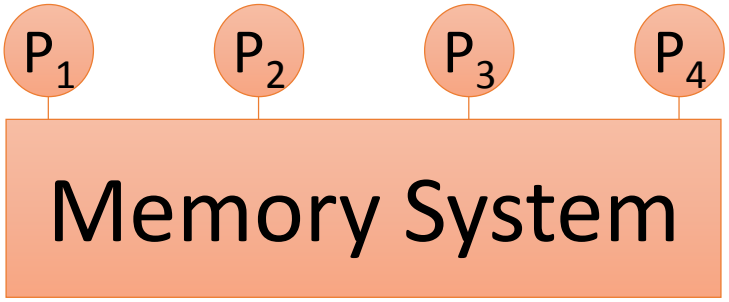
Cache Coherence

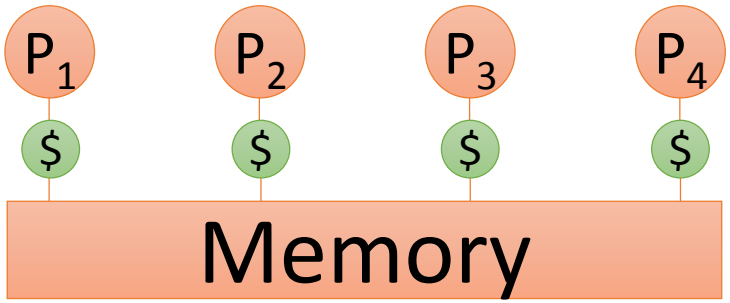
Nima Honarmand

Cache Coherence: Problem (Review)

- Problem arises when
  - There are multiple physical copies of one logical location
- Multiple copies of each cache block (In a shared-mem system)
  - One in main memory
  - Up to one in each cache
- Copies become inconsistent when writes happen
- Does it happen in a uniprocessor system too?
  - Yes, I/O writes can make the copies inconsistent



Logical View



Reality (more or less!)

Coherence: An Example Execution

Processor 0

```
0: addi r1,accts,r3
1: ld 0(r3),r4
2: blt r4,r2,6
3: sub r4,r2,r4
4: st r4,0(r3)
5: call spew_cash
```

Processor 1

```
0: addi r1,accts,r3
1: ld 0(r3),r4
2: blt r4,r2,6
3: sub r4,r2,r4
4: st r4,0(r3)
5: call spew_cash
```

- Two $100 withdrawals from account #241 at two ATMs
  - Each transaction maps to thread on different processor
  - Track `accts[241].bal` (address is in `r3`)

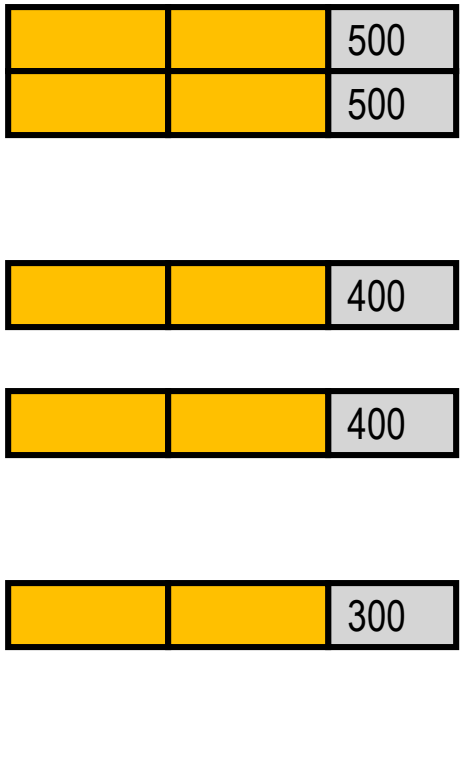
No-Cache, No-Problem

Processor 0

```
0: addi r1,accts,r3
1: ld 0(r3),r4
2: blt r4,r2,6
3: sub r4,r2,r4
4: st r4,0(r3)
5: call spew_cash
```

Processor 1

```
0: addi r1,accts,r3
1: ld 0(r3),r4
2: blt r4,r2,6
3: sub r4,r2,r4
4: st r4,0(r3)
5: call spew_cash
```



- Scenario I: processors have no caches
  - No problem

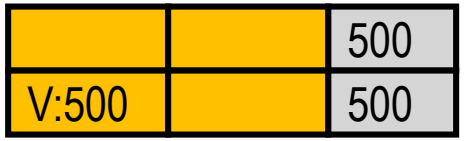
Cache Incoherence

Processor 0

```
0: addi r1,accts,r3
1: ld 0(r3),r4
2: blt r4,r2,6
3: sub r4,r2,r4
4: st r4,0(r3)
5: call spew_cash
```

Processor 1

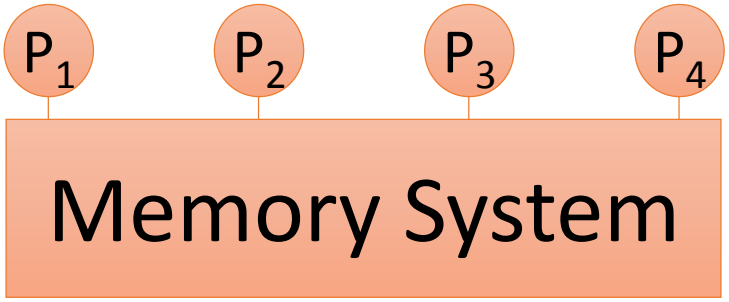
```
0: addi r1,accts,r3
1: ld 0(r3),r4
2: blt r4,r2,6
3: sub r4,r2,r4
4: st r4,0(r3)
5: call spew_cash
```



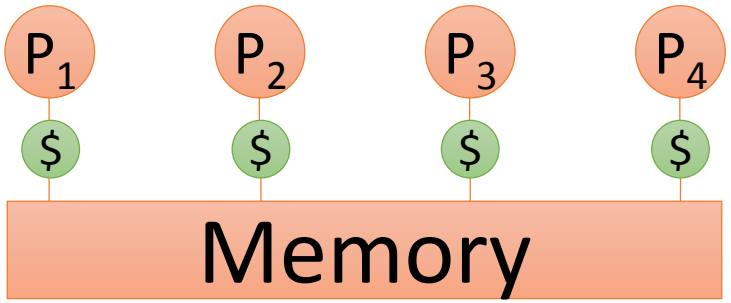
- Scenario II: processors have write-back caches
  - Potentially 3 copies of `accts[241].bal`: memory, P0 $, P1 $
  - Can get incoherent (inconsistent)

But What's the Problem w/ Incoherence?

- ***Problem***: the behavior of the physical system becomes different from the logical system
- Loosely speaking, cache coherence tries to hide the existence of multiple copies (real system)
  - And make the system behave as if there is just one copy (logical system)



Logical View



Reality (more or less!)

View of Memory in the Logical System

- In the logical system
  - For each mem. location ***M***, there is just one copy of the value
- Consider all the reads and writes to ***M*** in an execution
  - At most one write can update ***M*** at any moment
    - i.e., there will be a total order of writes to ***M***
    - Let's call them WR_1, WR_2, …
  - A read to ***M*** will return the value written by some write (say WR_i)
    - This means the read is ordered after WR_i and before WR_{i+1}
- The notion of "last write to a location" is globally well-defined

Cache Coherence Defined

- Coherence means to provide the same semantic in a system with multiple copies of ***M***

- Formally, a memory system is coherent iff it behaves as if for any given mem. location ***M***
  - There is a ***total order*** of all writes to ***M***
    - Writes to ***M*** are serialized
  - If RD_j happens after WR_i, it returns the value of WR_i or a write ordered after WR_i
  - If WR_i happens after RD_j, it does not affect the value returned by RD_j

- What does "happens after" above mean?

Coherence is only concerned w/ reads & writes on a single location

Coherence Protocols

Approaches to Cache Coherence

- ***Software-based solutions***
  - compiler or run-time software support
- ***Hardware-based solutions***
  - Far more common
- ***Hybrid solutions***
  - Combination of hardware/software techniques
  - E.g., a block might be under SW coherence first and then switch to HW cohrence
  - Or, hardware can track sharers and SW decides when to invalidate them
  - And many other schemes…

We'll focus on hardware-based coherence

Software Cache Coherence

- Software-based solutions
  - Mechanisms:
    - Add “Flush” and “Invalidate” instructions
    - “Flush” writes all (or some specified) dirty lines in my $ to memory
    - “Invalidate” invalidate all (or some specified) valid lines in my $
  - Could be done by compiler or run-time system
    - Should know what memory ranges are shared and which ones are private (i.e., only accessed by one thread)
    - Should properly use “invalidate” and “flush” instructions at “communication” points
  - Difficult to get perfect
    - Can induce a lot of unnecessary “flush”es and “invalidate”s → reducing cache effectiveness
- Know any “cache” that uses software coherence today?
  - TLBs are a form of cache and use software-coherence in most machines

Hardware Coherence Protocols

- Coherence protocols closely interact with
  - Interconnection network
  - Cache hierarchy
  - Cache write policy (write-through vs. write-back)
- Often designed together
- Hierarchical systems have different protocols at different levels
  - On chip, between chips, between nodes

Elements of a Coherence Protocol

- Actors
  - Elements that have a copy of memory locations and should participate in the coherence protocol
  - For now, caches and main memory
- States
  - Stable: states where there are no on-going transactions
  - Transient: states where there are on-going transactions
- State transitions
  - Occur in response to local operations or remote messages
- Messages
  - Communication between different actors to coordinate state transitions
- Protocol transactions
  - A group of messages that together take system from one stable state to another

Mostly Interconnect Independent

Interconnect Dependent

Coherence as a Distributed Protocol

- Remember, coherence is per memory location
  - For now, per cache line
- Coherence protocols are distributed protocols
  - Different types of actors have different FSMs
    - Coherence FSM of a cache is different from the memory's
  - Each actor maintains a state for each cache block
    - States at different actors might be different (***local states***)
    - The overall "protocol state" (***global state***) is the aggregate of all the per-actor states
  - The set of all local states should be consistent
    - *e.g.*, if one actor has exclusive access to a block, every one else should have the block as inaccessible (invalid)

Coherence Protocols Classification (1)

- ***Update*** vs. ***Invalidate***: what happens on a write?
  - update other copies, or
  - invalidate other copies
- Invalidation is bad when:
  - producer and (one or more) consumers of data
- Update is bad when:
  - multiple writes by one PE before data is read by another PE
  - Junk data accumulates in large caches (e.g. process migration)
- Today, invalidation schemes are by far more common
  - Partly because they are easier to implement

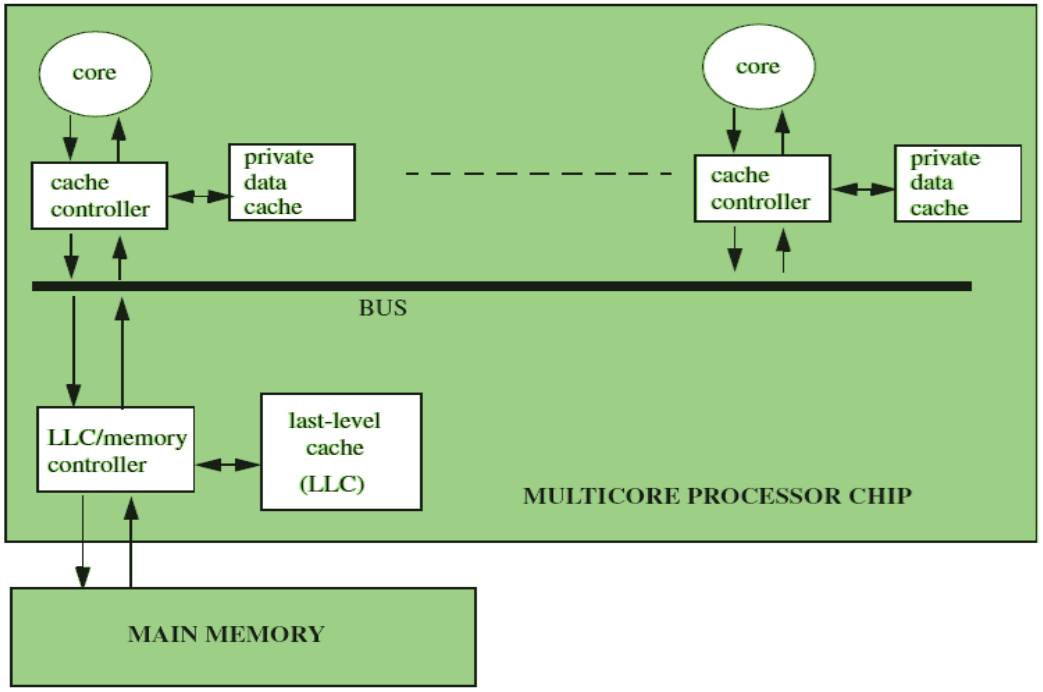
Coherence Protocols Classification (2)

- ***Broadcast*** vs. ***unicast***: make the transaction visible…
  - to all other processors (a.k.a. ***snoopy coherence***)
    - Small multiprocessors (a few cores)
  - only those that have a cached copy of the line (aka ***directory coherence*** or ***scalable coherence***)
    - > 10s of cores
- Many systems have hybrid mechanisms
  - Broadcast locally, unicast globally

Snoopy Protocols

Bus-based Snoopy Protocols

- For now assume a one-level coherence hierarchy
  - Like a single-chip multicore
  - Private L1 caches connected to last level cache/memory through a bus
- Assume write-back caches



Bus-based Snoopy Protocols

- Assume ***atomic bus***
  - Request goes out & reply comes back without relinquishing the bus
- Assume ***non-atomic request***
  - It takes while from when a cache makes a request until the bus is granted and the request goes on the bus
- All actors listen to (***snoop***) the bus requests and change their local state accordingly
  - And if need be provide replies
- Shared bus and its being atomic makes it easy to enforce write serialization
  - Any write that goes on the bus will be seen by everyone at the same time
  - We say bus is the ***point of serialization*** in the protocol

Example 1: MSI Protocol

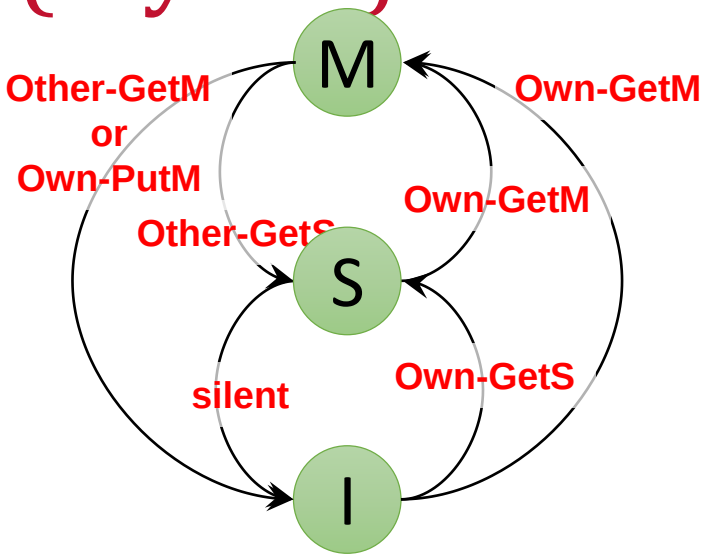
- Three states tracked per-block at each cache and LLC
  - Invalid – cache does not have a copy
  - Shared – cache has a read-only copy; clean
    - Clean == memory is up to date
  - Modified – cache has the only copy; writable; dirty
    - Dirty == memory is out of date
- Transactions
  - GetS(hared), GetM(odified), PutM(odified)
- Messages
  - GetS, GetM, PutM, Data (data reply)

Describing Coherence Protocols

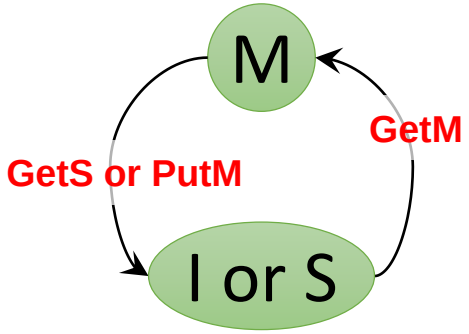
- Two methods
  - High-level: interaction between stable states and transactions
    - Often shown as an FSM diagram
  - Detailed: complete specification including transient states and messages in addition to the above
    - Often shown as a table
- Either way, should always describe protocol transitions and states for each actor type

MSI: High-Level Spec (Style 1)

- High-level state transitions in response to requests on the bus
  - Not showing local processor load/stores/evicts explicitly
  - Now showing responses
- Mem state aggregate of cache states
  - "I" at mem = all caches are "I"; "S" at mem = "S" in some caches; "M" at mem = "M" in one cache.
- ***Own*** means observing my cache's own request; ***Other*** means another cache's request



FSM at cache controller



FSM at memory controller

MSI: Detailed Specification (1/2)

- Detailed specification provides complete state transition + actions to be taken on a transition + transient states
- AB^X means a transient state during transition from state A to B which is waiting for event(s) X before moving to B

	GetS	GetM	PutM	Data From Owner	NoData
IorS	send data to requestor	send data to requestor/M	-/$IorS^D$		
$IorS^D$	(A)	(A)		write data to LLC/memory /IorS	-/IorS
M	-/$IorS^D$		-/M^D		
M^D	(A)	(A)		write data to LLC/IorS	-/M

Source: A Primer on Memory Consistency and Cache Coherence

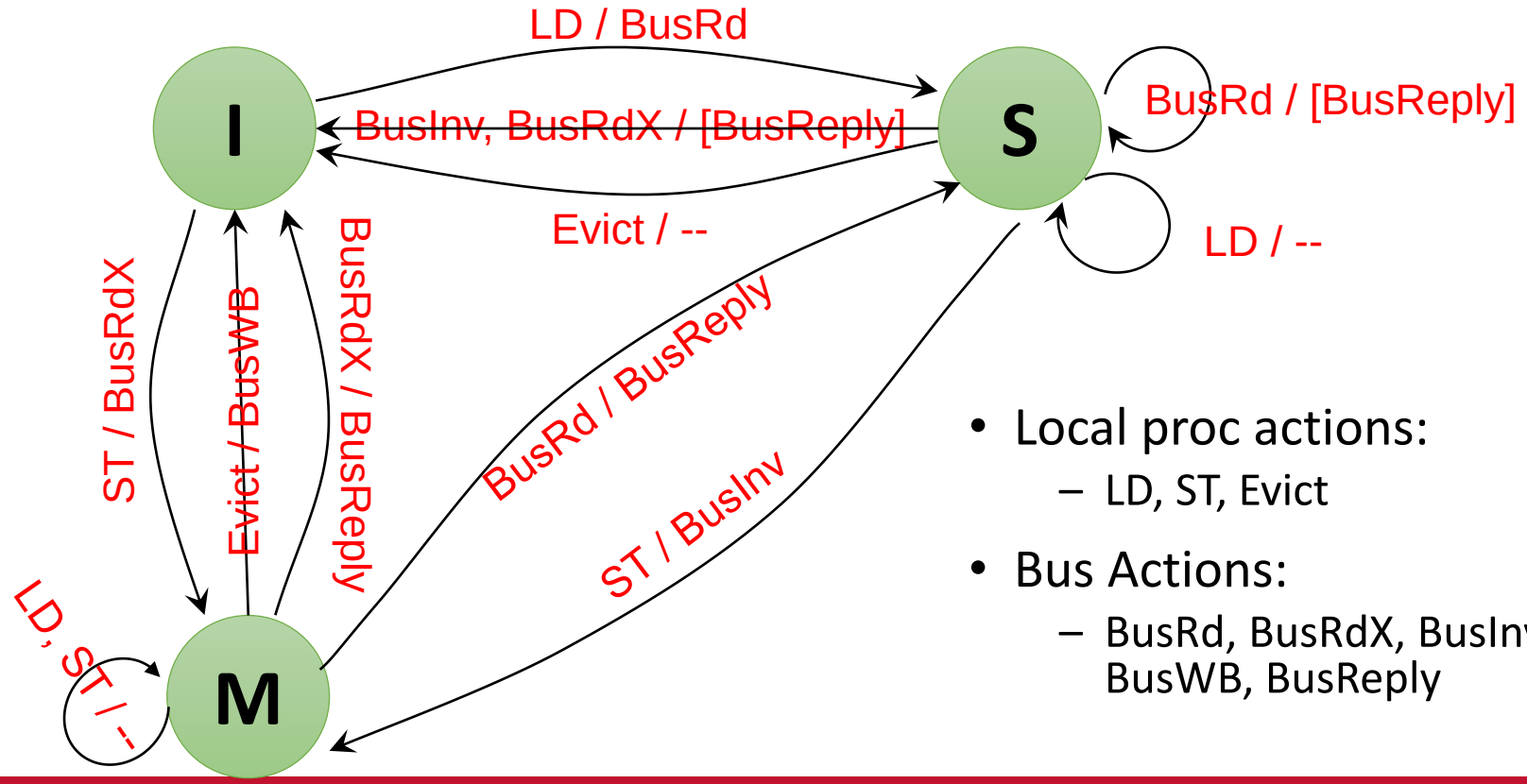
MSI: Detailed Specification (2/2)

	load	store	replacement	OwnGetS	OwnGetM	OwnPutM	OtherGetS	OtherGetM	OtherPutM	Own Data response
I	issue GetS/IS^{AD}	issue GetM/IM^{AD}					-	-	-	
IS^{AD}	stall	stall	stall	-/IS^{D}			-	-	-	
IS^{D}	stall	stall	stall				(A)	(A)		-/S
IM^{AD}	stall	stall	stall		-/IM^{D}		-	-	-	
IM^{D}	stall	stall	stall				(A)	(A)		-/M
S	hit	issue GetM/SM^{AD}	-/I				-	-/I	-	
SM^{AD}	hit	stall	stall		-/SM^{D}		-	-/IM^{AD}	-	
SM^{D}	hit	stall	stall				(A)	(A)		-/M
M	hit	hit	issue PutM/MI^{A}				send data to requestor and to memory/S	send data to requestor/I	-	
MI^{A}	hit	hit	stall			send data to memory/I	send data to requestor and to memory/II^{A}	send data to requestor/II^{A}		
II^{A}	stall	stall	stall			send NoData to memory/I	-	-		

Source: A Primer on Memory Consistency and Cache Coherence

MSI: High-level Spec (Style 2)

- Only shows $ transitions; mem transitions must be inferred
- "X"/ "Y" means do "Y" in response to "X"



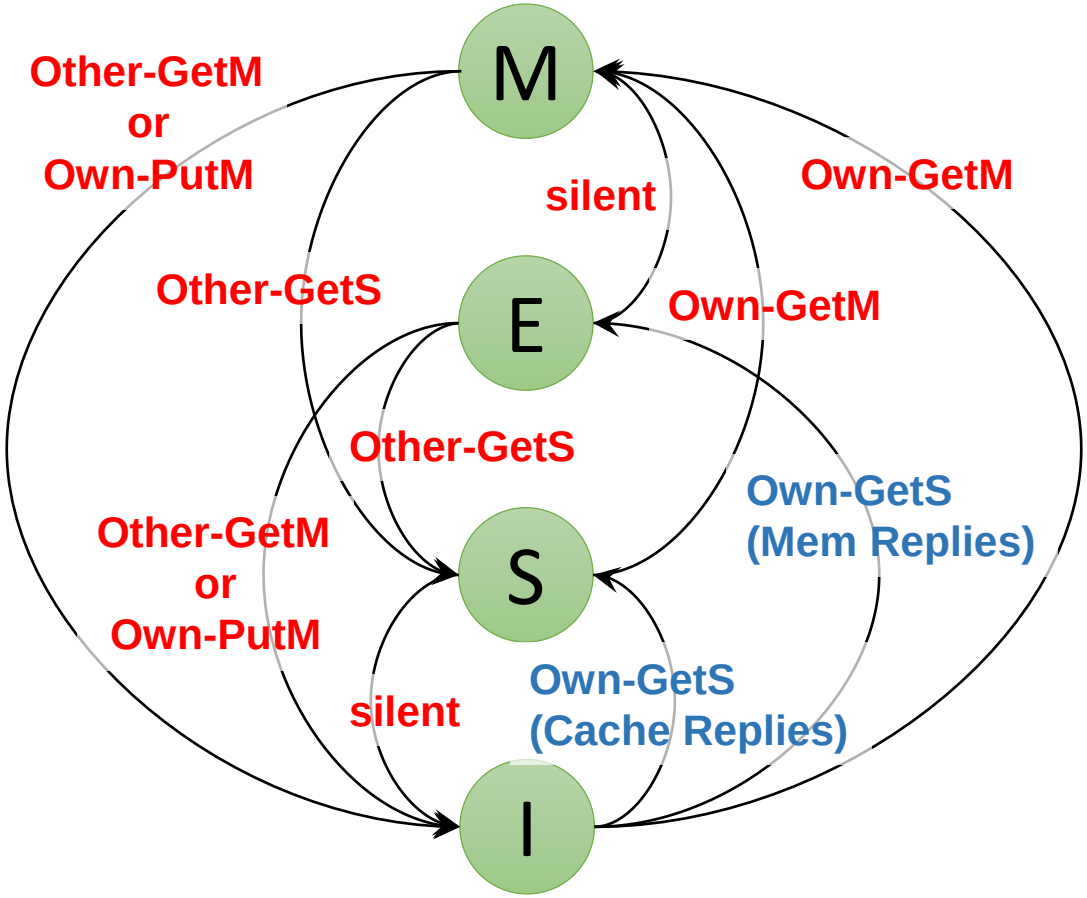
- Local proc actions:
  - LD, ST, Evict
- Bus Actions:
  - BusRd, BusRdX, BusInv, BusWB, BusReply

Often you see FSMs like these

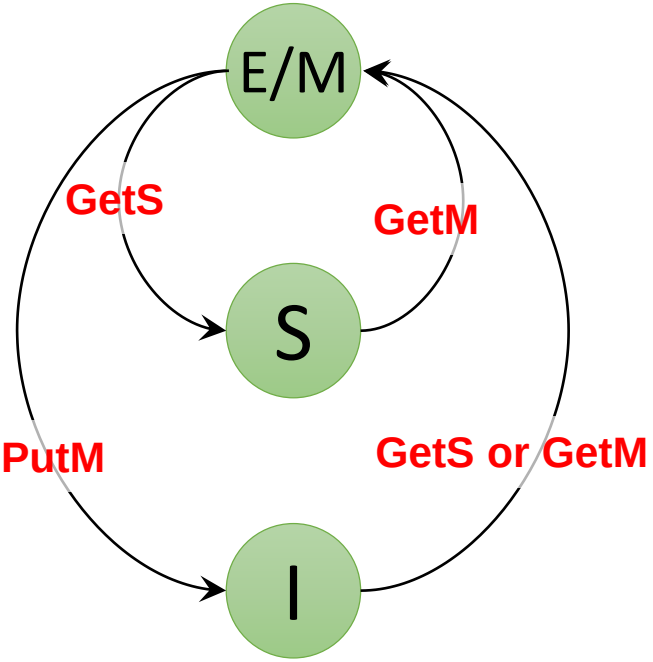
Example 2: Illinois Protocol

- States: I, E (Exclusive), S (Shared), M (Modified)
  - Called ***MESI*** ☺
  - Widely used in real machines
- Two features :
  - The cache knows if it has an Exclusive (E) copy
  - If some cache has a copy in E state, cache-cache transfer is used
- Advantages:
  - In E state no invalidation traffic on write-hits
    - Cuts down on ***upgrade*** traffic for lines that are first read and then written
  - Closely approximates traffic on a uniprocessor for sequential programs
  - Cache-cache transfer can cut down latency in some machine
- Disadvantages:
  - complexity of mechanism that determines exclusiveness
  - memory needs to wait before sharing status is determined

Illinois: High-Level Specification



FSM at cache controller



FSM at memory controller

See the "Primer" (Sec 7.3) for the detailed spec

Adding an "Owned" State: MOESI

- MESI must write-back to memory on M→S transition (a.k.a. ***downgrade***)
  - Because protocol allows "silent" evicts from shared state, a dirty block might otherwise be lost
  - But, the writebacks might be a waste of bandwidth
    - *e.g.*, if there is a subsequent store (common in producer-consumer scenarios)
- Solution: add an "Owned" state
  - Owned – shared, but dirty; only one owner (others enter S)
  - Owner is responsible for replying to read requests
  - Owner is responsible for writeback upon eviction
    - Or should transfer "ownership" to another cache

MOESI Framework
[Sweazey & Smith, ISCA'86]

M - Modified (dirty)

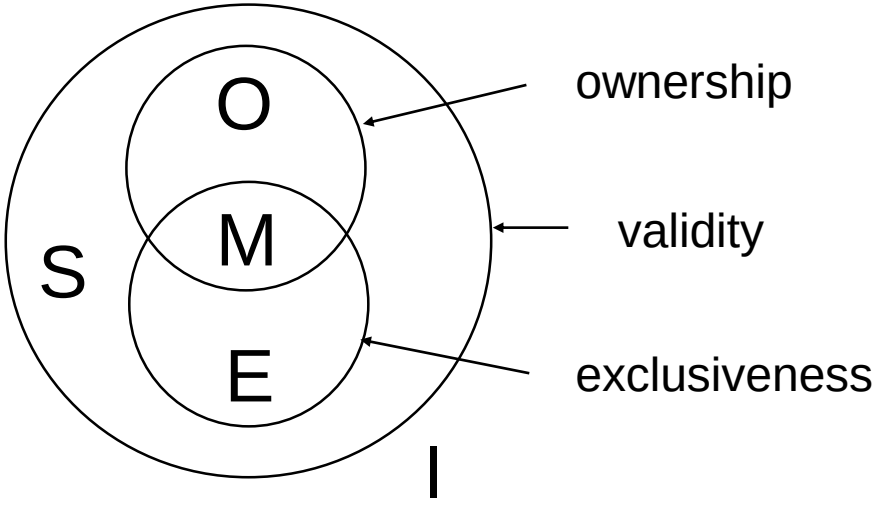
O - Owned (dirty but shared)

E - Exclusive (clean unshared) only copy, not dirty

S - Shared

I - Invalid

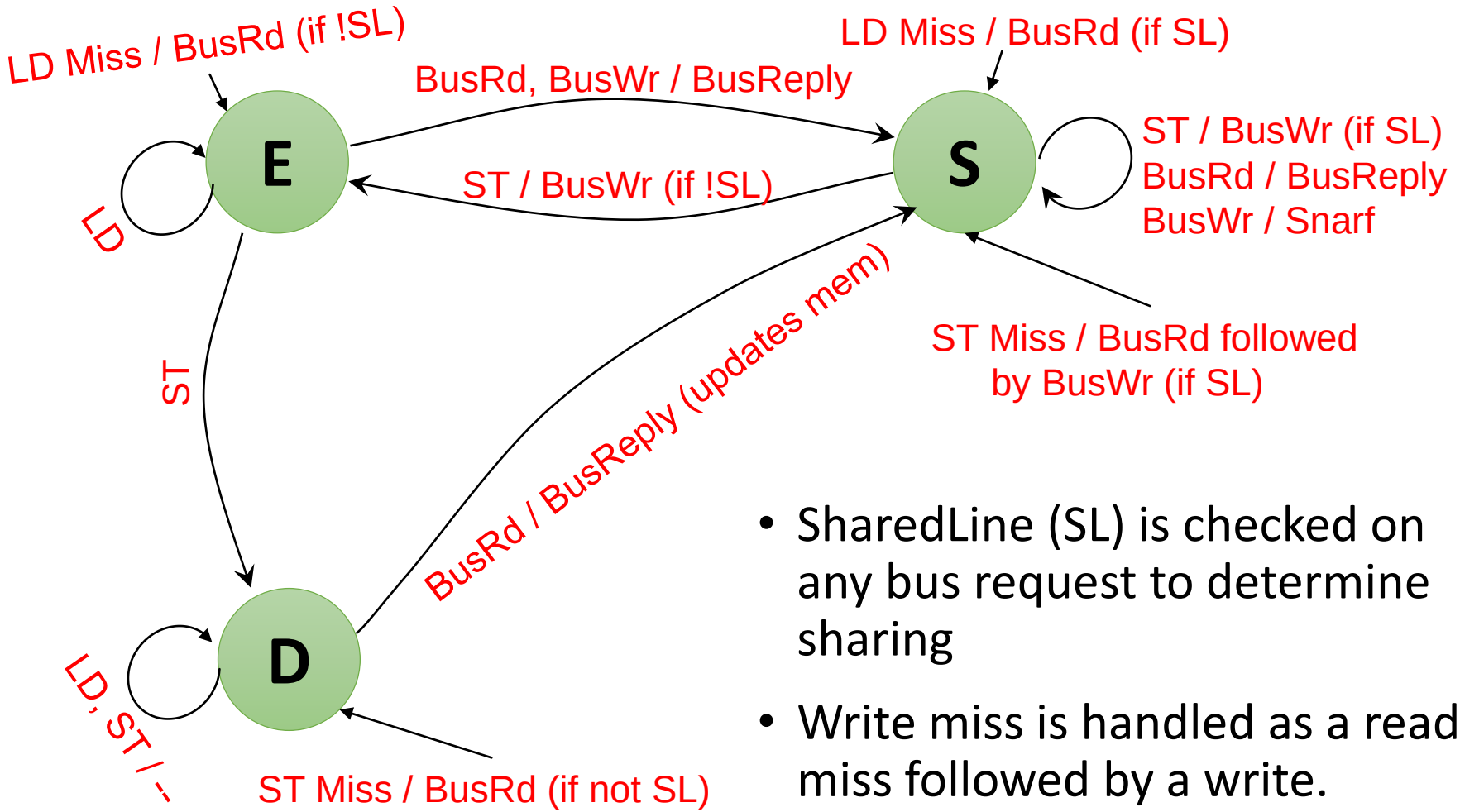
Variants
- MSI
- MESI
- MOSI
- MOESI



Example 3: DEC Firefly

- An update-based protocol for write-back caches
- States
  - Exclusive – only one copy; writeble; clean
  - Shared – multiple copies; write hits write-through to all sharers and memory
  - Dirty – only one copy; writeable; dirty
- Exclusive/dirty provide write-back semantics for private data
- Shared state provides update semantics for shared data
  - Uses “shared line” bus wire to detect sharing status

DEC Firefly: High-level Specification

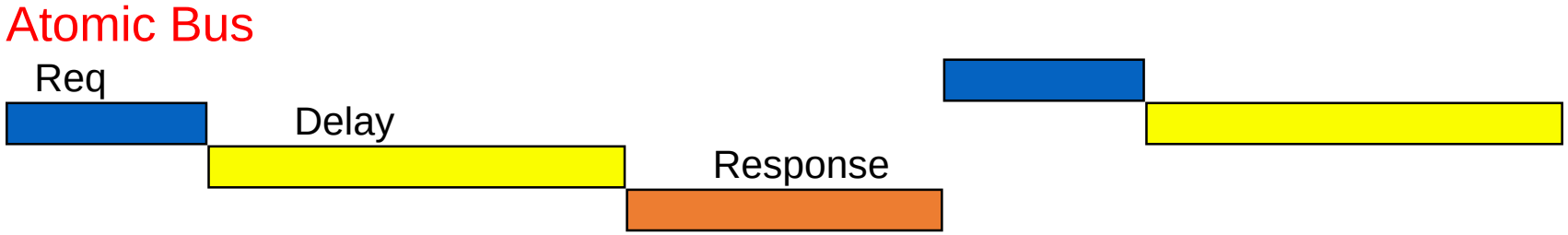


- SharedLine (SL) is checked on any bus request to determine sharing
- Write miss is handled as a read miss followed by a write.

Advanced Issues in Snoopy Coherence

Non-Atomic Interconnect

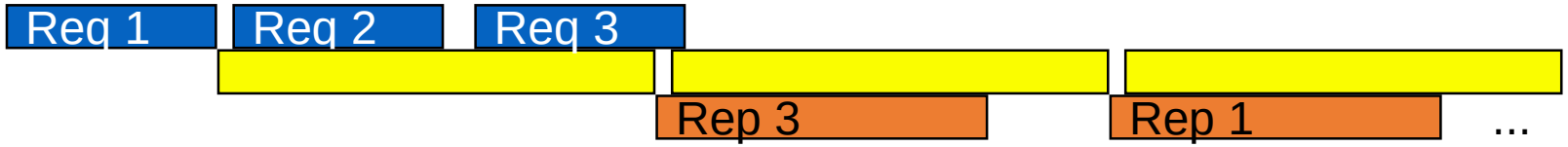
- (A) entries in the previous tables: situations that cannot happen because bus is atomic
  - *i.e.*, bus is not released until the transaction is complete
  - Cannot have multiple on-going requests for the same line
- Atomic buses waste time and bus bandwidth
  - Responding to a request involves multiple actions
    - Look up cache tags on a snoop
    - Inform upper cache layers (if multi-level cache)
    - Access lower levels (*e.g.*, LLC accessing memory before replying)



Split-Transaction Buses

- Can overlap req/resp of multiple transactions on the bus
  - Need to identify request/response using a tag

Split-transaction Bus



Issues:

- ***Protocol races*** become possible
  - Protocol races result in more transient states
- Need to buffer requests and responses
  - Buffer own reqs: req bus might be busy (taken by someone else)
  - Buffer other actors' reqs: busy processing another req
  - Buffer resps: resp bus might be busy (taken by someone else)

More Races: MSI w/ Split-Txn Bus

- (A) entries are now possible
- New transient states

	load	store	replacement	OwnGetS or OwnGetM	OwnGetM	OwnPutM	OtherGetS	OtherGetM	OtherPutM	Own Data response (for own request)
I	issue GetS/IS^{AD}	issue GetM/IM^{AD}					-	-	-	
IS^{AD}	stall	stall	stall	-/IS^{D}			-	-	-	-/IS^{A}
IS^{D}	stall	stall	stall				-	**stall**		load hit/S
IS^{A}	**stall**	**stall**	**stall**	**load hit/S**			-	-		
IM^{AD}	stall	stall	stall		-/IM^{D}		-	-	-	-/IM^{A}
IM^{D}	stall	stall	stall				**stall**	**stall**		store hit/M
IM^{A}	**stall**	**stall**	**stall**		**store hit/M**		-	-		
S	hit	issue GetM/SM^{AD}	-/I				-	-/I		
SM^{AD}	hit	stall	stall		-/SM^{D}		-	-/IM^{AD}		-/SM^{A}
SM^{D}	hit	stall	stall				**stall**	**stall**		store hit/M
SM^{A}	**hit**	**stall**	**stall**		**store hit/M**		-	-/IM^{A}		
M	hit	hit	issue PutM/MI^{A}				send data to requestor and to memory/S	send data to requestor/I		
MI^{A}	hit	hit	stall			send data to requestor /I	send data to requestor and to memory/II^{A}	send data to requestor/II^{A}		
II^{A}	stall	stall	stall			-/I	-	-	-	

Source: A Primer on Memory Consistency and Cache Coherence

Ordering Issues (1)

- How to ensure "write serialization" property?
  - Recall: writes to the same location should be appear in the same order to all caches
- Solution: have a FIFO queue for all snooped requests
  - Own as well as others'
- Add snooped requests at the FIFO tail
- Process each requests when at the FIFO head

→ All controllers process all the reqs in the same order

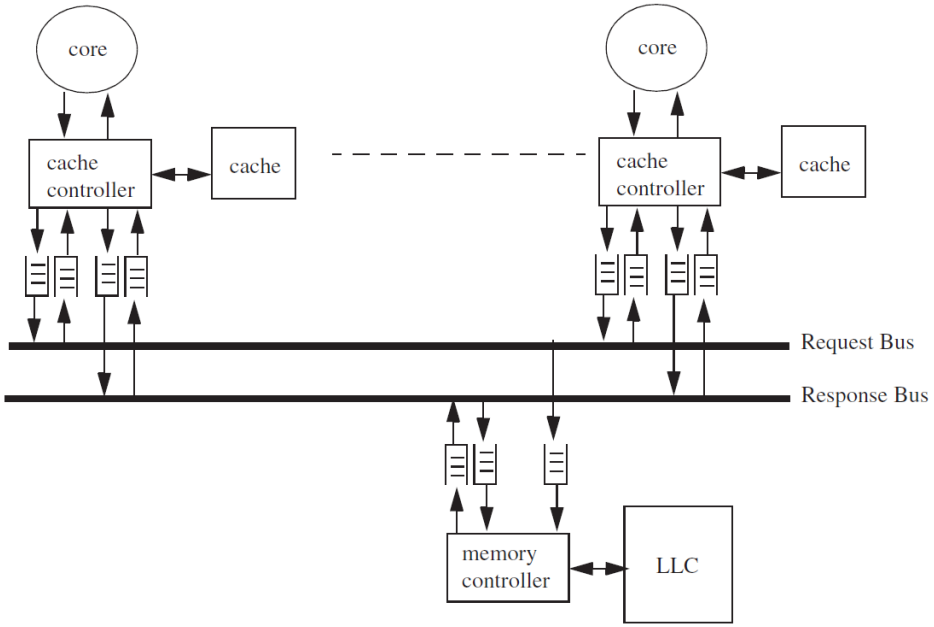
- i.e., the bus order
- Bus is the point of serialization

Buffering Issues (2)

- What to do if the buffers are full?
  - Someone puts a req on the bus but my buffer is full
    - NACK the request (sender should repeat)
    - Stall the request/bus until buffer space available
  - I have a resp but my resp buffer is full
    - Stall processing further requests until space is available
- Problem: Stalling can result in deadlock if not careful
  - ***Deadlock***: when two or more actors are circularly waiting on each other and cannot make progress
- Problem: NACKing can result in livelock if not careful
  - ***Livelock***: when two or more actors are busy (*e.g.*, sending messages) but cannot make forward progress
- Both caused by cyclic dependences

Buffering Issues (2) – Cont'd

- Common solution
  - Separate req & resp networks
  - Separate incoming req & resp queues
  - Separate outgoing req & resp queues
  - Make sure protocol can always absorb resps
    - *e.g.*, resps should not block for writebacks if replacement needed



Example Split-Txn System:

Multi-level Cache Hierarchies

- How to snoop with multi-level caches?
  - Assume private L1 and L2
  - L2 is the point of coherence

Common solutions:

1. Independent snooping at each level
   – Basically forwarding each snooped request to higher levels
2. Duplicate L1 tags at L2
3. Maintain cache ***inclusion***

The Inclusion Property

- Inclusion means L2 is a superset of L1 (ditto for L3, …)
  - Also, must propagate "dirty" bit through cache hierarchy

✓ Only need to snoop L2
  - If L2 says not present, can't be in L1 either

✘ Inclusion wastes space

✘ Inclusion takes effort to maintain
  - L2 must track what is cached in L1
  - On L2 replacement, must flush corresponding blocks from L1
  - Complicated due to (if):
    - L1 block size < L2 block size
    - Different associativity in L1
    - L1 filters L2 access sequence; affects LRU ordering

Many recent designs do not maintain inclusion

Handling Writebacks

- Allow CPU to proceed on a miss ASAP
  - Fetch the requested block
  - Do the writeback of the victim later
- Requires writeback (WB) buffer
  - Must snoop/handle bus transactions in WB buffer
- When to allocate WB buffer entry?
  - When sending request or upon receiving response?

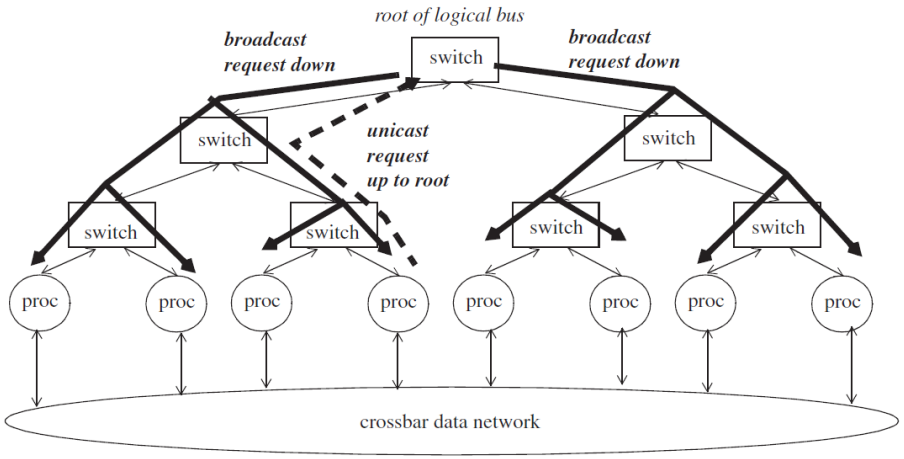
Non-Bus Snoopy Coherence

- Snoopy coherence does not need bus
- It needs totally ordered *logical* broadcasting of requests

→ Any request network w/ totally ordered broadcasts work

→ Response network can be completely unordered

- Example 1: Sun E10K
  - Tree-based point-to-point ordered req network
  - Crossbar unordered data network



- Example 2: Ring-based on-chip protocols in modern multicores

Directory Coherence Protocols

Problems w/ Snoopy Coherence

1. Bus bandwidth
   - Problem: Bus and Ring are not scalable interconnects
     - Limited bandwidth
     - Cannot support more than a dozen or so processors
   - Solution: Replace non-scalable bandwidth substrate (bus) with a scalable-bandwidth one (e.g., mesh)

2. Processor snooping bandwidth
   - Problem: All processors must monitor all bus traffic; most snoops result in no action
   - Solution: Replace non-scalable broadcast protocol (spam everyone) with scalable directory protocol (only spam cores that care)
     - The “directory” keeps track of “sharers”

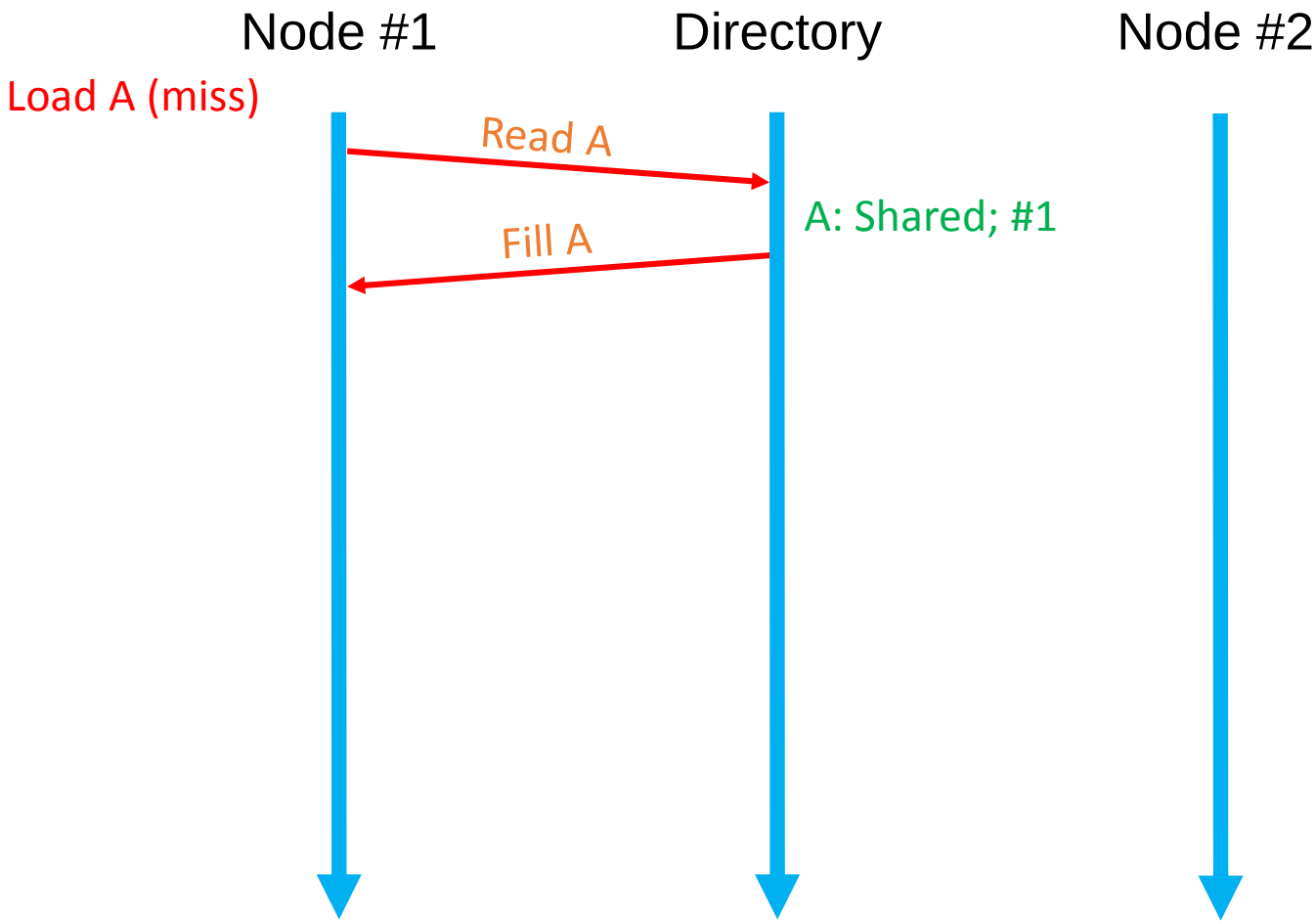
Directory Coherence 101

1. Maintain a global view of the coherence state of each block in a ***Directory***
   - ***Owner*:** which processor has a dirty copy (*i.e.*, M state)
   - ***Sharers:*** which processors have clean copies (*i.e.*, S state)

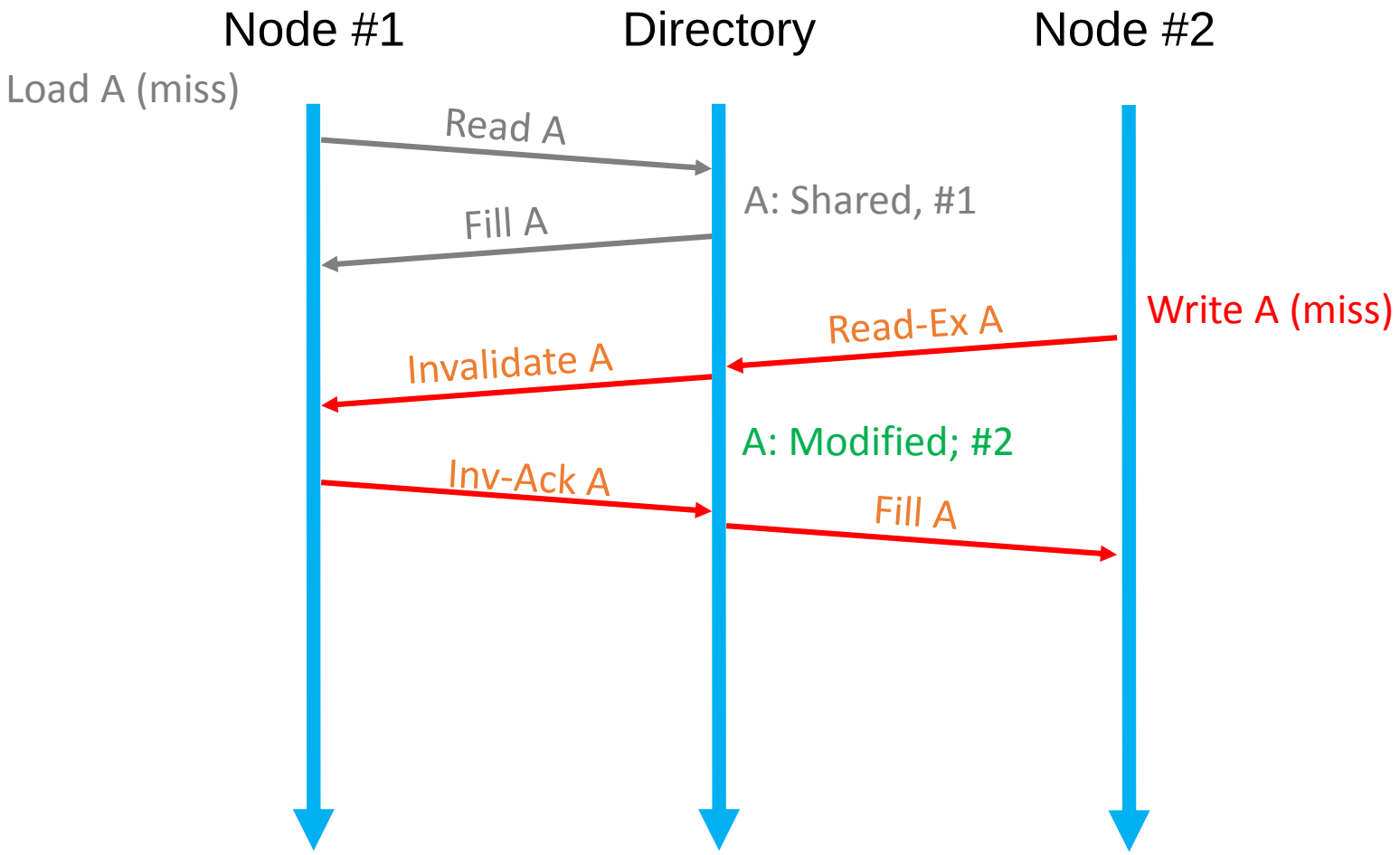
2. Instead of broadcasting, processors send coherence requests to the directory
   - Directory then informs other actors that care

3. Used with point-to-point networks (almost always)

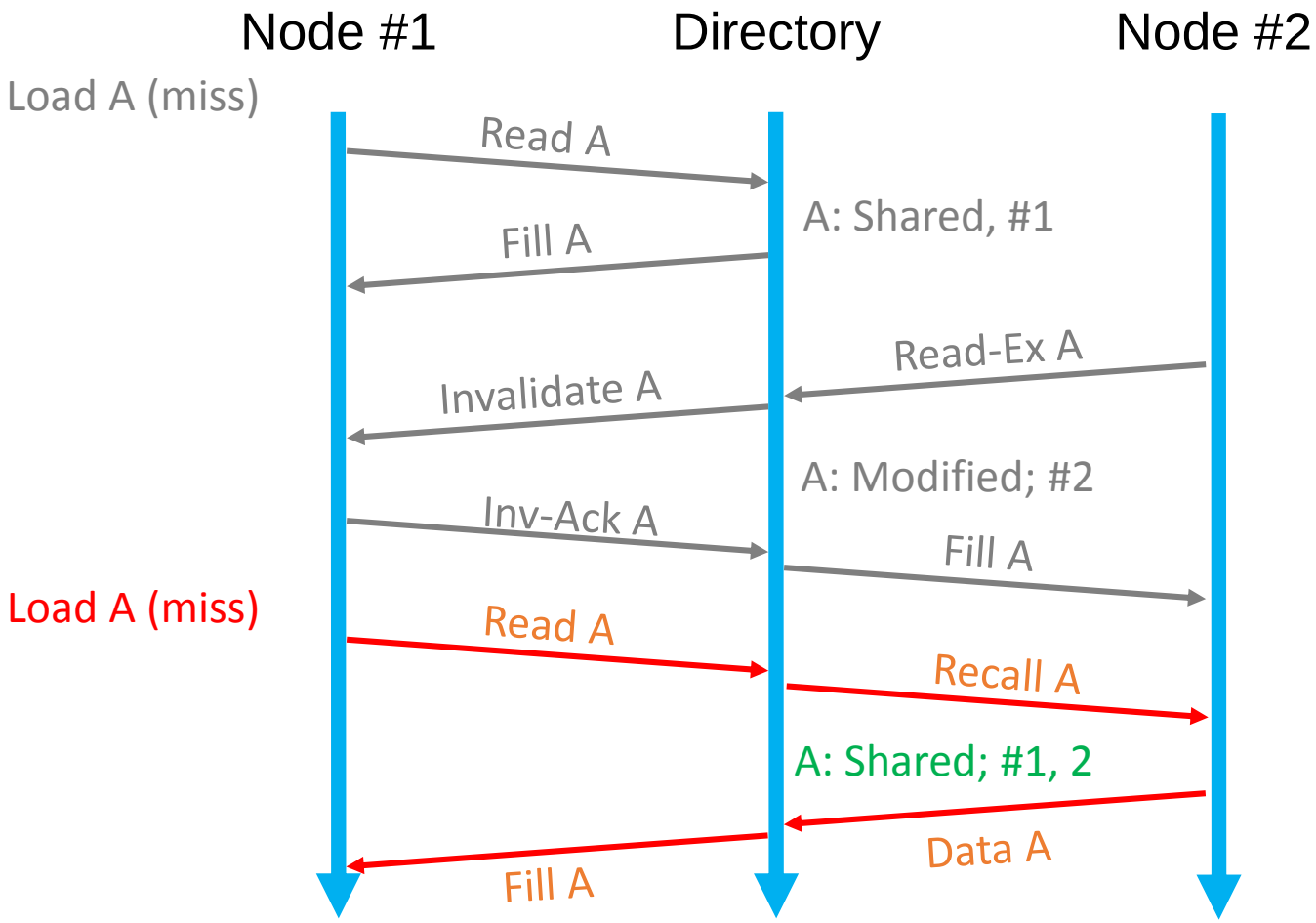
Basic Operation: Read Clean Data



Basic Operation: Write



Basic Operation: Read Dirty Data



Centralized vs. Distributed Directory

- ***Centralized***: Single directory tracks sharing state for all memory locations
  - ✓ Central serialization point: easy to get memory consistency
  - ✗ Not scalable (imagine traffic from 1000's of nodes…)
  - ✗ Directory size/organization changes with number of nodes
- ***Distributed:*** Distribute directory among multiple nodes
  - ✓ Scalable – directory size and BW grows with memory capacity
  - ✗ Directory can no longer serialize accesses across all addresses
    - Memory consistency becomes responsibility of CPU interface (more on this later)

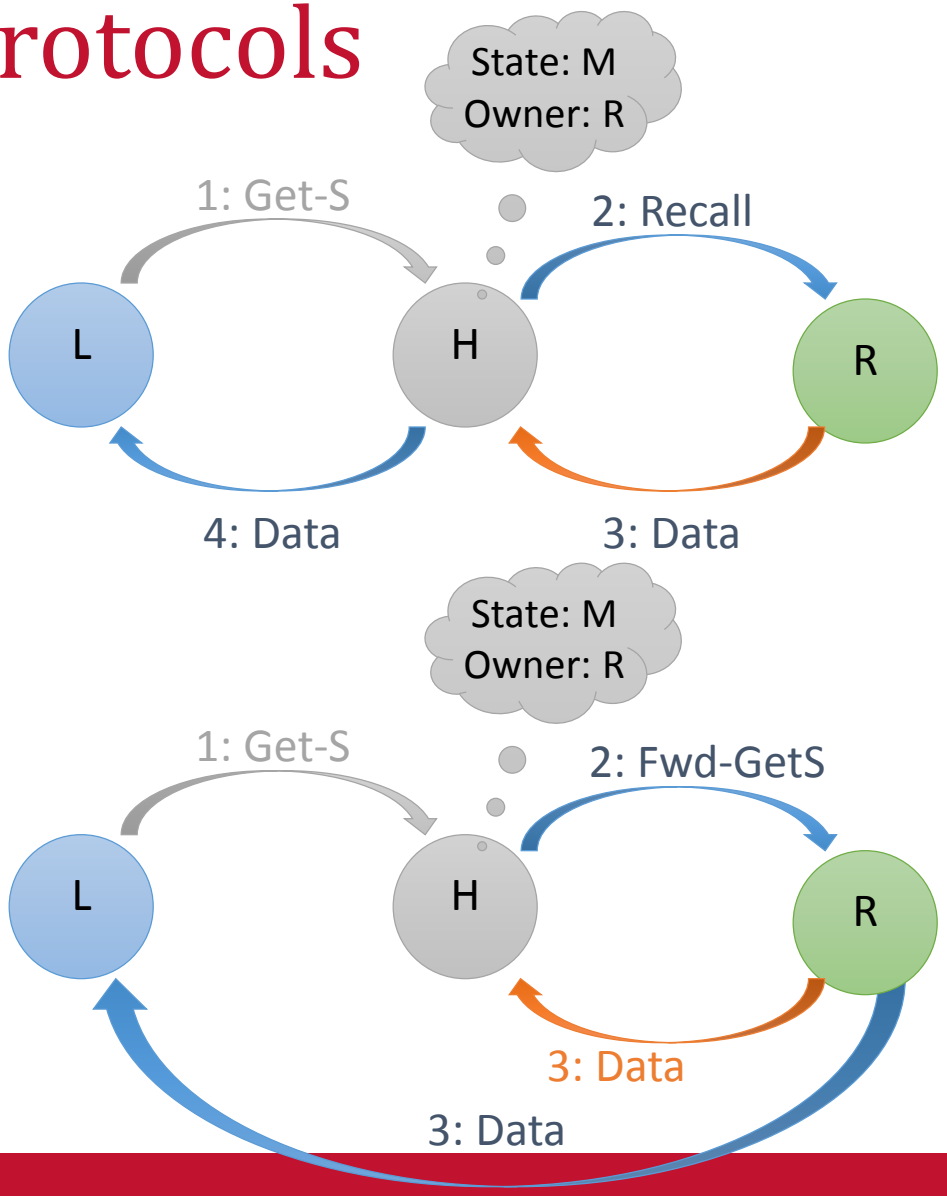
More Nomenclature

- ***Local Node (L)***
  - Node initiating the transaction we care about
- ***Home Node (H)***
  - Node where directory/main memory for the block lives
- ***Remote Node (R)***
  - Any other node that participates in the transaction

- 3-hop vs. 4-hop
  - Refers to the number of messages on the critical path of a transaction

4-hop vs. 3-hop Protocols

- Consider a cache miss
- L has a cache miss on a load instruction
  - Block was previously in modified state at R
- 3-hop protocols have higher performance but can be harder to get right



Main Challenges of Directory

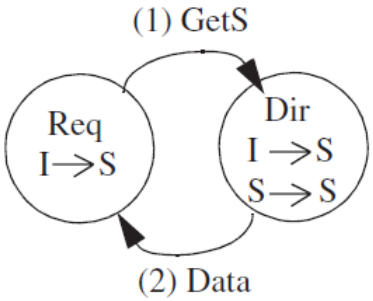
- More complex protocol (compared to snoopy)
  - Protocols have more message types
  - Transactions involve more messages
  - More actors should talk to each other to complete a transaction
- Deal with many possible race cases due to
  - Complex protocols
  - Complex network behavior (*e.g.*, network can deliver messages out of order)

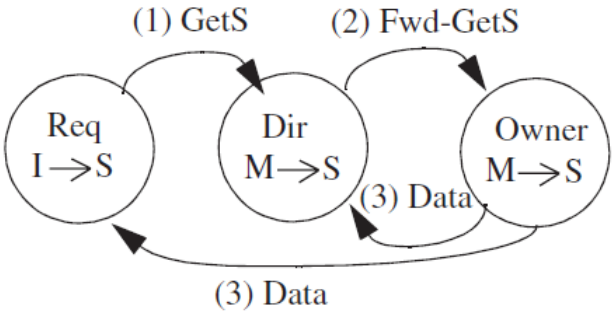
  → more transient states
- How to provide write serialization for coherence?
  - Directory acts as the point of serialization
  - Has to make sure anyone sees the writes in the same order as directory does
- Avoid deadlocks and livelocks
  - More difficult due to protocol and network complexity

Example: A 3-Hop MSI Protocol

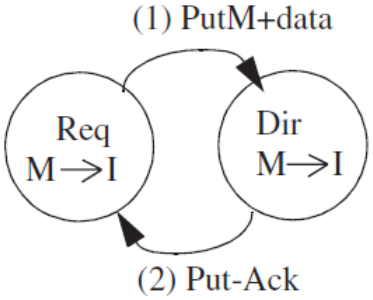
- Same three stable states as before: **M, S and I**
  - Directory owns a block unless in M state
  - Directory entry contains: stable coherence state, owner (if in M), sharers list (if in S)
- Transactions
  - GetS(hared), GetM(odified), PutM(odified), PutS(hared)
- Messages
  - GetS, GetM, PutM, PutS,
    Fwd-GetS, Fwd-GetM, Inv,
    Put-Ack, Inv-Ack,
    Data (from Dir or from Owner)
- Separate logical networks for Reqs, Forwarded Reqs and Reps
  - Networks can be physically separate, or,
  - Use Virtual Channels to share a physical network

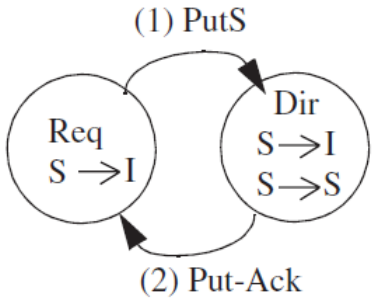
3-Hop MSI High-Level Spec (1/2)





I → S

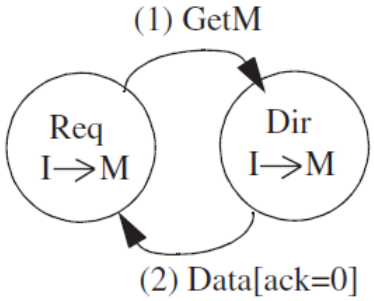


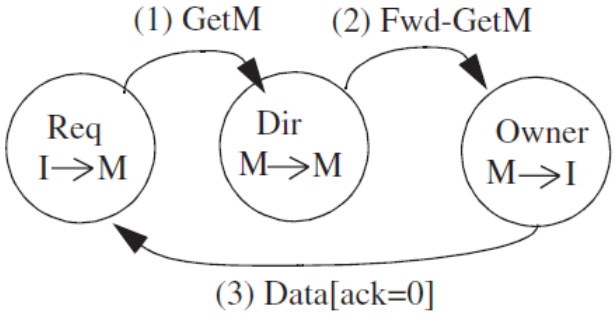


M or S → I

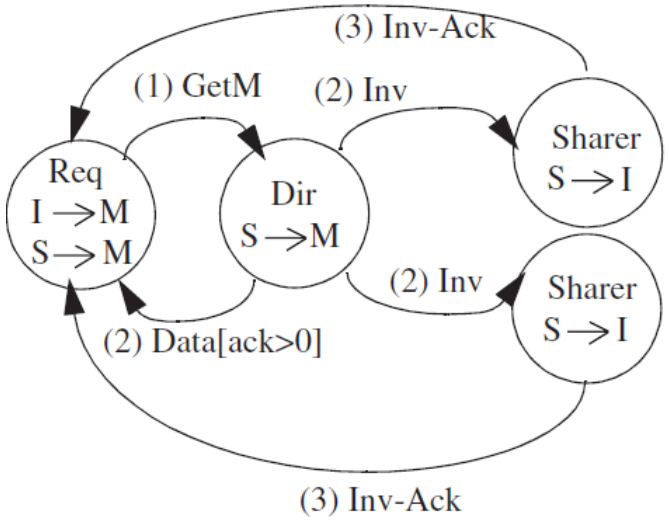
3-Hop MSI High-Level Spec (2/2)

- For S → M, dir sends the ***AckCount*** to the requestor
  - AckCount = # sharers
- Requestor collects the Inv-Acks





I or S → M



3-Hop MSI Detailed Spec (1/2)

	load	store	replacement	Fwd-GetS	Fwd-GetM	Inv	Put-Ack	Data from Dir (ack=0)	Data from Dir (ack>0)	Data from Owner	Inv-Ack	Last-Inv-Ack
I	send GetS to Dir/IS^D	send GetM to Dir/IM^{AD}										
IS^D	stall	stall	stall			stall		-/S		-/S		
IM^{AD}	stall	stall	stall	stall	stall			-/M	-/IM^A	-/M	ack--	
IM^A	stall	stall	stall	stall	stall						ack--	-/M
S	hit	send GetM to Dir/SM^{AD}	send PutS to Dir/SI^A			send Inv-Ack to Req/I						
SM^{AD}	hit	stall	stall	stall	stall	send Inv-Ack to Req/IM^{AD}		-/M	-/SM^A	-/M	ack--	
SM^A	hit	stall	stall	stall	stall						ack--	-/M
M	hit	hit	send PutM+data to Dir/MI^A	send data to Req and Dir/S	send data to Req/I							
MI^A	stall	stall	stall	send data to Req and Dir/SI^A	send data to Req/II^A		-/I					
SI^A	stall	stall	stall			send Inv-Ack to Req/II^A	-/I					
II^A	stall	stall	stall				-/I					

Source: A Primer on Memory Consistency and Cache Coherence

3-Hop MSI Detailed Spec (2/2)

	GetS	GetM	PutS-NotLast	PutS-Last	PutM+data from Owner	PutM+data from NonOwner	Data
I	send data to Req, add Req to Sharers/S	send data to Req, set Owner to Req/M	send Put-Ack to Req	send Put-Ack to Req		send Put-Ack to Req	
S	send data to Req, add Req to Sharers	send data to Req, send Inv to Sharers, clear Sharers, set Owner to Req/M	remove Req from Sharers, send Put-Ack to Req	remove Req from Sharers, send Put-Ack to Req/I		remove Req from Sharers, send Put-Ack to Req	
M	Send Fwd-GetS to Owner, add Req and Owner to Sharers, clear Owner/S^D	Send Fwd-GetM to Owner, set Owner to Req	send Put-Ack to Req	send Put-Ack to Req	copy data to memory, clear Owner, send Put-Ack to Req/I	send Put-Ack to Req	
S^D	stall	stall	remove Req from Sharers, send Put-Ack to Req	remove Req from Sharers, send Put-Ack to Req		remove Req from Sharers, send Put-Ack to Req	copy data to memory/S

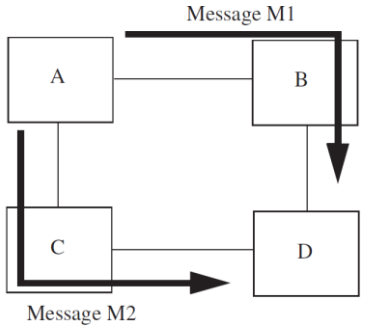
Source: A Primer on Memory Consistency and Cache Coherence

Clean Eviction Notification (PutS)

- Should directory learn when clean blocks are evicted?
- Advantages:
  - Allows directory to remove cache from sharers list
    - Avoids unnecessary invalidate messages in the future
    - Helps with providing line in E state in MESI protocol
    - Helps with limited-pointer directories (in a few slides)
  - Simplifies the protocol
- Disadvantages:
  - Notification traffic is unnecessary if block is re-read before anyone writes to it
    - Read-only data never invalidated (extra evict messages)

Ordered vs. Unordered Network

- A network is ***ordered*** if messages sent from *src* to *dst*, for any (*src*, *dst*) pair, are always delivered in order
  - Otherwise, the network is ***unordered***
  - e.g., adaptive routing can make a network unordered



- So far, we've assumed ordered networks
- Unordered networks can cause more races

- Example: consider re-ordered PutM-Ack and Fwd-GetM during a "writeback/store" race in the previous MSI protocol

Directory Implementation

Sharer-List Representation

- How to keep track of the sharers of a cache line?
- ***Full Bit Vector*** scheme: One bit of directory memory per main-memory block per actor

0	1	1	0	0	0	0	1

  - Not scalable
  - List can be very long for very large (1000s of nodes) systems
  - Searching the list to find the sharers can become an overhead
- ***Coarse Bit Vector*** scheme: Each bit represents multiple sharers

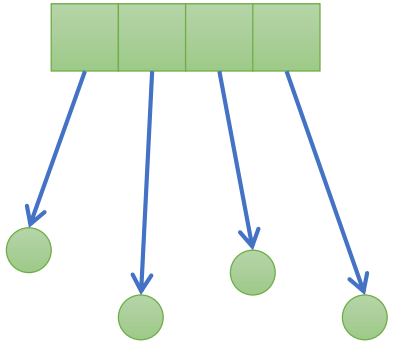
1	1	0	1

  - Reduces overhead by a constant factor
  - Still not very scalable

Limited-Pointer Schemes (1/3)

- ***Observation***: each cache line is very often only shared by a few actors

→ Only store a few pointers per cache line



- ***Overflow strategy***: what to do when there are more sharers than pointers?
  - Many different solutions

Limited-Pointer Schemes (2/3)

- Classification: **Dir_{<num-pointers>}<Action-upon-overflow>**
- **Dir$_i$B** (B = Broadcast):
  - Beyond ***i*** pointers, set the inval-broadcast bit ON
  - Expected to do well since widely shared data is not written often
- **Dir$_i$NB** (NB = No Broadcast)
  - When sharers exceed ***i***, invalidate one of the existing sharers
  - Significant degradation for widely-shared, mostly-read data

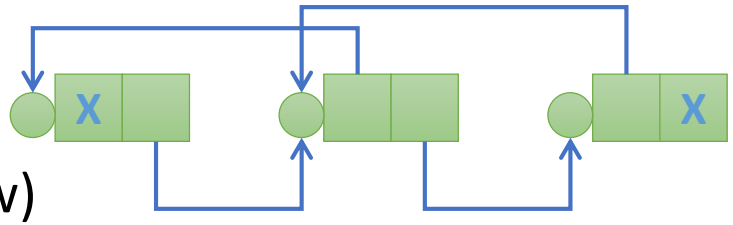
Limited-Pointer Schemes (3/3)

- **Dir$_i$CV$_r$** (CV = Coarse Vector)
  - When sharers exceed ***i***, use the bits as a *coarse vector*
    - ***r*** : # of actors that each bit in the coarse vector represents
  - Always results in less coherence traffic than **Dir$_i$B**
  - Example: **Dir$_3$CV$_4$** for 64 processors

- **Dir$_i$SW** (SW = Software)
  - When sharers exceed ***i***, trap to software
  - Software can maintain full sharer list in software-managed data structures
  - Trap handler needs full access to the directory controller

Linked-List Schemes (Historical)

- Each cache frame has fixed storage for next (prev) sharer
  - Directory has a pointer to the head of the list
  - Can be combined with limited-pointer schemes (to handle overflow)



- Variations:
  - Doubly-linked (Scalable Coherent Interconnect)
  - Singly-linked (S3.mp)

✘ Poor performance
  - Long invalidation latency
  - Replacements – difficult to get out of sharer list
    - Especially with singly-linked list… – how to do it?

✘ Difficult to verify

Directory Organization

- Logically, directory has an entry for every block of memory
  - How to implement this?
- ***Dense directory***: one dir entry per physical block
  - Merge dir controller with mem controller and store dir in RAM
  - Older implementations were like this (like SGI Origin)
  - Can use ECC bits to avoid adding extra DRAM chips for directory
- ✗ Drawbacks:
  - Shared accesses need checking directory in RAM
    - Slow and power-hungry
    - Even when access itself is served by cache-to-cache transactions
  - Most memory blocks not cached anywhere → waste of space
  - Example: 16 MB cache, 4 GB mem. → 99.6% idle

→ Does not make much sense for todays machines

Sparse Directories

- Solution: Cache the directory info in fast (on-chip) mem
  - Avoids most off-chip directory accesses
- **DRAM-backed directory cache**
  - On a miss should go to the DRAM directory
  - ✗ Still wastes space
  - ✗ Incurs DRAM writes on a dir cache replacement
- **Non-DRAM-backed directory cache**
  - A miss means the line is not cached anywhere
  - On a dir cache replacement, should invalidate the cache line corresponding to the replaced dir entry in the whole system
  - ✓ No DRAM directory access
  - ✗ Extra invalidations due to limited dir space
- **Null directory cache**
  - No directory entries → Similar to **Dir_0B**
  - All requests broadcasted to everyone
  - Used in AMD and Intel's inter-socket coherence (HyperTransport and QPI)
  - ✗ Not scalable but ✓ simple

Sharing and Cache-Invalidation Patterns

Cache Invalidation Patterns

- Hypothesis: On a write to a shared location, # of caches to be invalidated is typically small
- If not true, unicast (directory) is not much better than broadcast (snoopy)
- Experience tends to validate this hypothesis

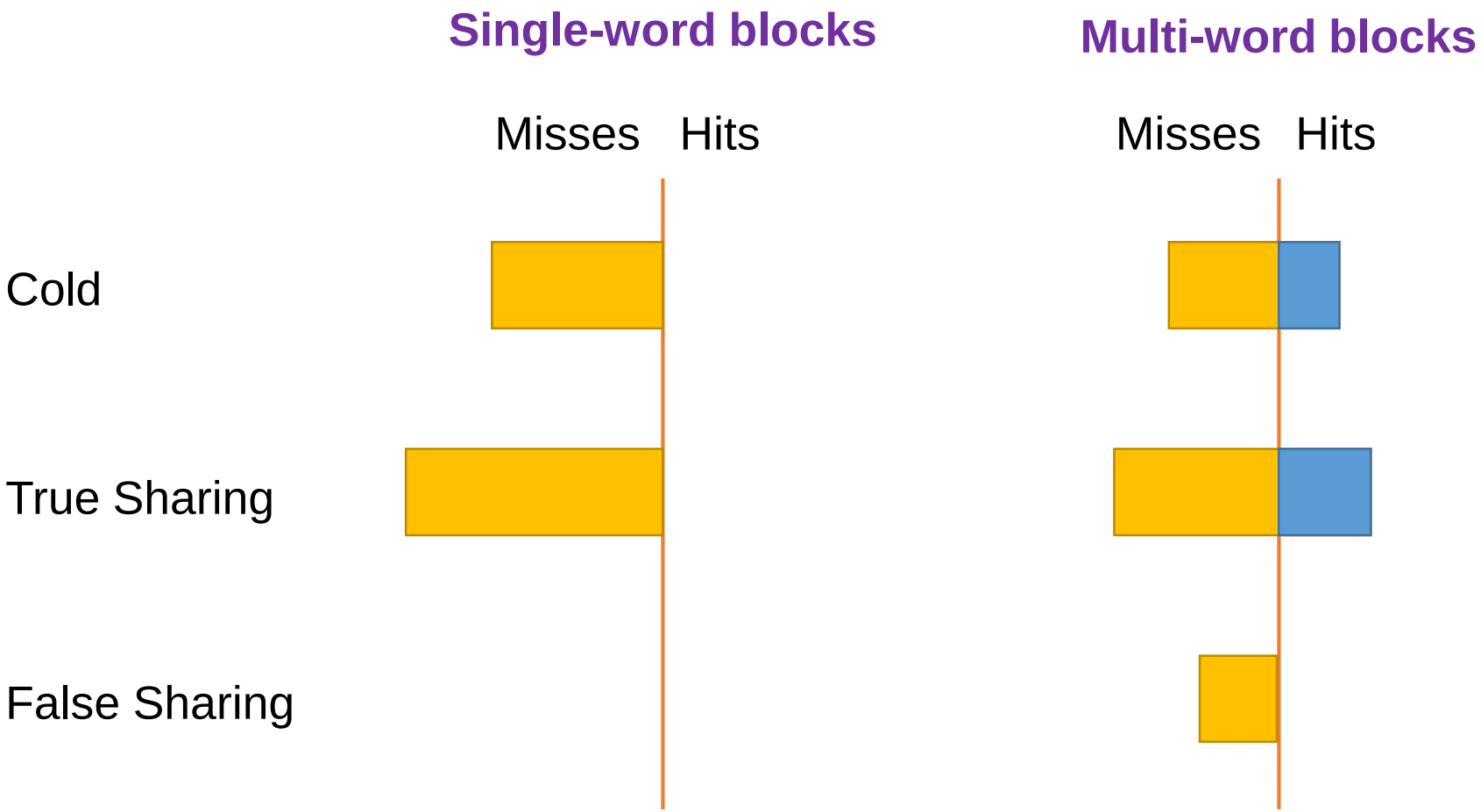
Common Sharing Patterns

- Code and read-only objects
  - No problem since rarely written
- Migratory objects
  - Even as number of caches grows, only 1-2 invalidations
- Mostly-read objects
  - Invalidations are expensive but infrequent, so OK
- Frequently read/written objects (e.g., task queues)
  - Invalidations frequent, hence sharer list usually small
- Synchronization objects
  - Low-contention locks result in few invalidations
  - High contention locks may need special hardware support or complex software design (next lecture)
- Badly-behaved objects

Misses on Shared Data

- Assume infinite caches
- Uniprocessor misses: *cold (compulsory)*
  - Ignoring *capacity* and *conflict* misses due to infinite cache
- Multiprocessing adds a new one: ***coherence misses***
  - When cache misses on an invalidated or without-sufficient-permission line
- Reasons for coherence misses
  - ***True sharing***
  - ***False sharing***
    - Due to prefetching effects of ***multi-word*** cache blocks

Effects of Multi-Word Cache Blocks



How to Improve

- By changing layout of shared variables in memory
- Reduce false sharing
  - Scalars with false sharing: put in different lines
  - Synchronization variables: put in different lines
  - Heap allocation: per-thread heaps vs. global heaps
  - Padding: padding structs to cache-line boundaries
- Improve the spatial locality of true sharing
  - Scalars protected by locks: pack them in the same line as the lock variable

Reducing Coherence Miss Cost

- By discovering the sharing pattern
  - In HW: similar to branch prediction, look for access patterns
  - SW hints: compiler/programmer knows the pattern and tells the HW
  - Hybrid of the two

Examples:

- Migratory pattern
  - On a remote read, self invalidate + pass in E state to requester
- Producer-Consumer pattern
  - Keep track of prior readers
  - Forward data to prior readers upon downgrade
- Last-touch prediction
  - Once an access is predicted as last touch, self-invalidate the line
  - Makes next processor's access faster: 3-hop → 2-hop