Directory-Based Cache Coherence

Daniel Sanchez
Computer Science and Artificial Intelligence Lab
M.I.T.

Maintaining Cache Coherence

It is sufficient to have hardware such that

- only one processor at a time has write permission for a location
- no processor can load a stale copy of the location after a write

⇒ A correct approach could be:

write request:

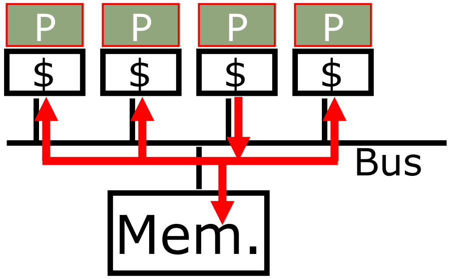
The address is *invalidated* in all other caches *before* the write is performed

read request:

If a dirty copy is found in some cache, a write-back is performed before the memory is read

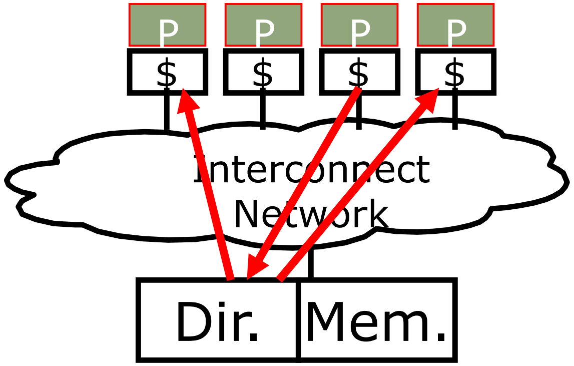
Directory-Based Coherence
(Censier and Feautrier, 1978)

Snoopy Protocols



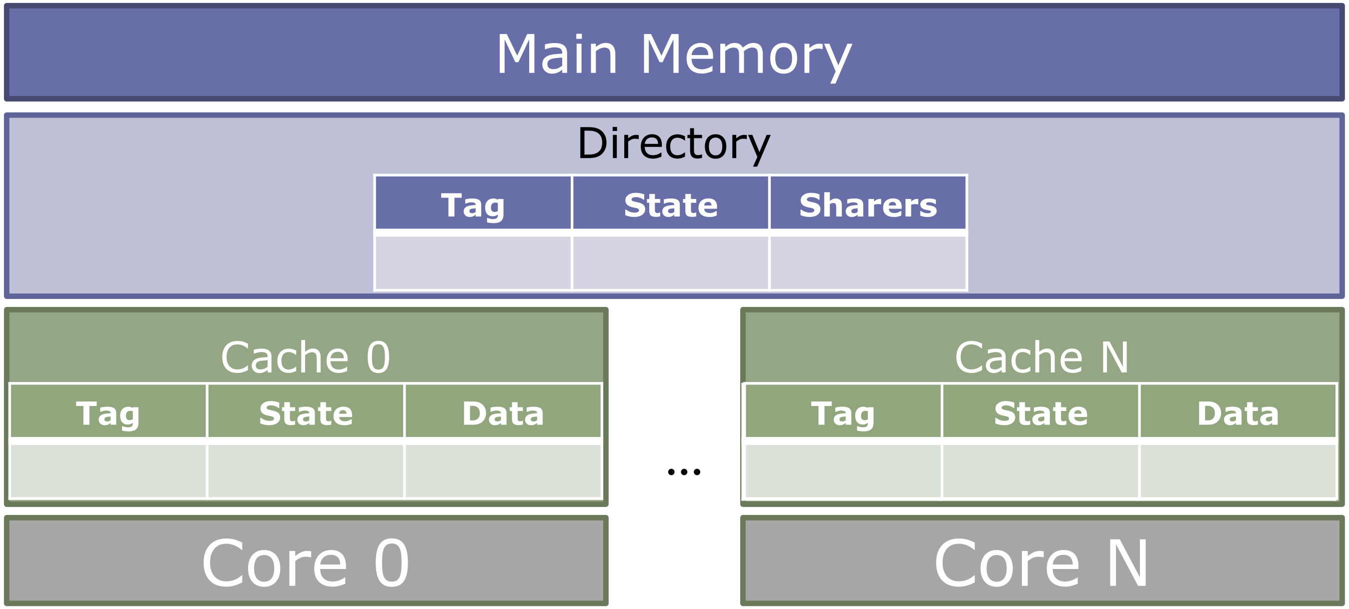
- Snoopy schemes broadcast requests over memory bus
- Difficult to scale to large numbers of processors
- Requires additional bandwidth to cache tags for snoop requests

Directory Protocols



- Directory schemes send messages to only those caches that might have the line
- Can scale to large numbers of processors
- Requires extra directory storage to track possible sharers

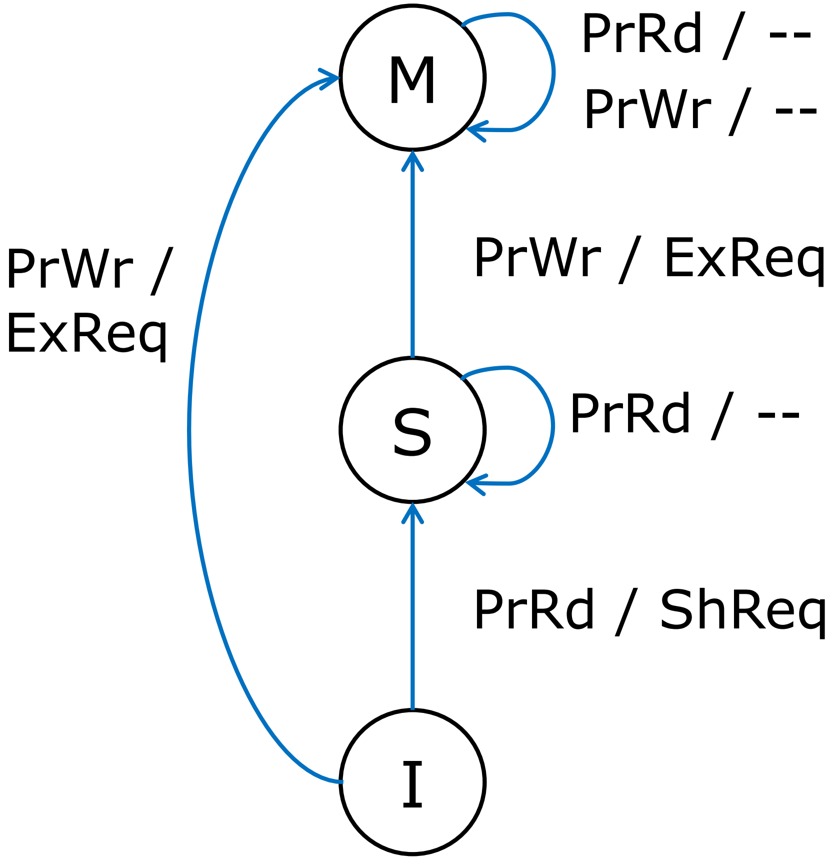
An MSI Directory Protocol



- Cache states: Modified (M) / Shared (S) / Invalid (I)
- Directory states:
  - Uncached (Un): No sharers
  - Shared (Sh): One or more sharers with read permission (S)
  - Exclusive (Ex): A single sharer with read & write permissions (M)
- Transient states not drawn for clarity; for now, assume no racing requests

MSI Protocol: Caches (1/3)

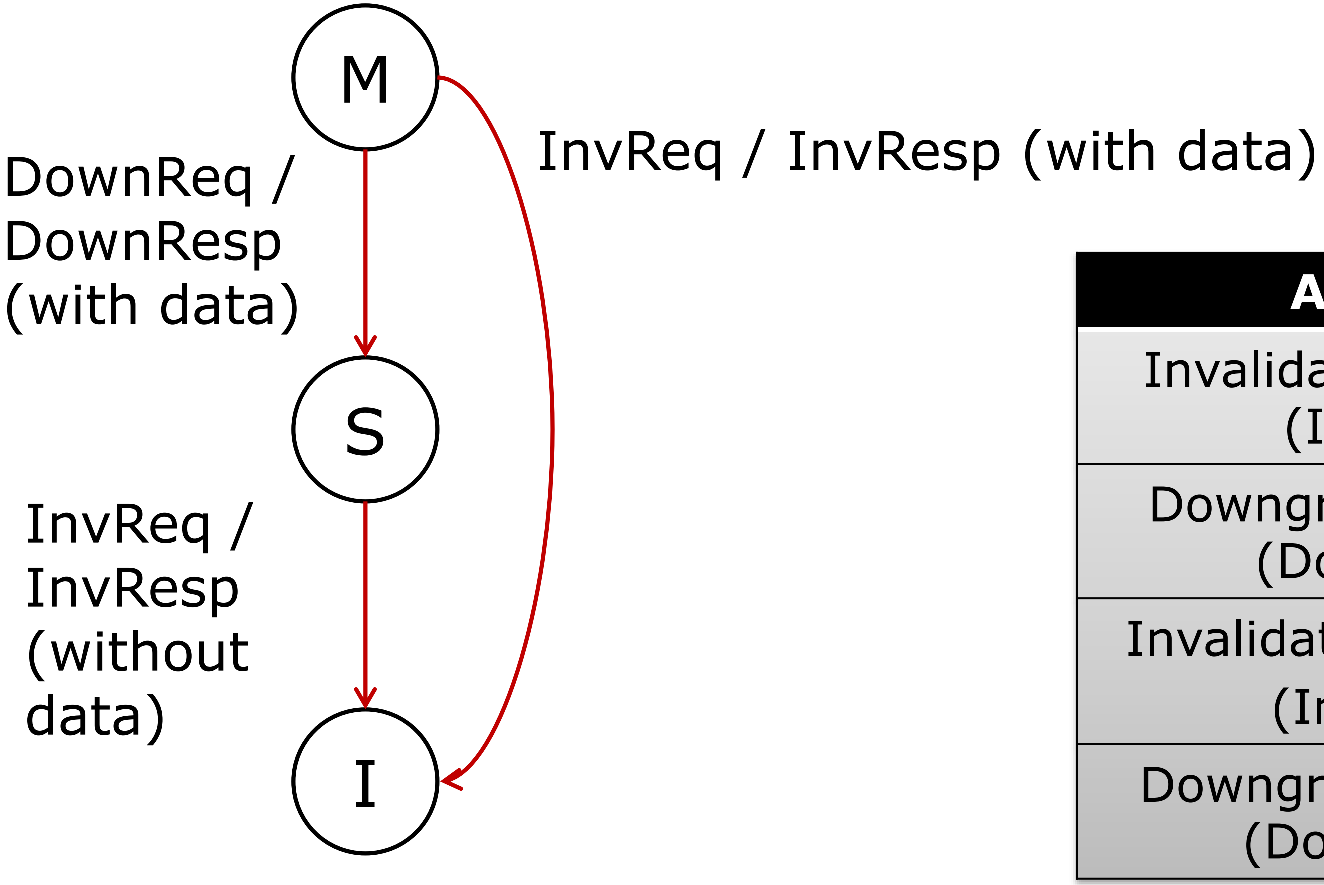
Transitions initiated by processor accesses:



Actions
Processor Read (PrRd)
Processor Write (PrWr)
Shared Request (ShReq)
Exclusive Request (ExReq)

MSI Protocol: Caches (2/3)

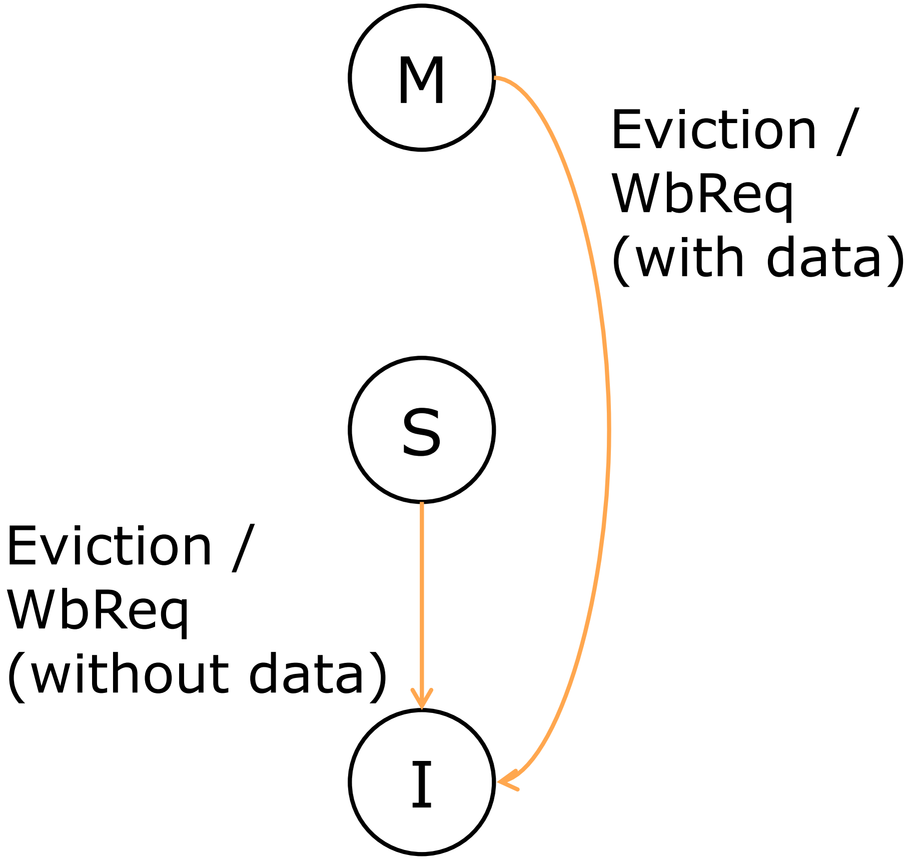
Transitions initiated by directory requests:


M
S
I
InvReq / InvResp (with data)
DownReq / DownResp (with data)
InvReq / InvResp (without data)

Actions
Invalidation Request (InvReq)
Downgrade Request (DownReq)
Invalidation Response (InvResp)
Downgrade Response (DownResp)

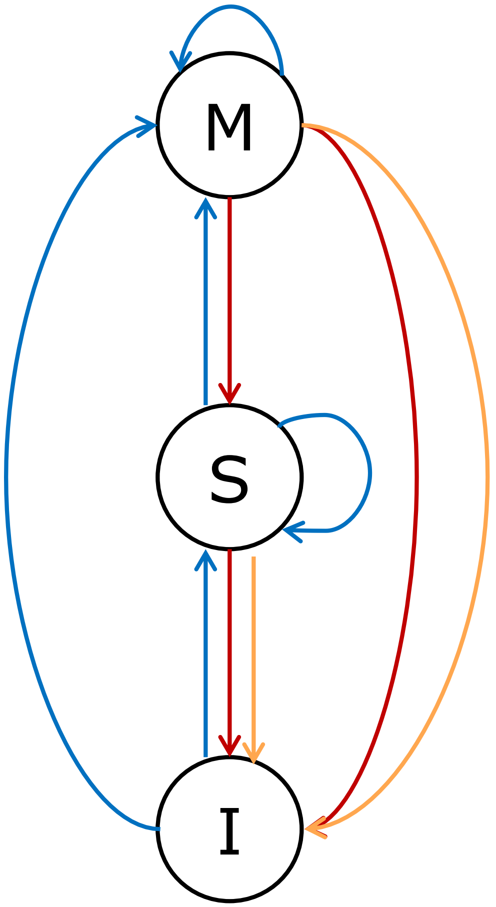
MSI Protocol: Caches (3/3)

Transitions initiated by evictions:



Actions
Writeback Request (WbReq)

MSI Protocol: Caches



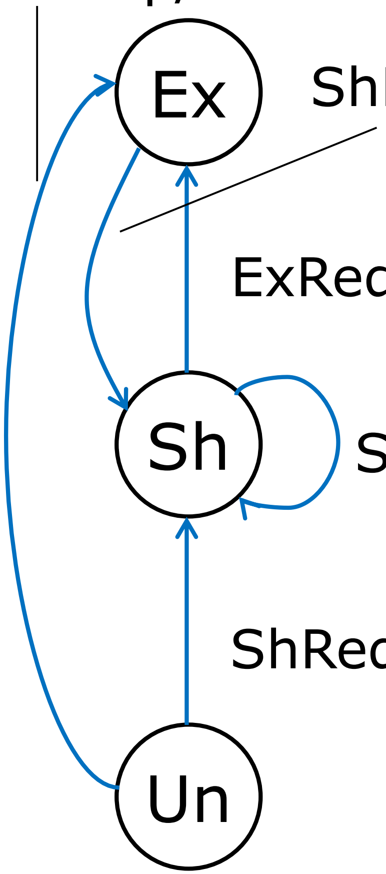
→ Transitions initiated by processor accesses

→ Transitions initiated by directory requests

→ Transitions initiated by evictions

MSI Protocol: Directory (1/2)

Transitions initiated by data requests:



ExReq / Sharers = {P}; ExResp

ShReq / Down(Sharer); Sharers = Sharer + {P}; ShResp

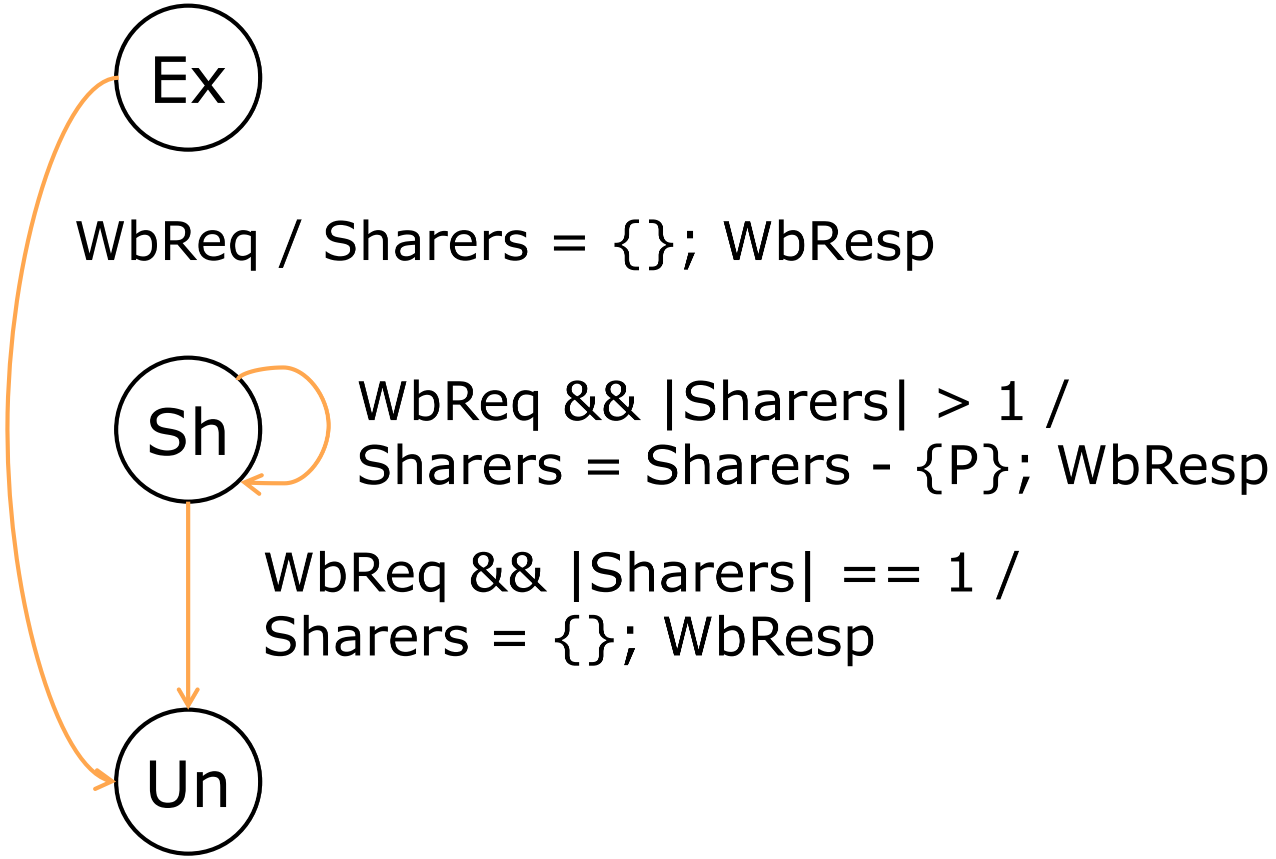
ExReq / Inv(Sharers – {P}); Sharers = {P}; ExResp

ShReq / Sharers = Sharers + {P}; ShResp

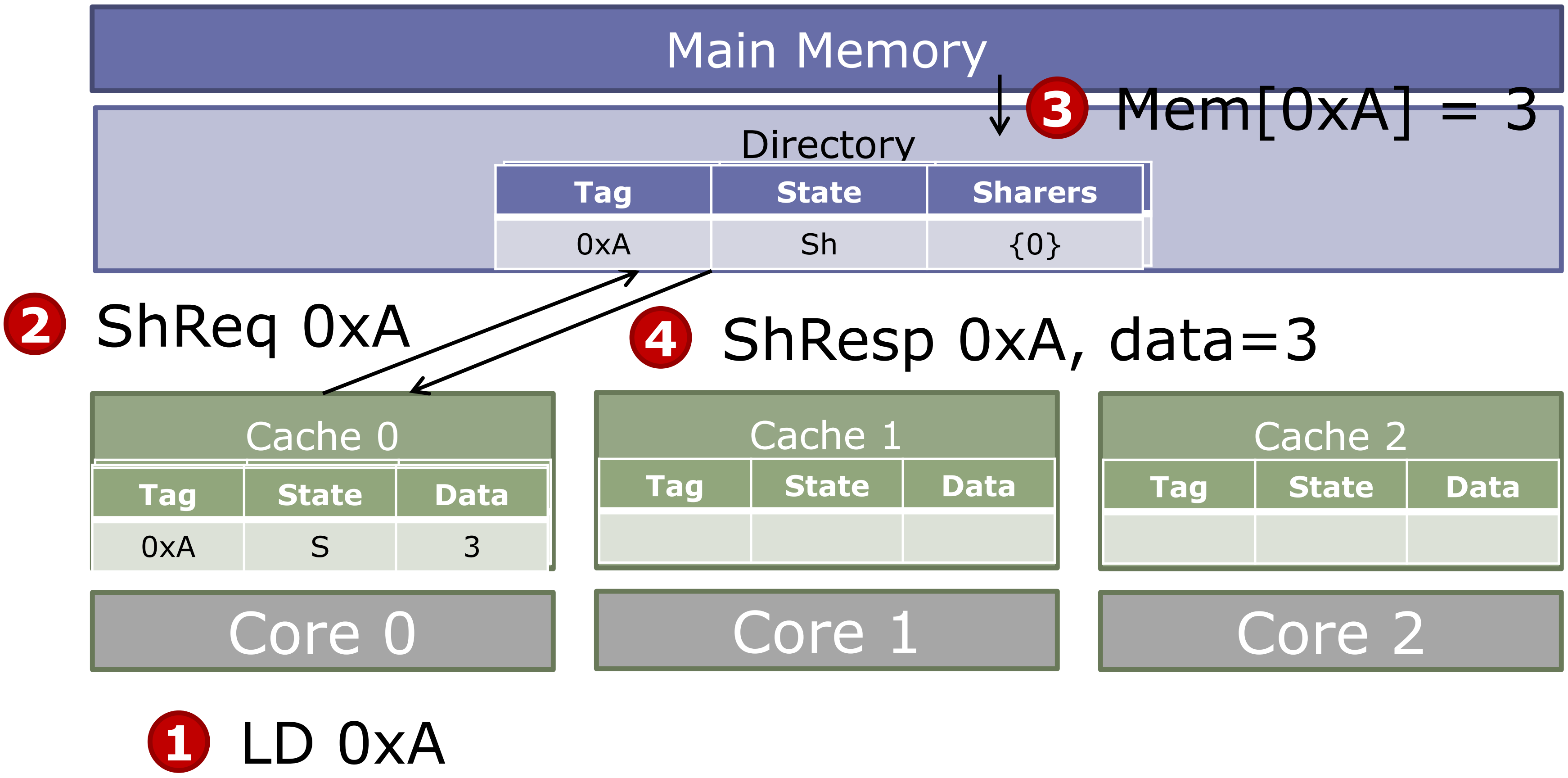
ShReq / Sharers = {P}; ShResp

MSI Protocol: Directory (2/2)

Transitions initiated by writeback requests:


Ex
WbReq / Sharers = {}; WbResp
Sh
WbReq && |Sharers| > 1 /
Sharers = Sharers - {P}; WbResp
WbReq && |Sharers| == 1 /
Sharers = {}; WbResp
Un

MSI Directory Protocol Example



Main Memory

3 Mem[0xA] = 3

Directory

Tag	State	Sharers
0xA	Sh	{0}

2 ShReq 0xA

4 ShResp 0xA, data=3

Cache 0

Tag	State	Data
0xA	S	3

Cache 1

Tag	State	Data

Cache 2

Tag	State	Data

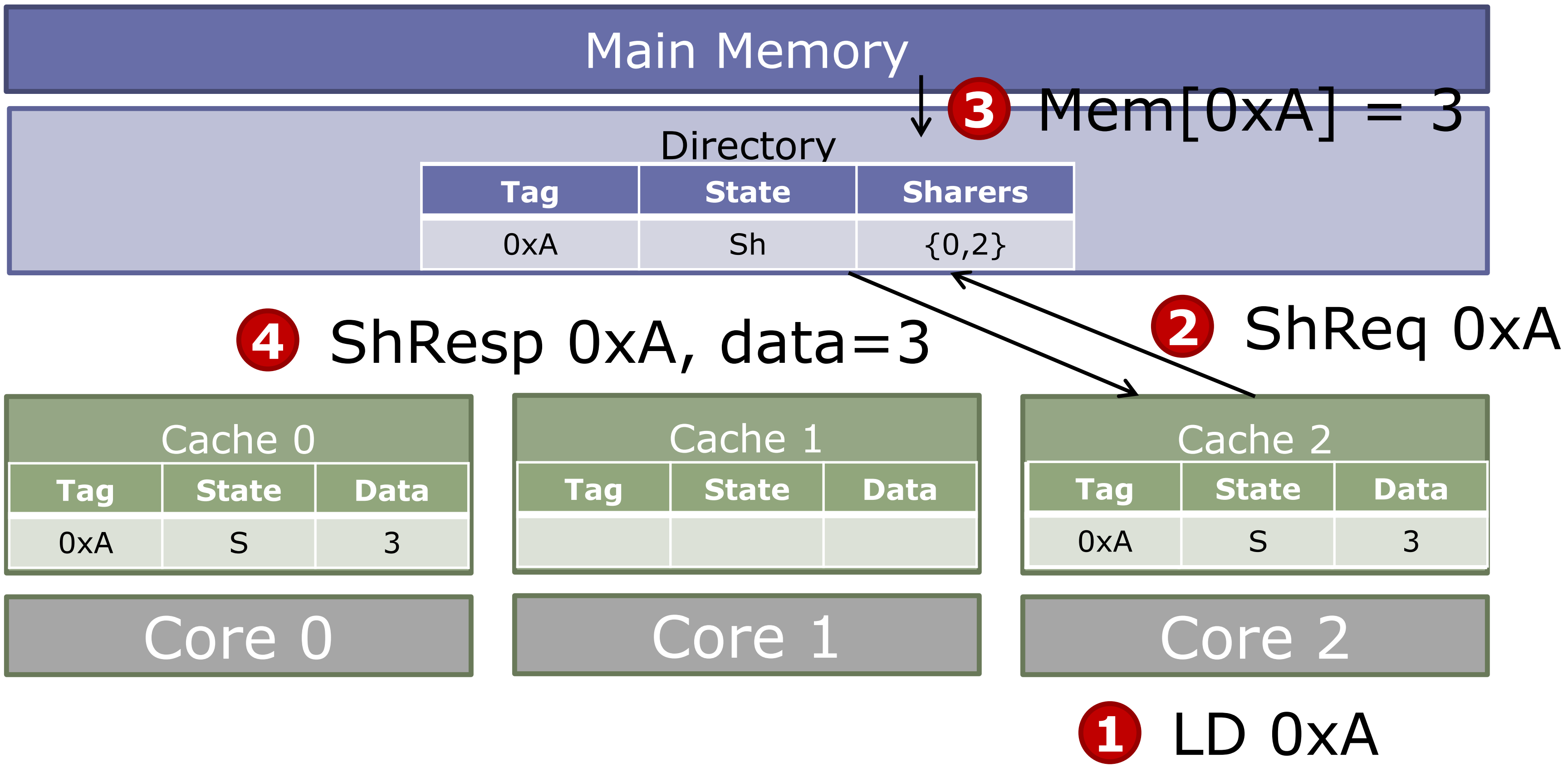
Core 0

Core 1

Core 2

1 LD 0xA

MSI Directory Protocol Example



Main Memory

3 Mem[0xA] = 3

Directory

Tag	State	Sharers
0xA	Sh	{0,2}

4 ShResp 0xA, data=3

2 ShReq 0xA

Cache 0

Tag	State	Data
0xA	S	3

Cache 1

Tag	State	Data

Cache 2

Tag	State	Data
0xA	S	3

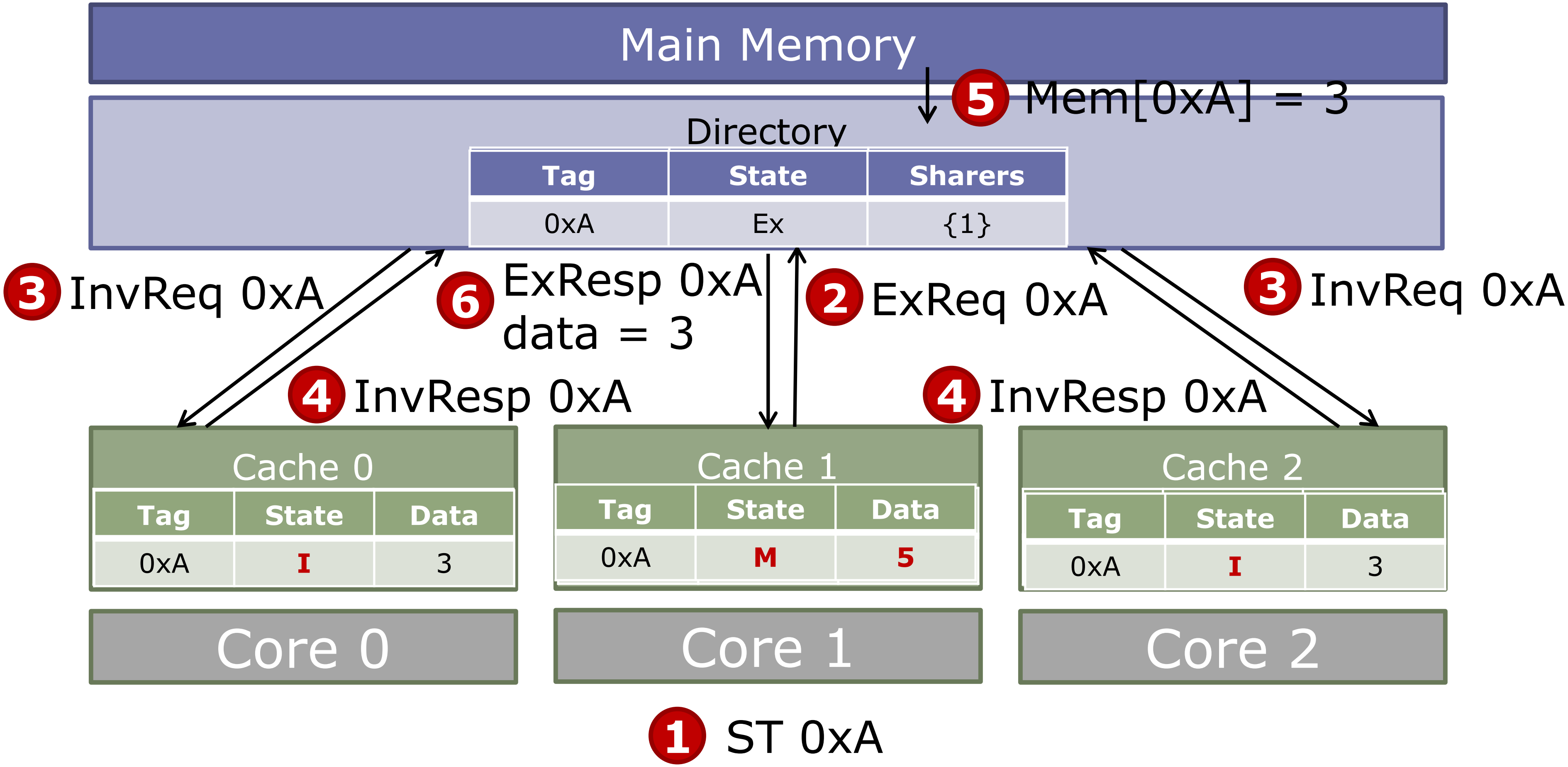
Core 0

Core 1

Core 2

1 LD 0xA

MSI Directory Protocol Example



Directory

Tag	State	Sharers
0xA	Ex	{1}

Cache 0

Tag	State	Data
0xA	I	3

Cache 1

Tag	State	Data
0xA	M	5

Cache 2

Tag	State	Data
0xA	I	3

1 ST 0xA

2 ExReq 0xA

3 InvReq 0xA

3 InvReq 0xA

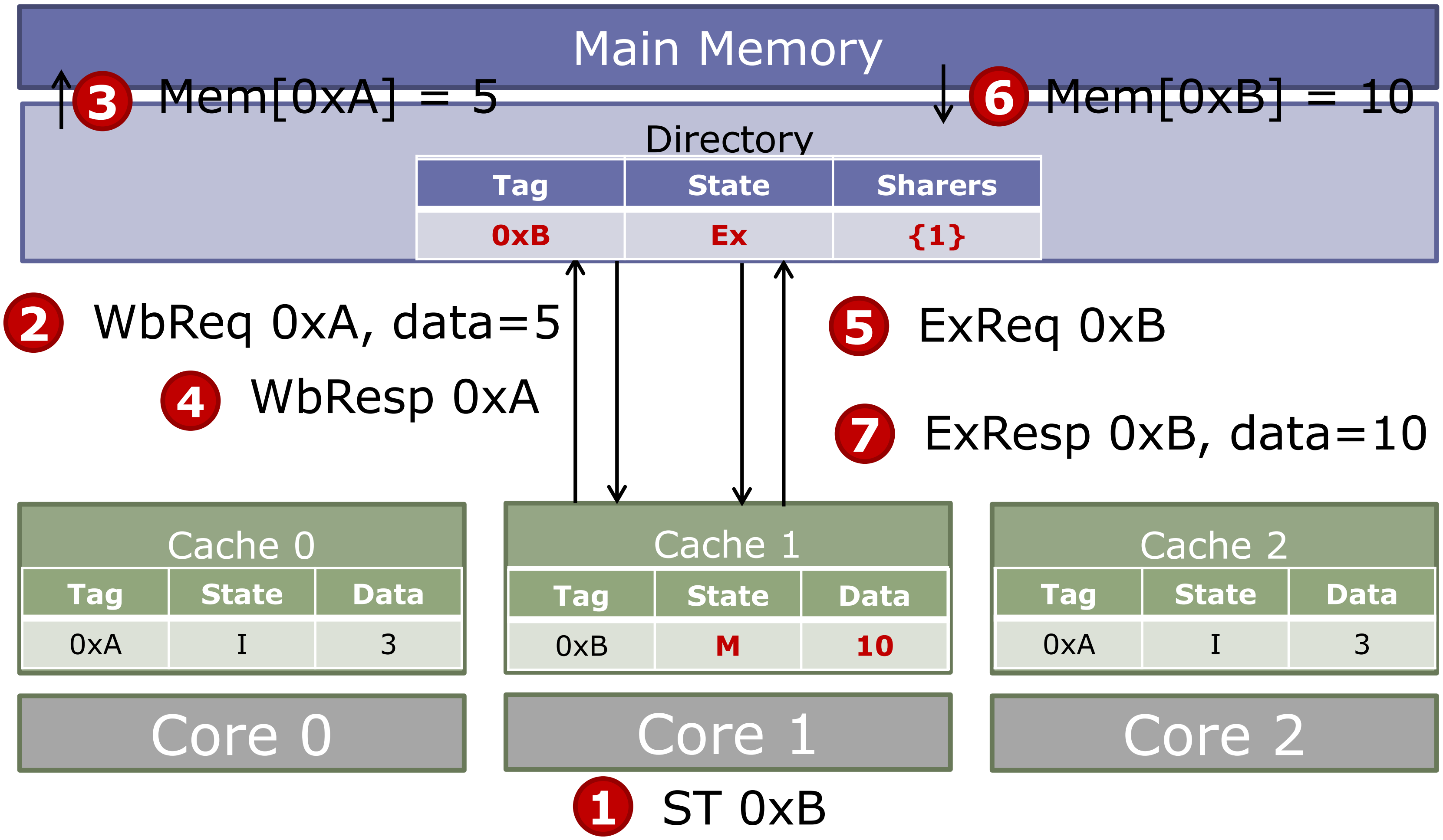
4 InvResp 0xA

4 InvResp 0xA

5 Mem[0xA] = 3

6 ExResp 0xA
data = 3

MSI Directory Protocol Example



Main Memory

3 Mem[0xA] = 5

6 Mem[0xB] = 10

Directory

Tag	State	Sharers
0xB	**Ex**	**{1}**

2 WbReq 0xA, data=5

4 WbResp 0xA

5 ExReq 0xB

7 ExResp 0xB, data=10

Cache 0

Tag	State	Data
0xA	I	3

Cache 1

Tag	State	Data
0xB	**M**	**10**

Cache 2

Tag	State	Data
0xA	I	3

Core 0

Core 1

Core 2

1 ST 0xB

Why are 0xA's wb and 0xB's req serialized?

Possible solutions?

Miss Status Handling Register

MSHR – Holds load misses and writes outside of cache

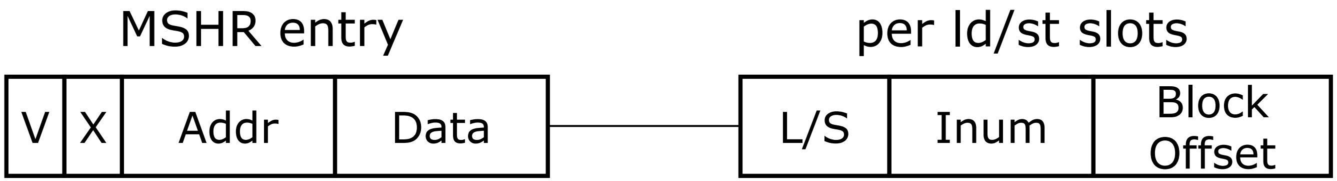
MSHR entry

V	X	Addr	Data

- On eviction/writeback
  - No free MSHR entry: stall
  - Allocate new MSHR entry
  - When channel available send WBReq and data
  - Deallocate entry on WBResp

Miss Status Handling Register

MSHR – Holds load misses and writes outside of cache



MSHR entry

V | X | Addr | Data

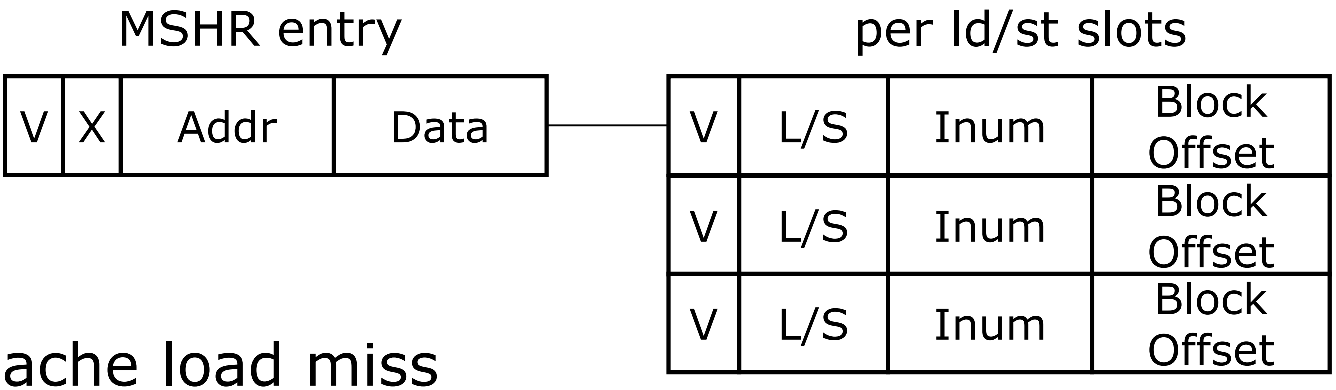
per ld/st slots

L/S | Inum | Block Offset

- On cache load miss
  - No free MSHR entry: stall
  - Allocate new MSHR entry
  - Send ShReq (or ExReq)
  - On *Resp forward data to CPU and cache
  - Deallocate MSHR

Miss Status Handling Register

MSHR – Holds load misses and writes outside of cache



MSHR entry

V	X	Addr	Data

per ld/st slots

V	L/S	Inum	Block Offset
V	L/S	Inum	Block Offset
V	L/S	Inum	Block Offset

- On cache load miss
  - Look for matching address is MSHR
    - If not found
      - If no free MSHR entry: stall
      - Allocate new MSHR entry and fill in
    - If found, just fill in per ld/st slot
  - Send ShReq (or ExReq)
  - On *Resp forward data to CPU and cache
  - Deallocate MSHR

Per ld/st slots allow servicing multiple requests with one entry

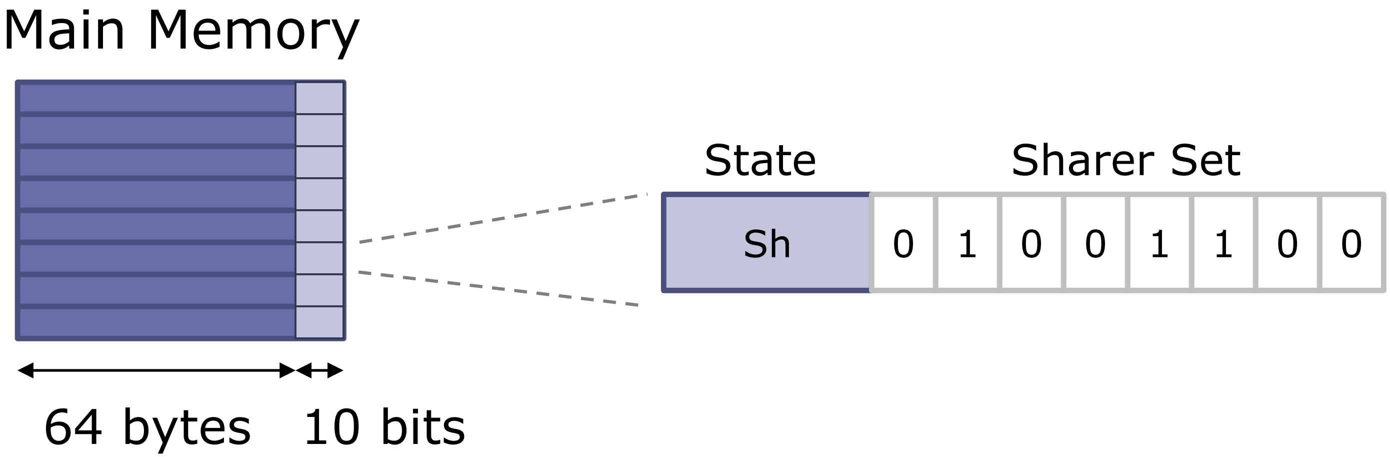
Directory Organization

- 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…

Flat, Memory-based Directories

- Dedicate a few bits of main memory to store the state and sharers of every line
- Encode sharers using a bit-vector



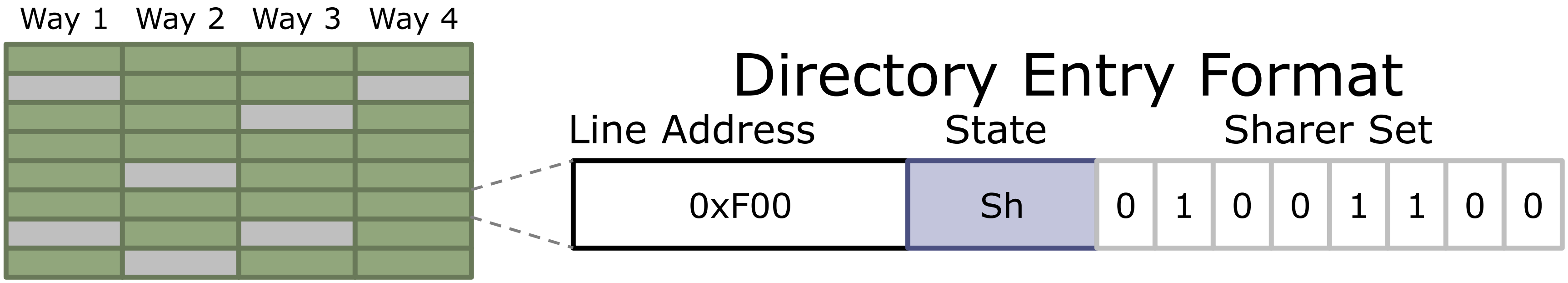
✓ Simple

✘ Slow

✘ Very inefficient with many processors (~P bits/line)

Sparse Full-Map Directories

- Not every line in the system needs to be tracked – only those in private caches!
- Idea: Organize directory as a cache



Directory Entry Format

Line Address	State	Sharer Set							
0xF00	Sh	0	1	0	0	1	1	0	0

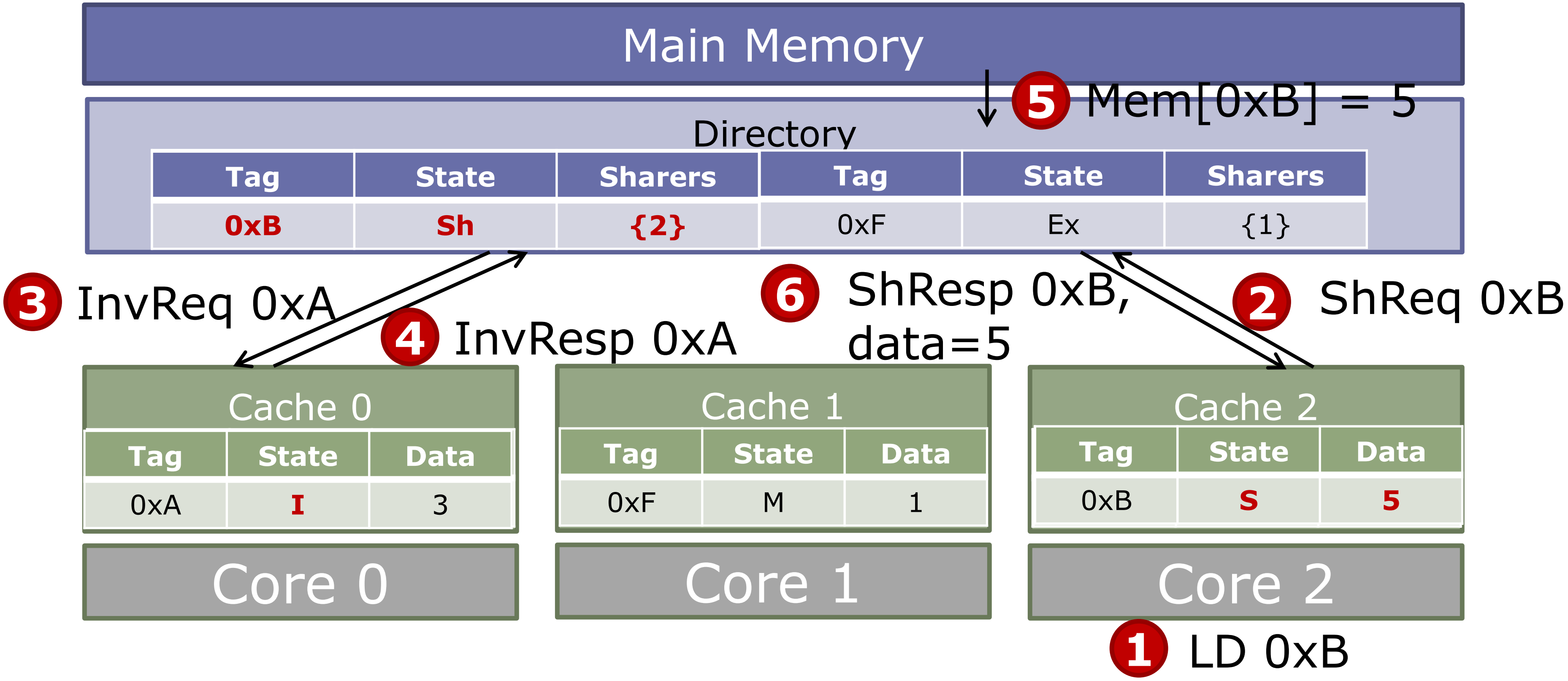
✓ Low latency, energy-efficient

✗ Bit-vectors grow with # cores → Area scales poorly

✗ Limited associativity → Directory-induced invalidations

Directory-Induced Invalidations

- To retain inclusion, must invalidate all sharers of an entry before reusing it for another address
- Example: 2-way set-associative sparse directory



Main Memory

5 Mem[0xB] = 5

Directory

Tag	State	Sharers	Tag	State	Sharers
0xB	Sh	{2}	0xF	Ex	{1}

3 InvReq 0xA

4 InvResp 0xA

6 ShResp 0xB, data=5

2 ShReq 0xB

Cache 0

Tag	State	Data
0xA	I	3

Cache 1

Tag	State	Data
0xF	M	1

Cache 2

Tag	State	Data
0xB	S	5

Core 0

Core 1

Core 2

1 LD 0xB

How many entries should the directory have?

Inexact Representations of Sharer Sets

- Coarse-grain bit-vectors (e.g., 1 bit per 4 cores)

Sharer Set

0	0	0	0	0	0
0-3	4-7	8-11	12-15	16-19	20-23

- Limited pointers: Maintain a few sharer pointers, on overflow mark ‘all’ and broadcast (or invalidate another sharer)

Sharer Set

0	8	14	33
all	sharer 1	sharer 2	sharer 3

- Allow false positives (e.g., Bloom filters)

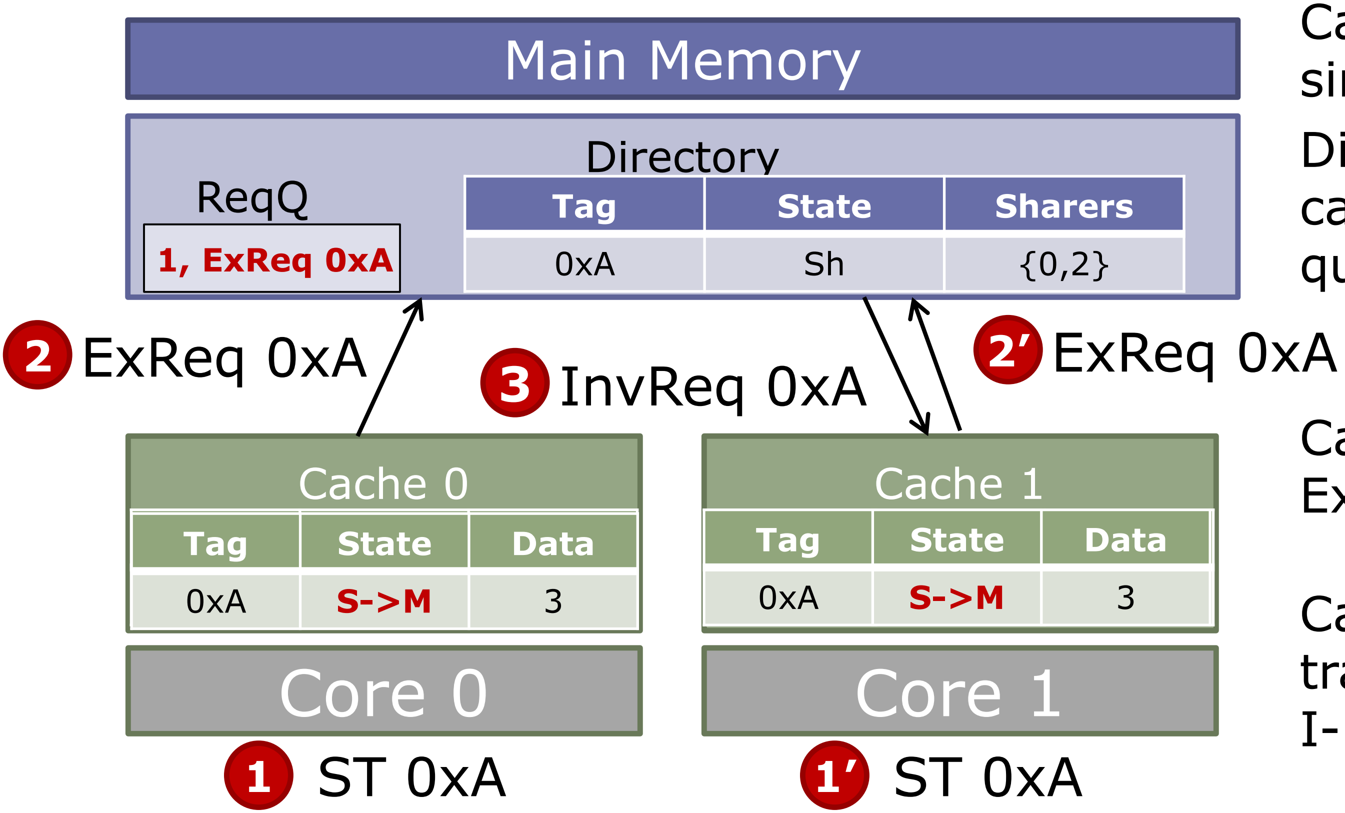
✓ Reduced area & energy

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

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

Protocol Races

- Directory serializes multiple requests for the same address
  - Same-address requests are queued or NACKed and retried
- But races still exist due to conflicting requests
- Example: Upgrade race



Directory

Tag	State	Sharers
0xA	Sh	{0,2}

Cache 0

Tag	State	Data
0xA	S->M	3

Cache 1

Tag	State	Data
0xA	S->M	3

Caches 0 and 1 issue simultaneous ExReqs

Directory starts serving cache 0's ExReq, queues cache 1's

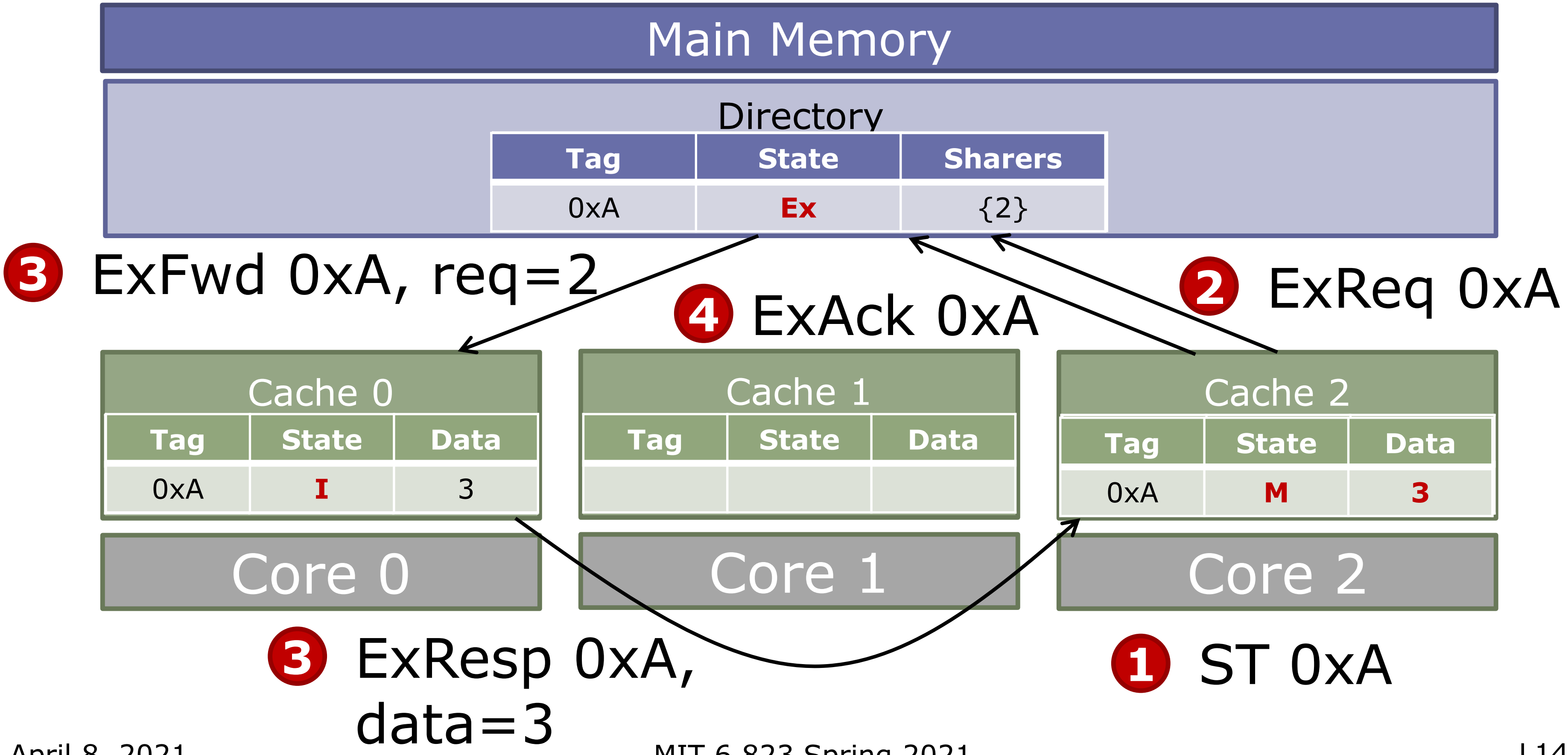
Cache 1 expected ExResp, but got InvReq!

Cache 1 should transition from S->M to I->M and send InvResp

Extra Hops and 3-Hop Protocols

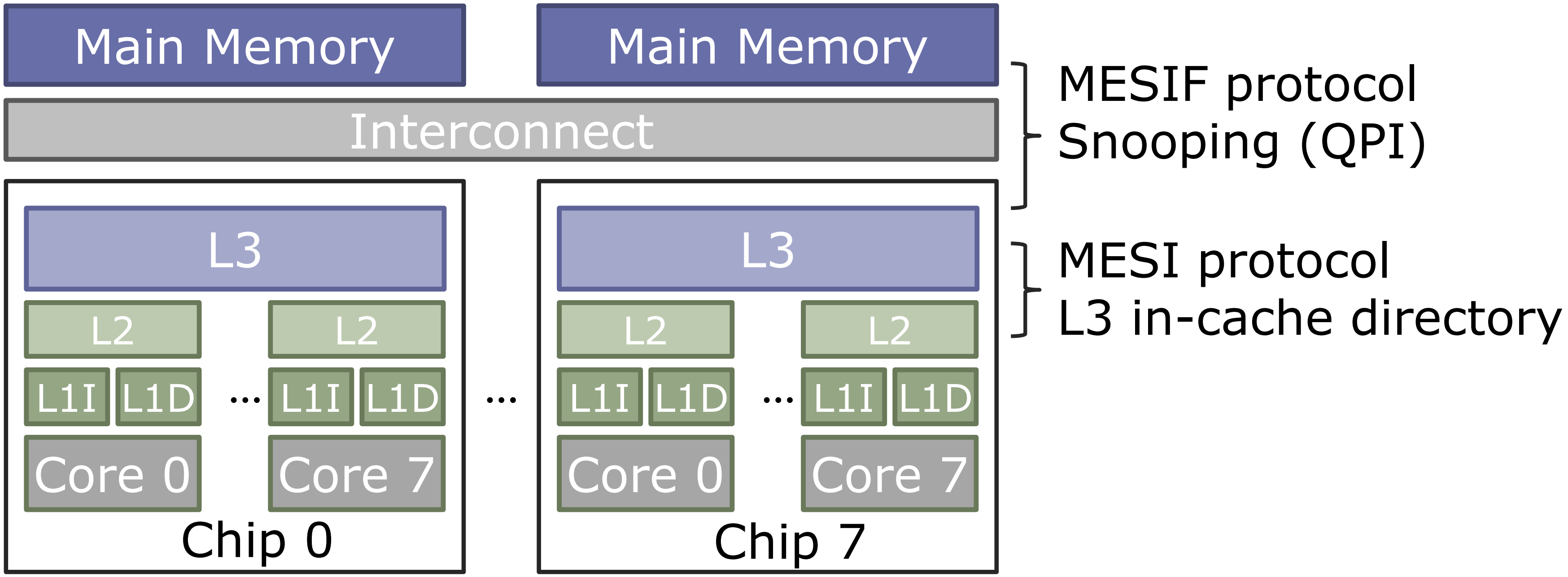
Reducing Protocol Latency

- Problem: Data in another cache needs to pass through the directory, adding latency
- Optimization: Forward data to requester directly



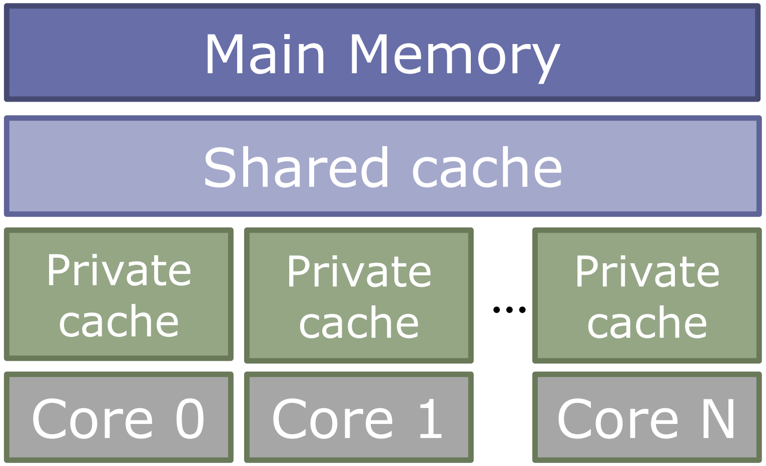
Coherence in Multi-Level Hierarchies

- Can use the same or different protocols to keep coherence across multiple levels
- Key invariant: Ensure sufficient permissions in all intermediate levels
- Example: 8-socket Xeon E7 (8 cores/socket)



In-Cache Directories

- Common multicore memory hierarchy:
  - 1+ levels of private caches
  - A shared last-level cache
  - Need to enforce coherence among private caches
- Idea: Embed the directory information in shared cache tags
  - Shared cache must be inclusive

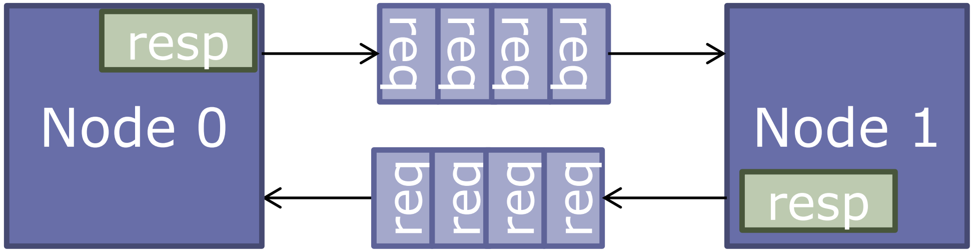


✓ Avoids tag overheads & separate lookups

✗ Can be inefficient if shared cache size >> sum(private cache sizes)

Avoiding Protocol Deadlock

- Protocols can cause deadlocks even if network is deadlock-free! (*more on this later*)



Example: Both nodes saturate all intermediate buffers with requests to each other, blocking responses from entering the network

- Solution: Separate *virtual networks*
  - Different sets of virtual channels and endpoint buffers
  - Same physical routers and links

- Most protocols require at least 2 virtual networks (for requests and replies), often >2 needed

Implementing Atomic Instructions

- In general, an *atomic read-modify-write* instruction requires two memory operations without intervening memory operations by other processors

- Implementation options:
  - *With snoopy coherence, lock the bus -> expensive*
  - *With directory-based coherence, lock the line in the cache (prevent invalidations or evictions until atomic op finishes) -> complex*

- Modern processors often use

  load-reserve
  store-conditional

Load-reserve & Store-conditional

Special register(s) to hold reservation flag and address, and the outcome of store-conditional

```
Load-reserve R, (a):
    <flag, adr> ← <1, a>;
    R ← M[a];
```

```
Store-conditional (a), R:
    if <flag, adr> == <1, a>
    then  cancel other procs'
            reservation on a;
          M[a] ← <R>;
          status ← succeed;
    else  status ← fail;
```

If the cache receives an invalidation to the address in the reserve register, the reserve bit is set to 0

- Several processors may reserve 'a' simultaneously
- These instructions are like ordinary loads and stores with respect to the bus traffic

Load-Reserve/Store-Conditional

Swap implemented with Ld-Reserve/St-Conditional

```
#       Swap(R1, mutex):

L:      Ld-Reserve R2, (mutex)
        St-Conditional (mutex), R1
        if (status == fail) goto L
        R1 <- R2
```

Performance:
Load-reserve & Store-conditional

The total number of coherence transactions is not necessarily reduced, but splitting an atomic instruction into load-reserve & store-conditional:

- *increases utilization* (and reduces processor stall time), especially in split-transaction buses and directories

- *reduces cache ping-pong effect* because processors trying to acquire a semaphore do not have to perform stores each time

Thank you!

Next Lecture:
Consistency and
Relaxed Memory Models