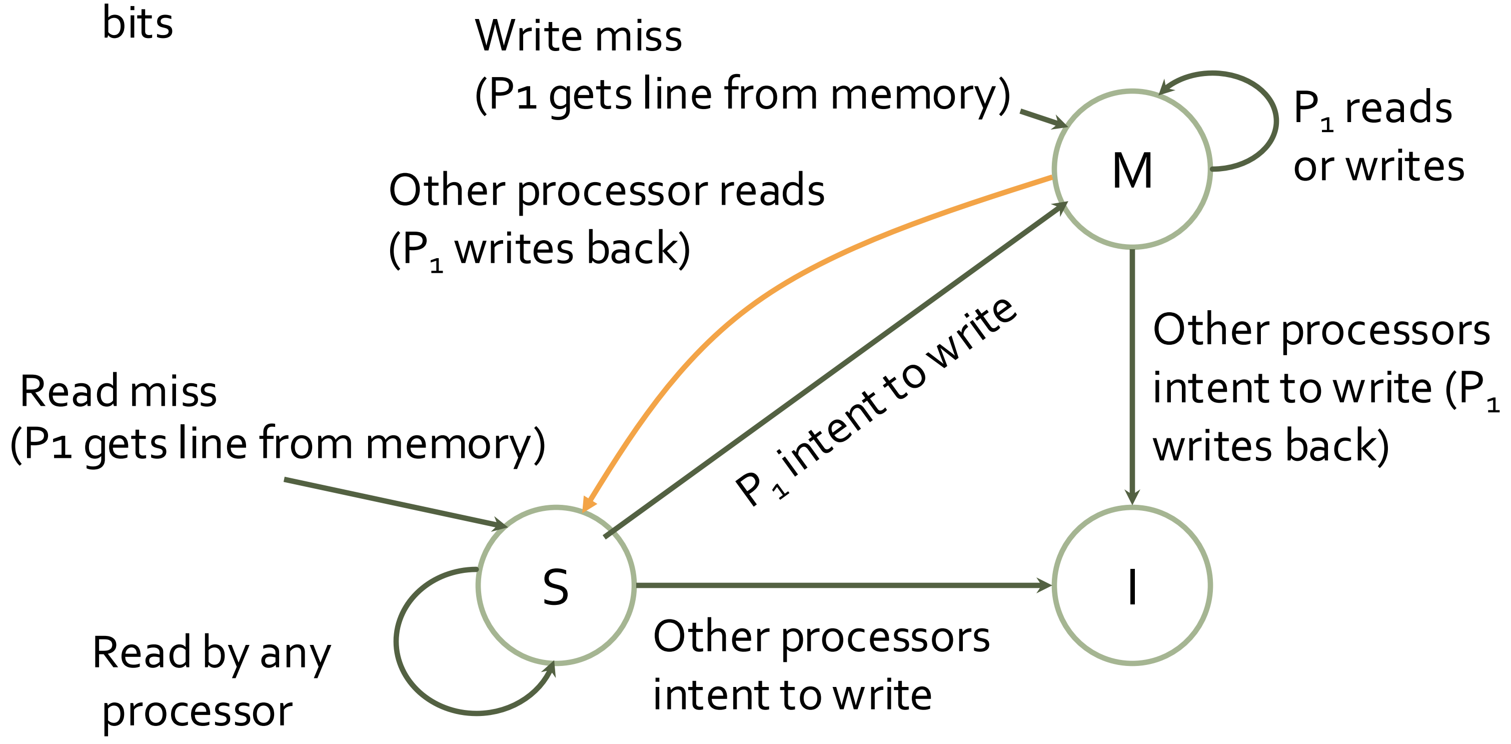
Cache State Transition Diagram

The MSI protocol



M: Modified
S: Shared
I: Invalid

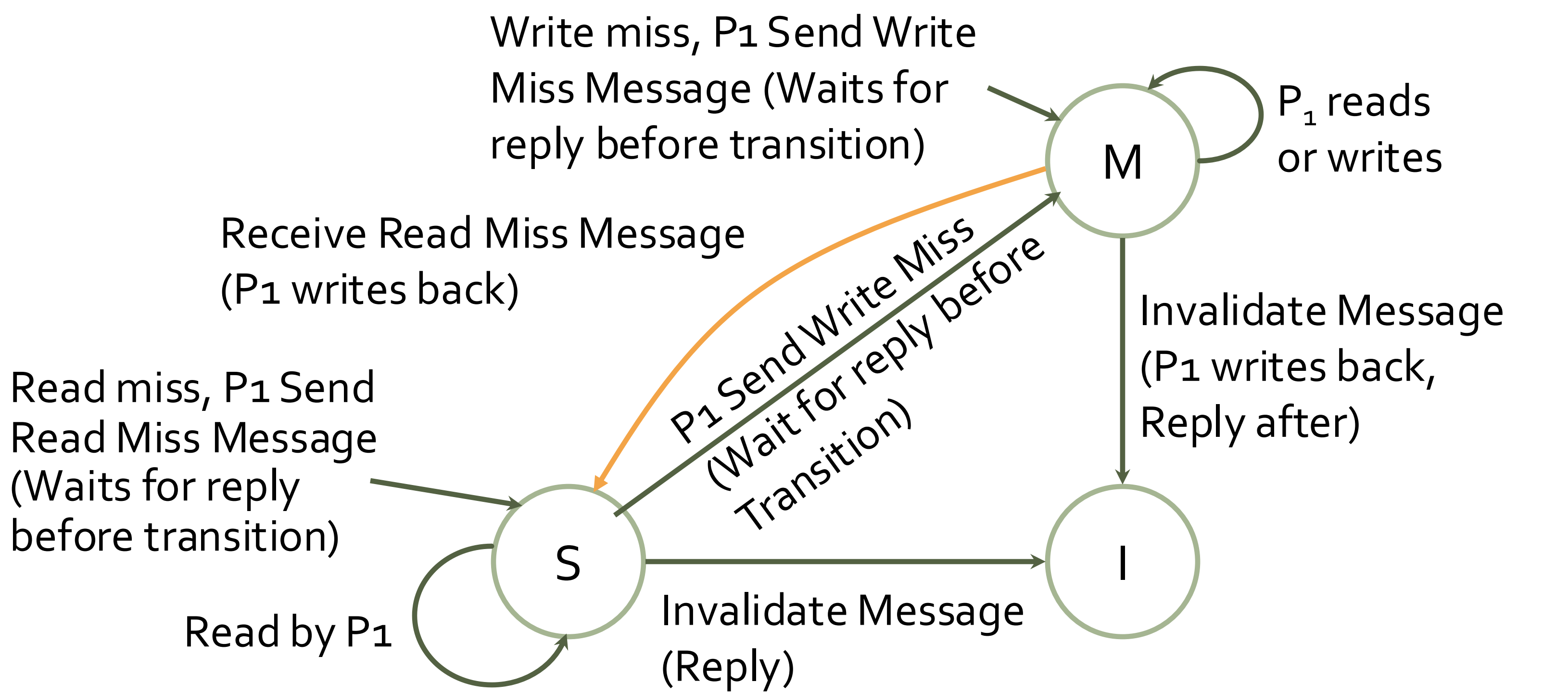
Cache state in processor P_1


Write miss (P1 gets line from memory)
P₁ reads or writes
M
Other processor reads (P₁ writes back)
P₁ intent to write
Other processors intent to write (P₁ writes back)
Read miss (P1 gets line from memory)
S
I
Read by any processor
Other processors intent to write

Cache State Transition Diagram

For Directory Coherence

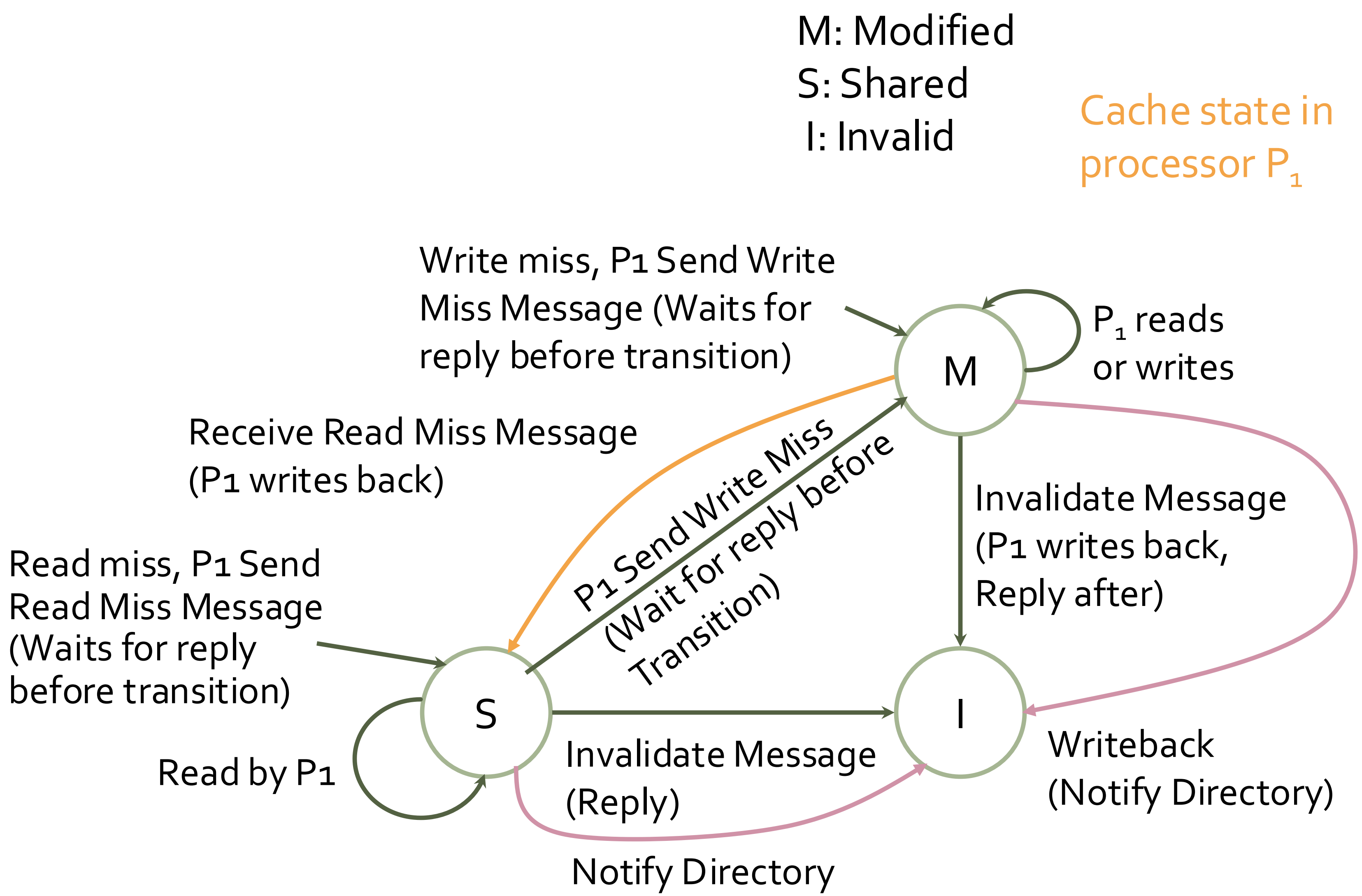
M: Modified
S: Shared
I: Invalid



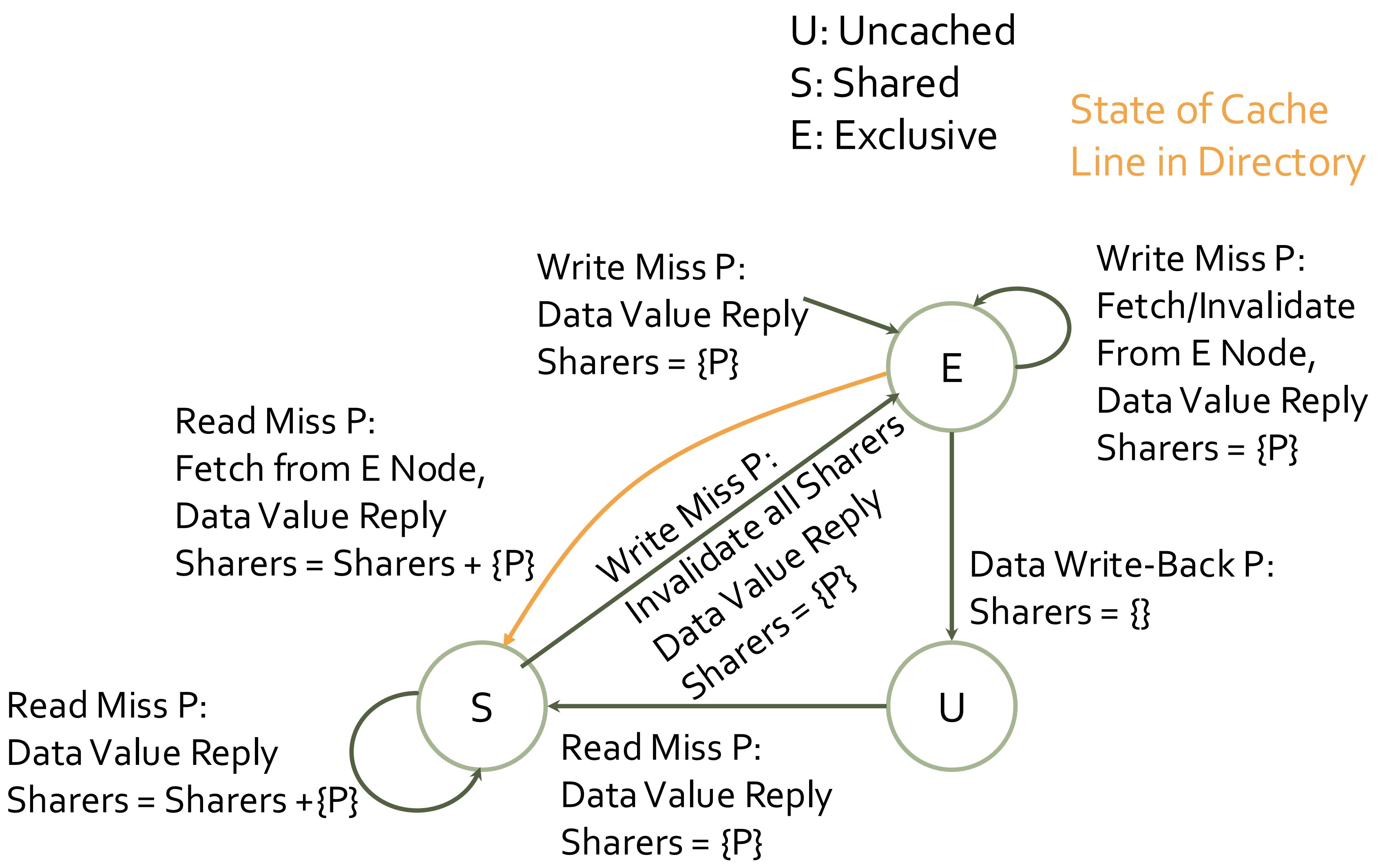
Cache state in processor P_1
Write miss, P1 Send Write Miss Message (Waits for reply before transition)
M
P_1 reads or writes
Receive Read Miss Message (P1 writes back)
P1 Send Write Miss (Wait for reply before Transition)
Invalidate Message (P1 writes back, Reply after)
Read miss, P1 Send Read Miss Message (Waits for reply before transition)
S
I
Read by P1
Invalidate Message (Reply)

Cache State Transition Diagram

For Directory Coherence


M: Modified
S: Shared
I: Invalid
Cache state in processor P_1
Write miss, P1 Send Write Miss Message (Waits for reply before transition)
M
P_1 reads or writes
Receive Read Miss Message (P1 writes back)
P1 Send Write Miss (Wait for reply before Transition)
Invalidate Message (P1 writes back, Reply after)
Read miss, P1 Send Read Miss Message (Waits for reply before transition)
S
I
Read by P1
Invalidate Message (Reply)
Writeback (Notify Directory)
Notify Directory

Directory State Transition Diagram



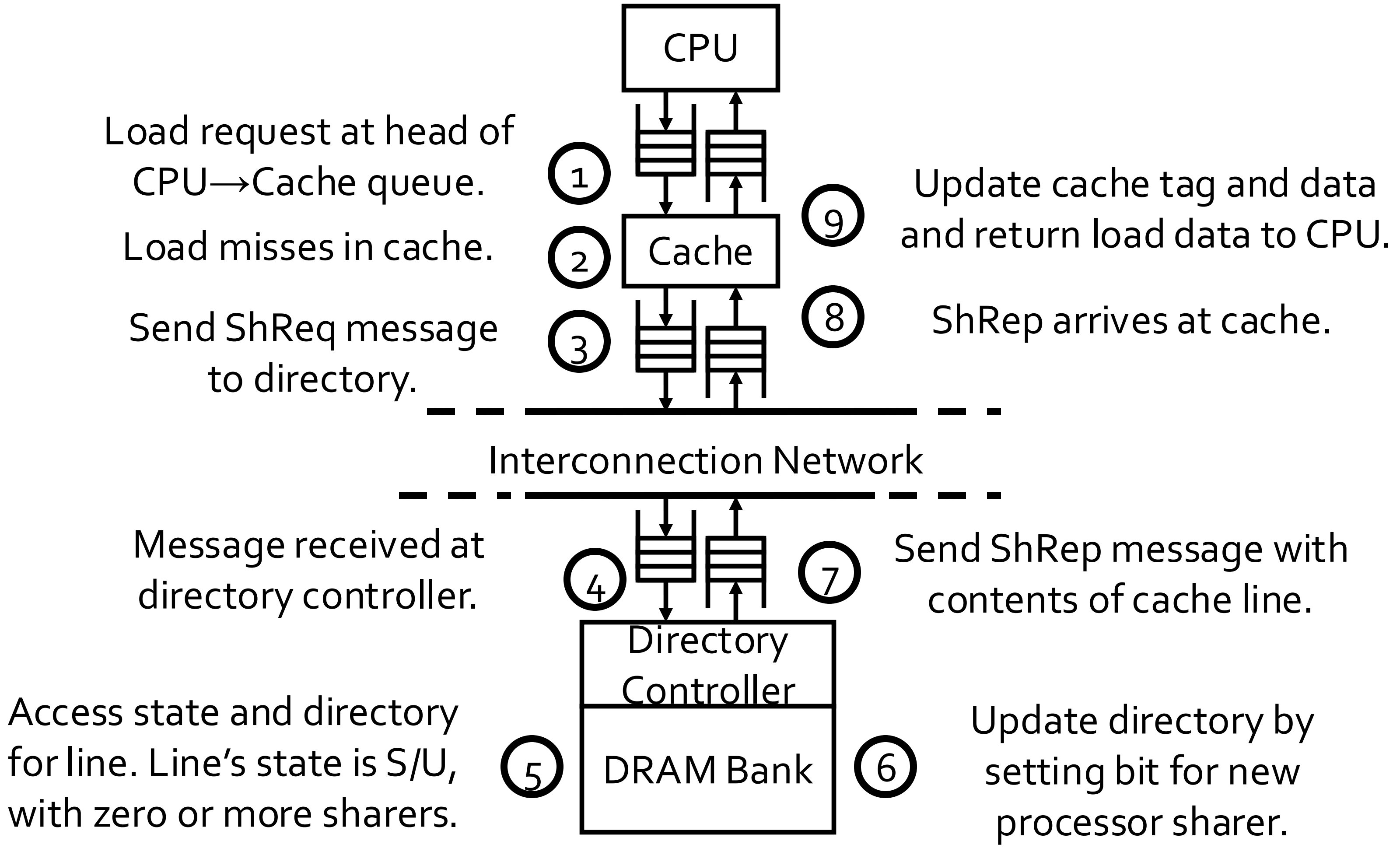
U: Uncached
S: Shared
E: Exclusive
State of Cache Line in Directory
Write Miss P: Data Value Reply Sharers = {P}
Write Miss P: Fetch/Invalidate From E Node, Data Value Reply Sharers = {P}
E
Read Miss P: Fetch from E Node, Data Value Reply Sharers = Sharers + {P}
Write Miss P: Invalidate all Sharers Data Value Reply Sharers = {P}
Data Write-Back P: Sharers = {}
S
U
Read Miss P: Data Value Reply Sharers = Sharers +{P}
Read Miss P: Data Value Reply Sharers = {P}

Directory Protocols

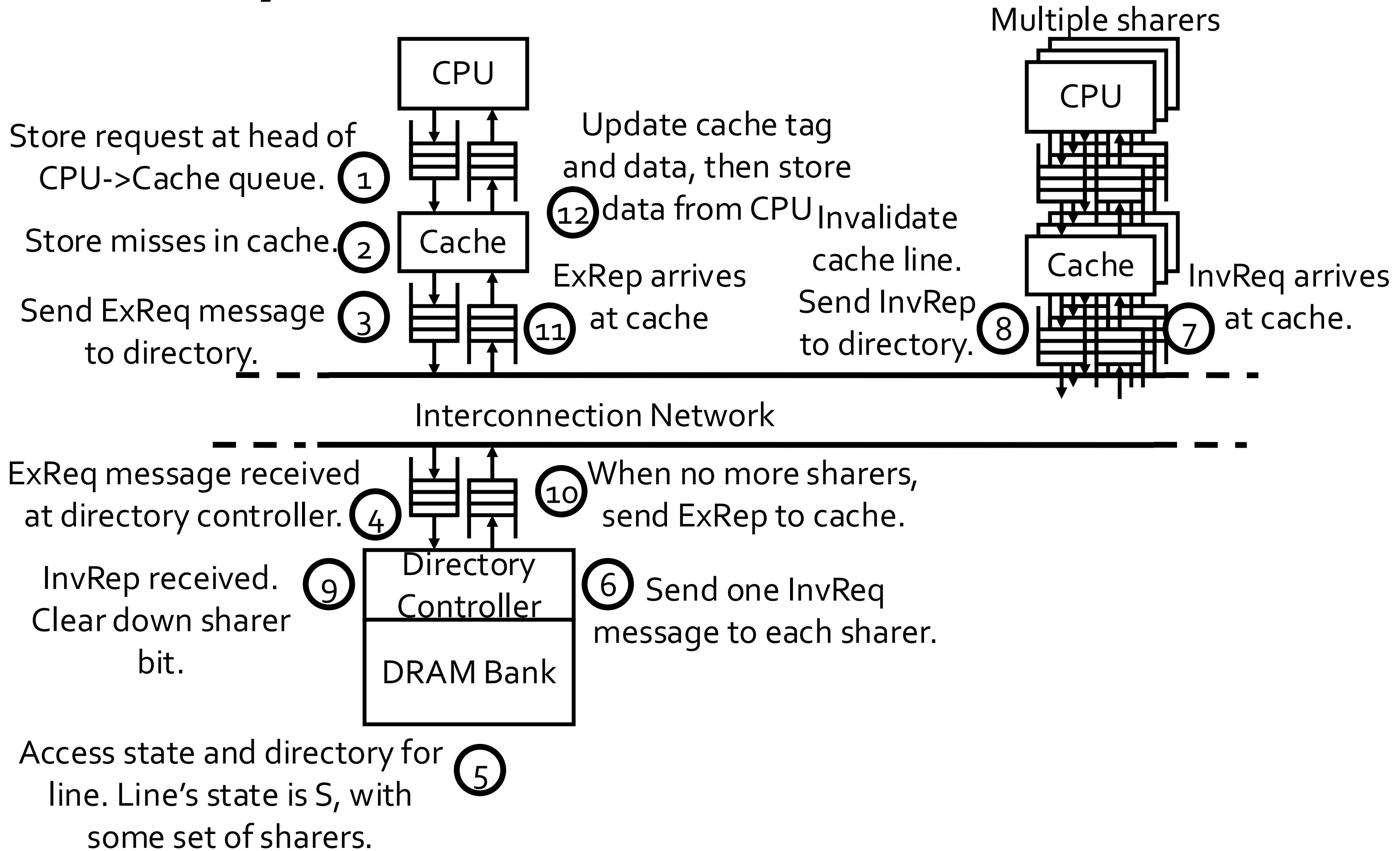
- For each block, maintain state:
  - Shared
    - One or more nodes have the block cached, value in memory is up-to-date
    - Set of node IDs
  - Uncached
  - Exclusive
    - Exactly one node has a copy of the cache block, value in memory is out-of-date
    - Owner node ID

- Directory maintains block states and sends invalidation messages

Read miss, to uncached or shared line

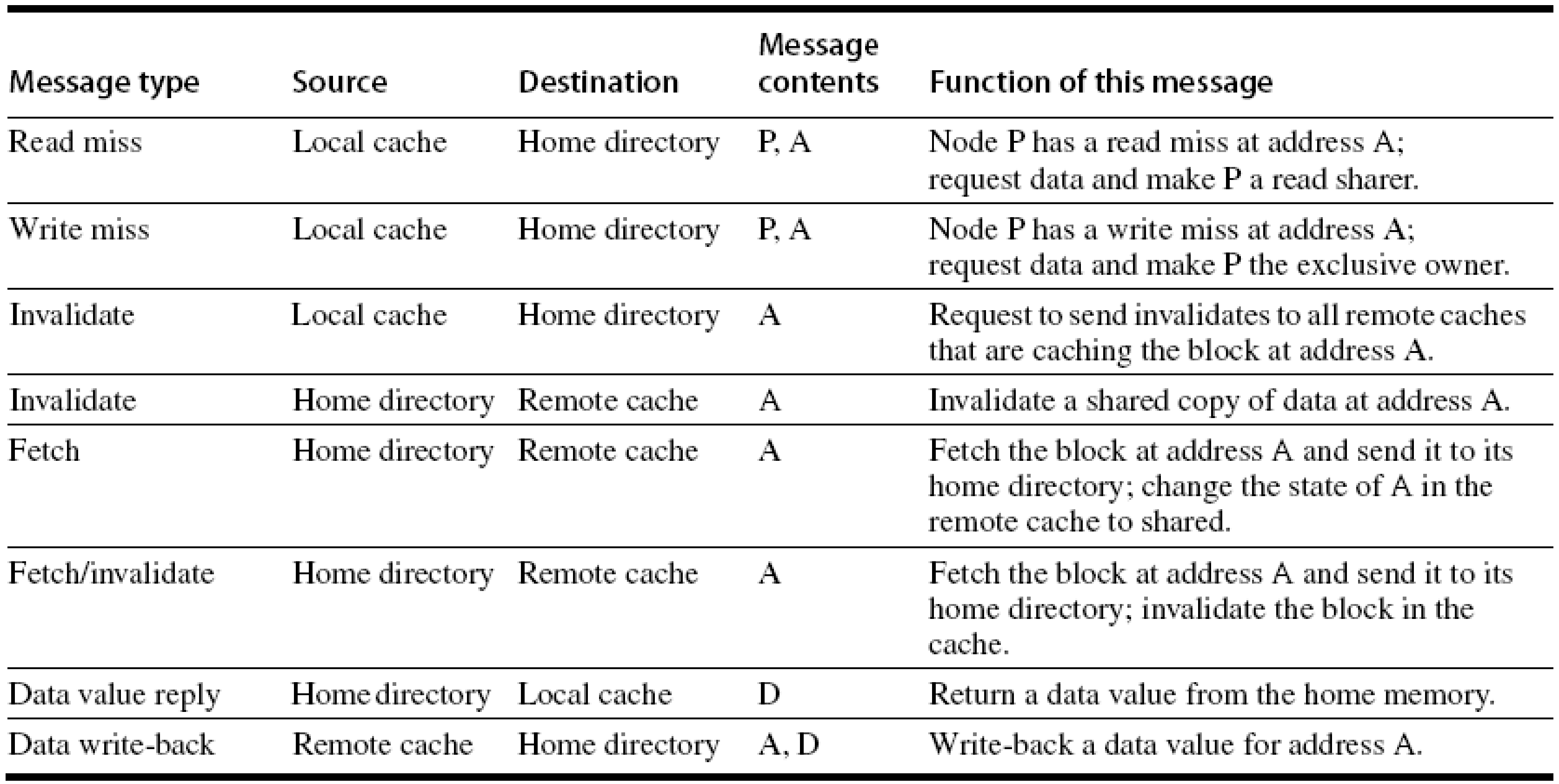

CPU
Load request at head of CPU→Cache queue.
1
Update cache tag and data and return load data to CPU.
9
Cache
Load misses in cache.
2
ShRep arrives at cache.
8
Send ShReq message to directory.
3
Interconnection Network
Message received at directory controller.
4
Send ShRep message with contents of cache line.
7
Directory Controller
Access state and directory for line. Line's state is S/U, with zero or more sharers.
5
DRAM Bank
6
Update directory by setting bit for new processor sharer.

Write miss, to read shared line



Multiple sharers
CPU
Cache
CPU
Cache
Store request at head of CPU->Cache queue. 1
Store misses in cache. 2
Send ExReq message to directory. 3
ExReq message received at directory controller. 4
Access state and directory for line. Line's state is S, with some set of sharers. 5
6 Send one InvReq message to each sharer.
7 InvReq arrives at cache.
Invalidate cache line. Send InvRep to directory. 8
InvRep received. Clear down sharer bit. 9
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
Interconnection Network
Directory Controller
DRAM Bank

Message Types



Message type	Source	Destination	Message contents	Function of this message
Read miss	Local cache	Home directory	P, A	Node P has a read miss at address A; request data and make P a read sharer.
Write miss	Local cache	Home directory	P, A	Node P has a write miss at address A; request data and make P the exclusive owner.
Invalidate	Local cache	Home directory	A	Request to send invalidates to all remote caches that are caching the block at address A.
Invalidate	Home directory	Remote cache	A	Invalidate a shared copy of data at address A.
Fetch	Home directory	Remote cache	A	Fetch the block at address A and send it to its home directory; change the state of A in the remote cache to shared.
Fetch/invalidate	Home directory	Remote cache	A	Fetch the block at address A and send it to its home directory; invalidate the block in the cache.
Data value reply	Home directory	Local cache	D	Return a data value from the home memory.
Data write-back	Remote cache	Home directory	A, D	Write-back a data value for address A.

Multiple Logical Communication Channels Needed

- **Avoid Deadlocks:** Prevent responses from being blocked by pending requests.
- **Manage Message Complexity:** Identify message types that generate additional traffic.
- **Use Separate Channels:** Assign different message types to distinct logical/physical channels.

Memory Ordering Point

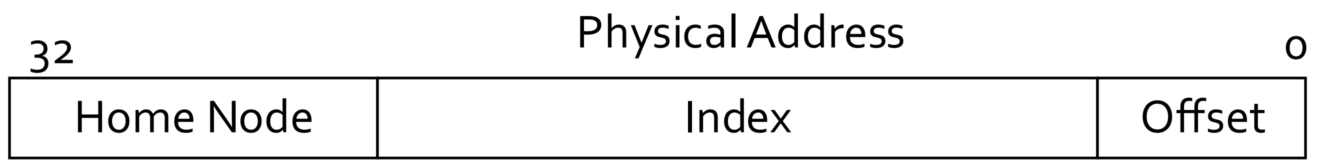
- Just like in bus based snooping protocol, need to guarantee that state transitions are atomic
- Directory used as ordering point (for specific addr)
    - Whichever message reaches home directory first wins
    - Other requests on same cache line given negative acknowledgement (NACK)
- NACK causes retry from other node
- Forward progress guarantee needed
    - After node acquires line, need to commit at least one memory operation before transitioning invalidating line

Organizing Directories

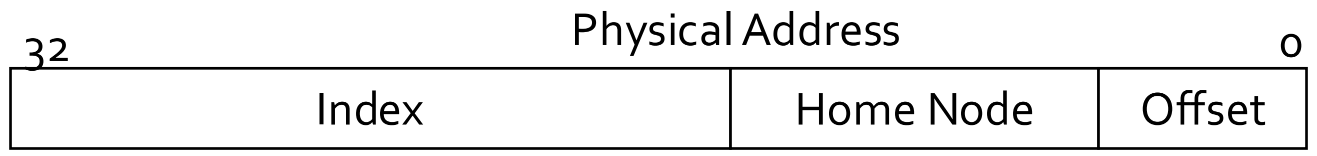
- Requirement: Directory needs to keep track of all the cores that are sharing a cache block
- Challenge: For each block, the space needed to hold the list of sharers grows with number of possible sharers…

Address to Home Directory

- High Order Bits Determine Home (Directory)



  - OS can control placement in NUMA
  - Homes can become hotspots
- Low Order Bits Determine Home (Directory)



  - OS loses control on placement
  - Load balanced well

Basic Full-Map Directory

State	Sharers/Owner
S	1011001100011
U	xxxxxxxxxxxxxxxx
U	xxxxxxxxxxxxxxxx
E	0001000000000
…	

State: State that the directory believes cache line is in {Shared, Uncached, Exclusive, (Pending)}

Sharers/Owner:
If State == Shared:
Bit vector of all of the nodes in system that have line in shared state.
If State == Modified:
Denotes which node has line in cache exclusively

Scalability of Directory Sharer List Storage

- Full-Map Directory (Bit per cache)
- Coarse-grain bit-vectors (e.g.,1 bit per 4 cores)
- Limited Pointer: Maintain a few sharer pointers, on overflow mark all and broadcast (or invalidate another sharer)
- LimitLess (Software assistance)

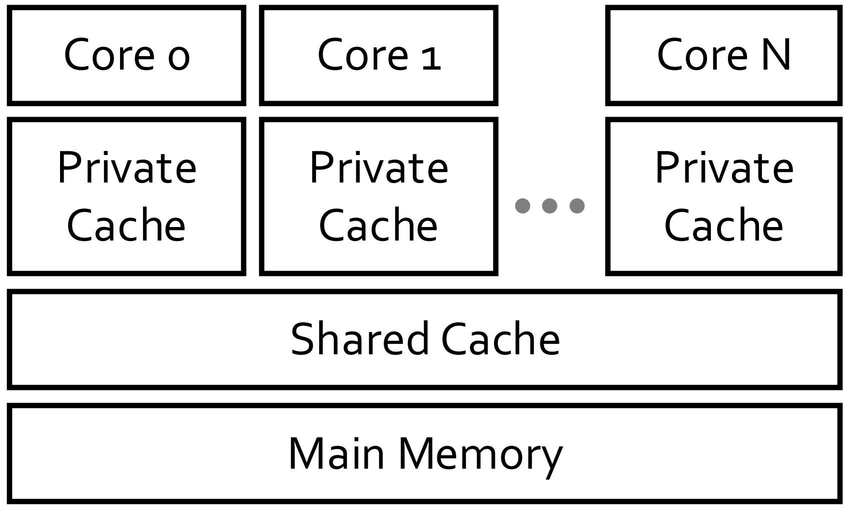
✓ Reduced area & energy

✗ Overheads still not scalable (these techniques simply play with constant factors)

✗ Inexact sharers → Broadcasts, invalidations or spurious invalidations and downgrades

In-Cache Directories

- Common multicore memory hierarchy:
  - 1+ levels of private caches
  - A shared last-level cache (LLC)
  - Cache coherence needed across private caches
- Idea: Embed the directory information in shared cache tags
  - Shared cache must be inclusive
  - Need extended directory if non-inclusive



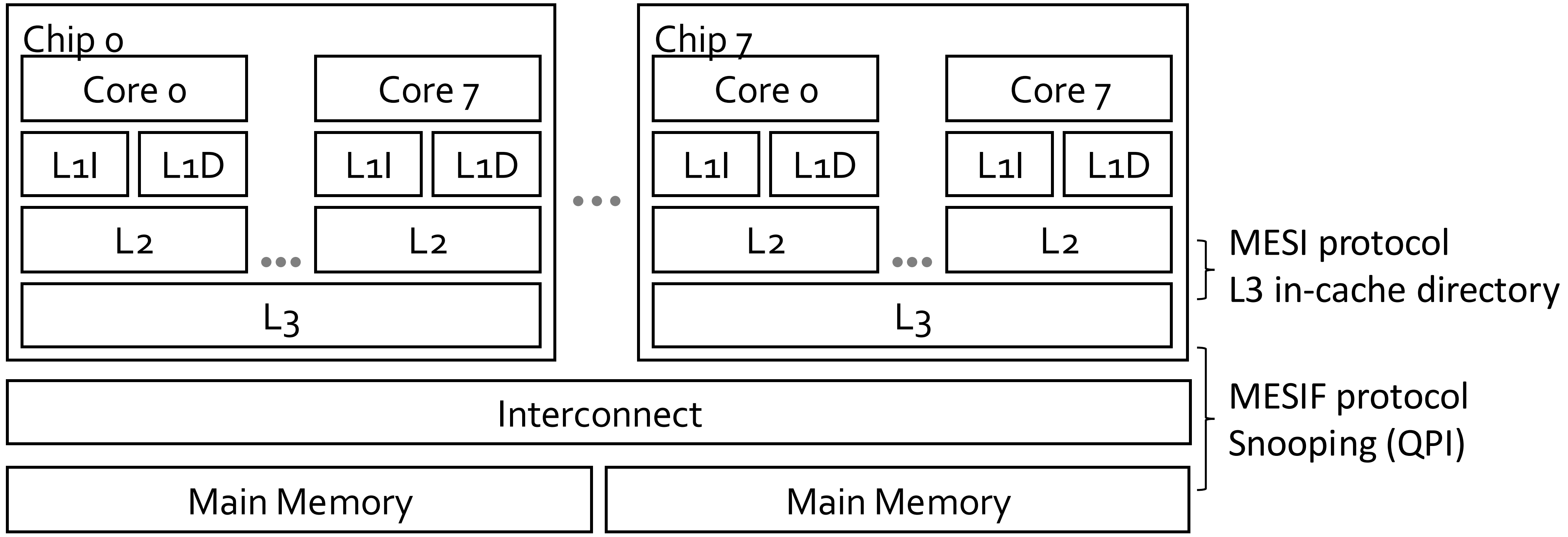
✓ Avoids tag overheads & separate lookups
✗ Inefficient if shared cache is much larger than private caches

Coherence in Multi-Level Hierarchies

- Can use the same or different protocols to keep coherence across multiple levels
- Key invariant: Ensure sufficient permissions in all intermediate levels

Coherence in Multi-Level Hierarchies

- Example: 8-socket Xeon E7 (8 cores/socket)

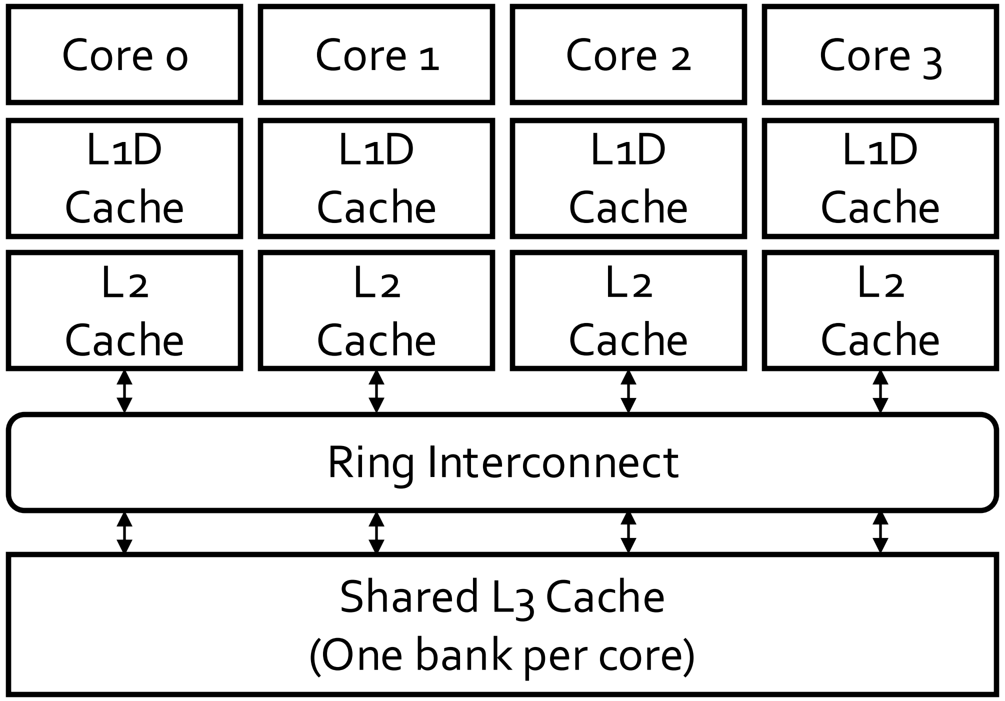


Directory coherence in Intel Core i7 CPU

- L3 hosts in-cache directory (and is inclusive)
- Directory maintains list of L2 caches containing line
- Instead of broadcasting coherence traffic to all L2's, only send coherence messages to L2's that contain the line

Core i7 interconnect is a ring, it is not a bus

- Directory dimensions:
  - P=4
  - M = number of L3 cache lines



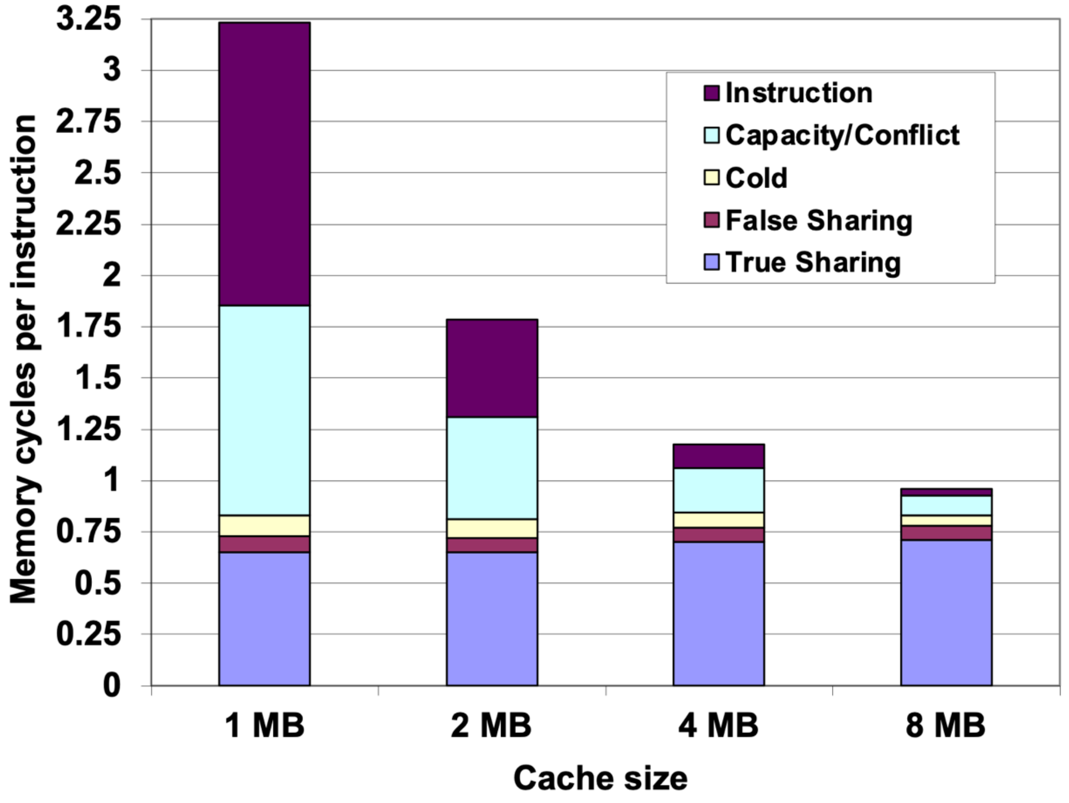
Performance of Symmetric Multiprocessors (SMPs)

- Cache performance is combination of:
  - Uniprocessor cache miss traffic
  - Traffic caused by communication
    - Results in invalidations and subsequent cache misses
- Adds 4th C: coherence miss
  - Joins Compulsory, Capacity, Conflict
  - (Sometimes called a Communication miss)

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

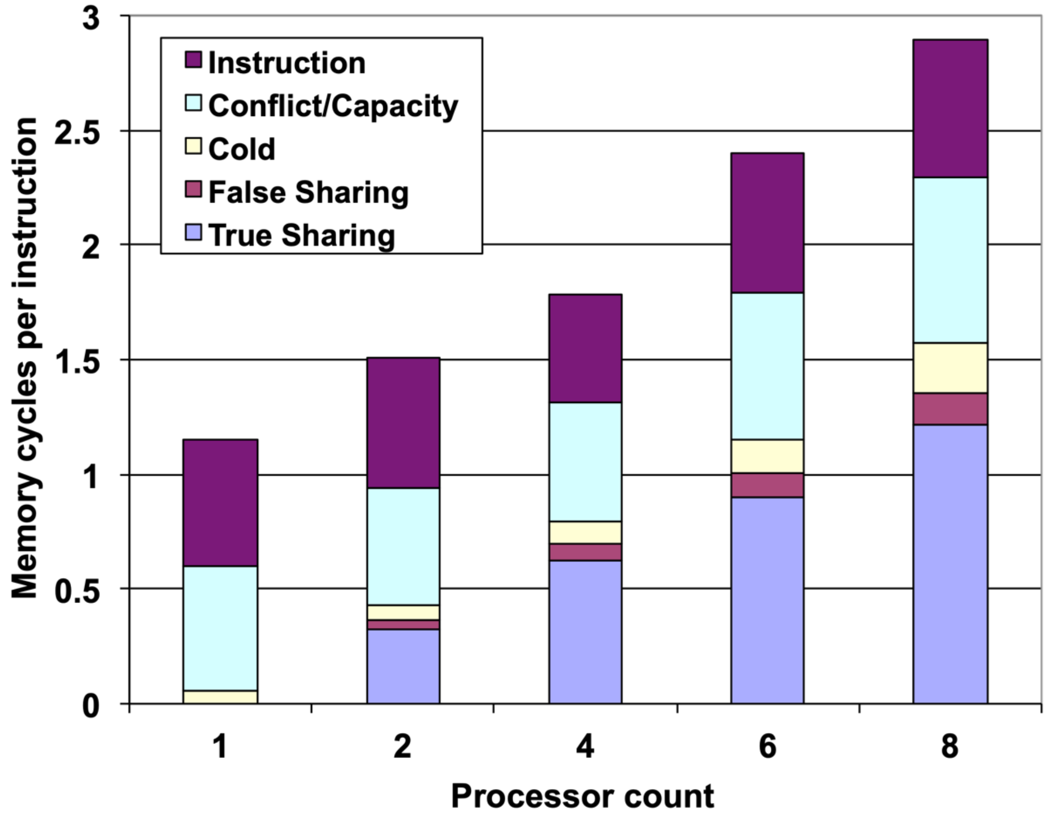
- True sharing and false sharing unchanged going from 1 MB to 8 MB (L3 cache)

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



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

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



Communication Overhead

- Communication time is key parallel overhead
    - Appears as increased memory latency in multiprocessor
        - Extra main memory accesses in UMA systems
            - Must determine lowering of cache miss rate vs. uniprocessor
    - Some accesses have higher latency in NUMA systems
        - Only a fraction of a% of these can be significant

False Sharing

Performance Issue #1

state	blk addr	data0	data1	…	dataN

- A cache block contains more than one word
- Cache-coherence is done at the block-level and not word-level
- Suppose P_1 writes $word_i$ and P_2 writes $word_k$ and both words have the same block address.

- What can happen?
  - Cache line ping-pongs between caches of writing processors, generating significant amounts of communication due to the coherence protocol
  - No inherent communication, this is entirely artifactual communication
  - False sharing can be a factor in when programming for cache-coherent architectures

Coherency Misses

1. 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
2. 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
     $\Rightarrow$ miss would not occur if block 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

Demo: False Sharing

```
void* worker(void* arg) {
   volatile int* counter = (int*)arg;
   for (int i = 0; i < MANY_ITERATIONS; i++)
      (*counter)++;
   return NULL;
}

void test1(int num_threads) {
   pthread_t threads[MAX_THREADS];
   int counter[MAX_THREADS] = {0};
   for (int i = 0; i < num_threads; i++)
      pthread_create(&threads[i], NULL, worker, &counter[i]);
   for (int i = 0; i < num_threads; i++)
      pthread_join(threads[i], NULL);
}
```

6.863 sec with 48 threads on 24 core machine

```
typedef struct {
   int counter;
   char padding[CACHE_LINE_SIZE - sizeof(int)];
} padded_t;


void test2(int num_threads) {
   pthread_t threads[MAX_THREADS];
   padded_t counter[MAX_THREADS];

   for (int i = 0; i < num_threads; i++)
      pthread_create(&threads[i], NULL, worker, &counter[i].counter);

   for (int i = 0; i < num_threads; i++)
      pthread_join(threads[i], NULL);
}
```

0.115 sec with 48 threads on 24 core machine

Coherence and Synchronization

Performance Issue #2

- Cache coherence protocols may cause mutex to ping-pong between caches.
- Ping-ponging can be reduced by first checking the lock value with an ordinary read and performing the expensive atomic operation only when the lock appears available.

Coherence and Bus Occupancy

Performance Issue #3

- In general, an atomic read-modify-write instruction requires two memory (bus) operations without intervening memory operations by other processors
- In a multiprocessor setting, bus needs to be locked for the entire duration of the atomic read and write operation
  - expensive for simple buses
  - very expensive for split-transaction buses
- modern processors use
  - load-reserve
  - store-conditional

Recap

- Cache Coherence

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)
  - Christopher Batten (Cornell)
  - David Wentzlaff (Princeton)
- MIT material derived from course 6.823
- UCB material derived from course CS252
- Cornell material derived from course ECE 4750
- Princeton material derived from course ECE 475