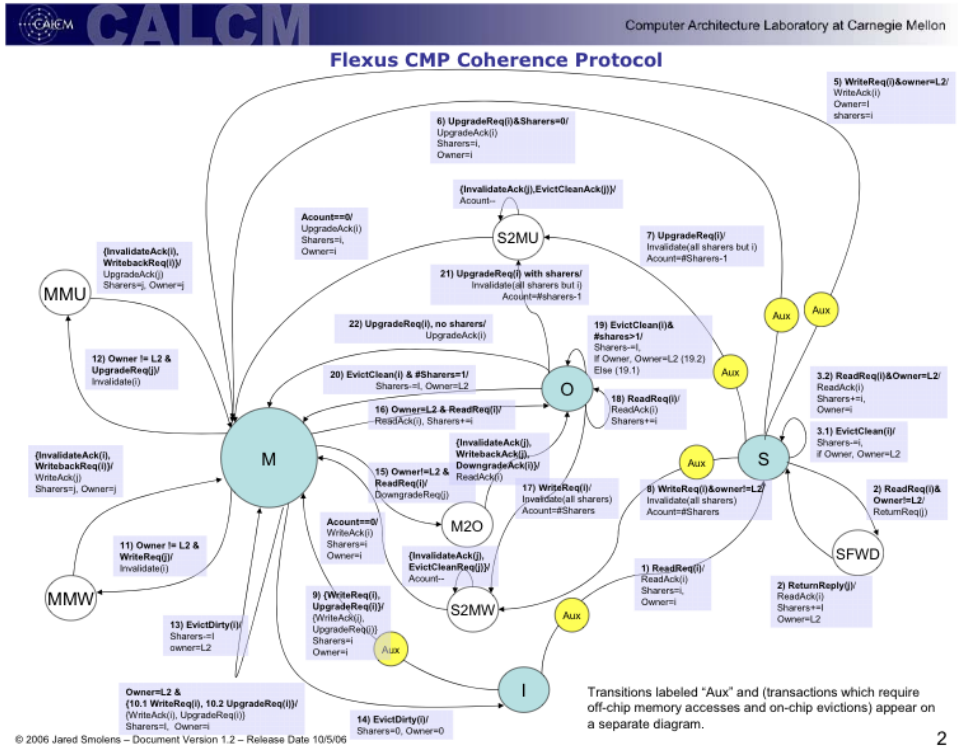
EECS 570

Lecture 9

Directory-based Coherence



Winter 2025

Prof. Satish Narayanasamy

http://www.eecs.umich.edu/courses/eecs570/

Slides developed in part by Profs. Adve, Falsafi, Hill, Lebeck, Martin, Narayanasamy, Nowatzyk, Reinhardt, Roth, Smith, Singh, and Wenisch.

Directory-Based Coherence

Scalable Cache Coherence

- **Scalable cache coherence**: two part solution

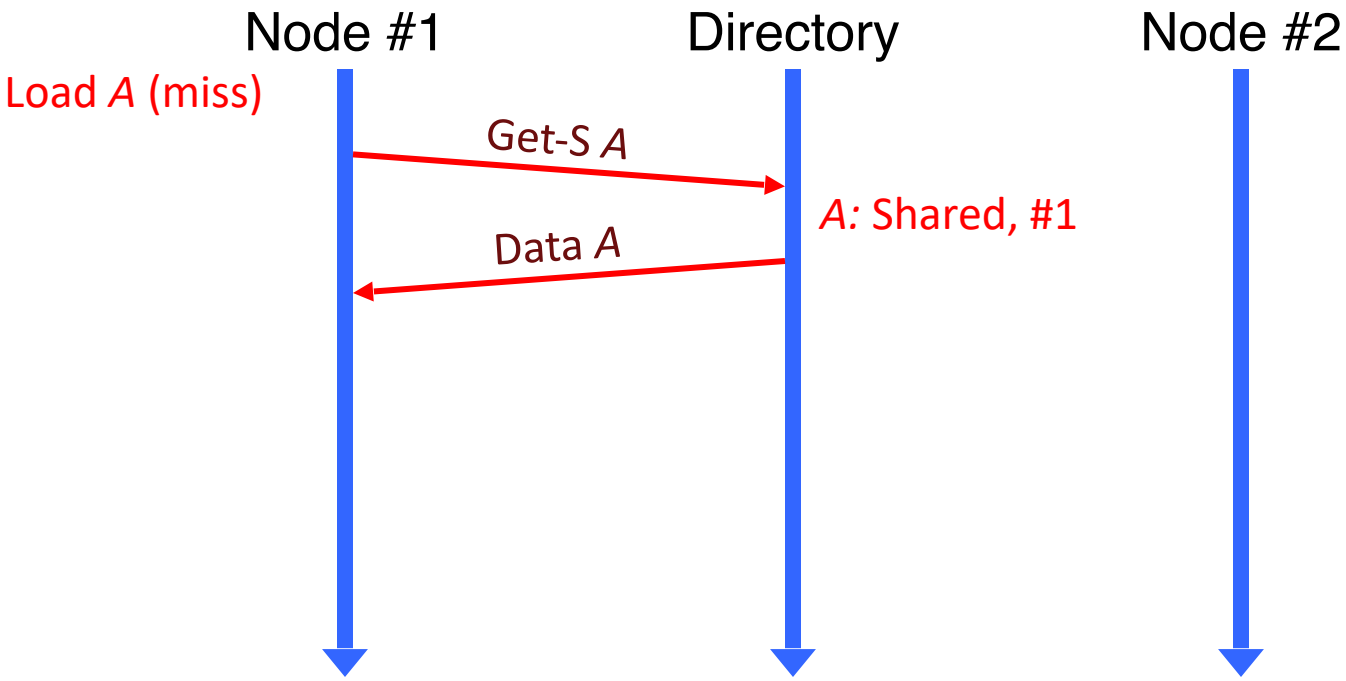
- Part I: **bus bandwidth**
  - Replace non-scalable bandwidth substrate (bus)…
  - …with scalable bandwidth one (point-to-point network, e.g., mesh)

- Part II: **processor snooping bandwidth**
  - Interesting: most snoops result in no action
  - Replace non-scalable broadcast protocol (spam everyone)…
  - …with scalable **directory protocol** (only spam processors that care)

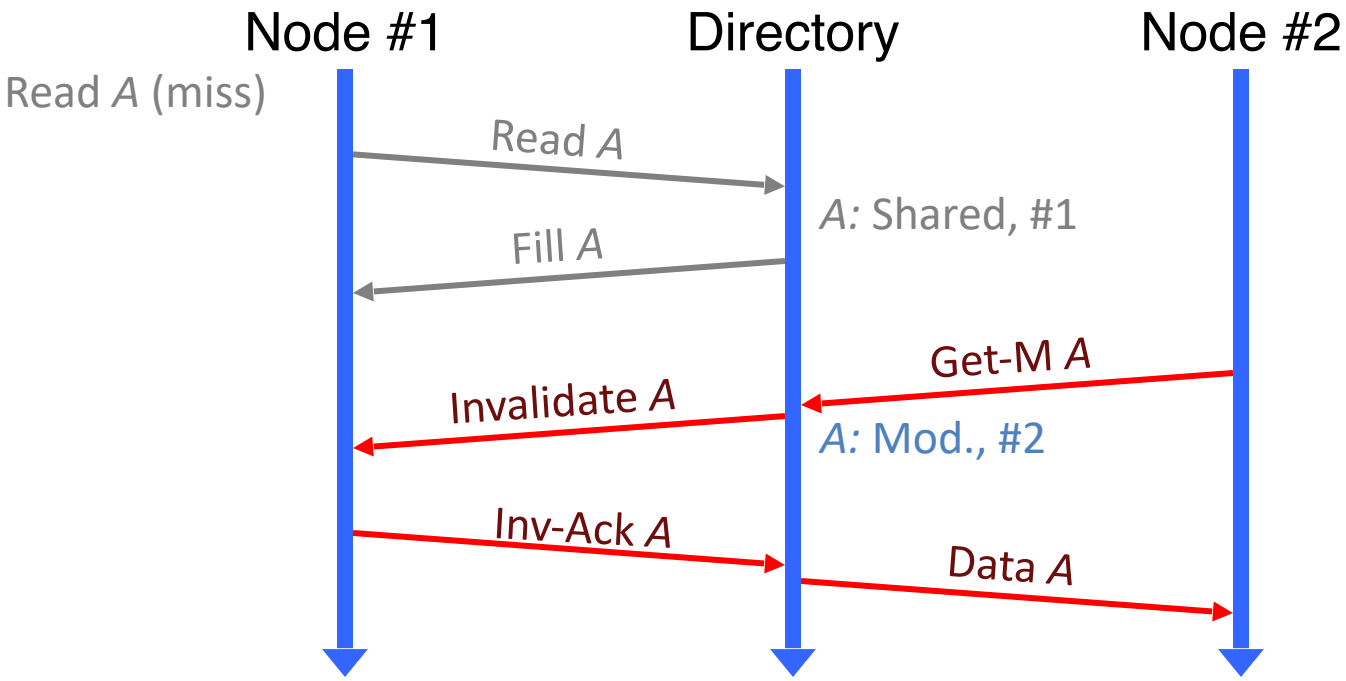
Directory Coherence Protocols

- Observe: physical address space statically partitioned
  - + Can easily determine which memory module holds a given line
    - That memory module sometimes called "**home**"
  - – Can't easily determine which processors have line in their caches
  - Bus-based protocol: broadcast events to all processors/caches
    - ± Simple and fast, but non-scalable
- **Directories**: non-broadcast coherence protocol
  - Extend memory to track caching information
  - For each physical cache line whose home this is, track:
    - **Owner**: which processor has a dirty copy (I.e., M state)
    - **Sharers**: which processors have clean copies (I.e., S state)
  - Processor sends coherence event to home directory
    - Home directory only sends events to processors that care

Basic Operation: Read



Basic Operation: Write

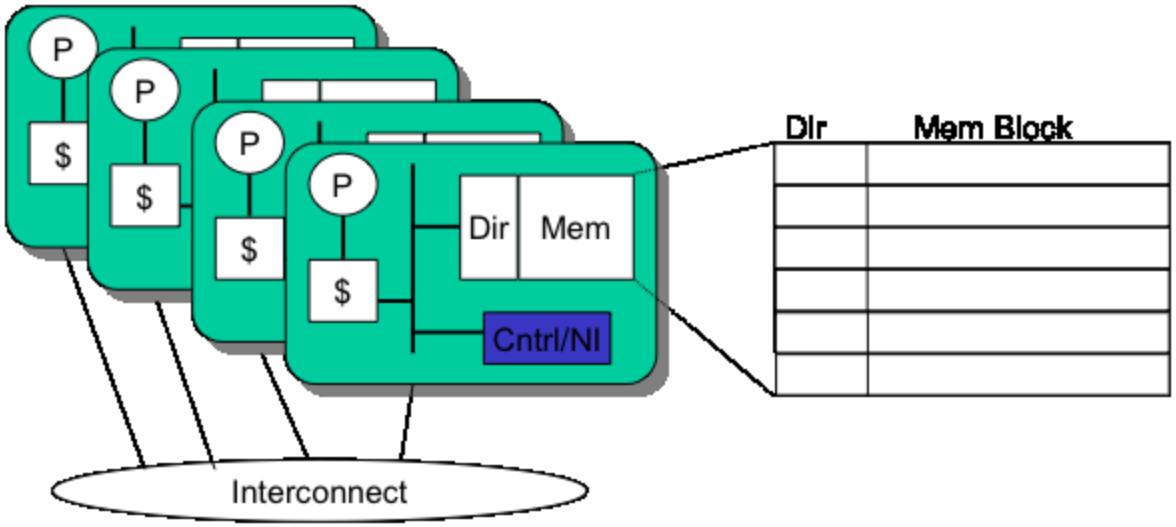


Centralized Directory

- Single directory contains a copy of cache tags from all nodes
- Advantages:
  - Central serialization point: easier to get memory consistency (just like a bus…)
- Problems:
  - Not scalable (imagine traffic from 1000's of nodes…)
  - Directory size/organization changes with number of nodes

Distributed Directory

- Distribute directory among memory modules
  - Memory block = coherence block (usually = cache line)
  - "Home node" → node with directory entry
  - Scalable – directory grows with memory capacity
    - Common trick: steal bits from ECC for directory state
  - Directory can no longer serialize accesses across all addresses
    - Memory consistency becomes responsibility of CPU interface



What is in the directory?

- Directory State
  - Invalid, Exclusive, Shared, … (“stable” states)
  - # outstanding invalidation messages, … (“transient” states)
- Pointer to exclusive owner
- Sharer list
  - List of caches that may have a copy
  - May include local node
  - Not necessarily precise, but always conservative

Directory State

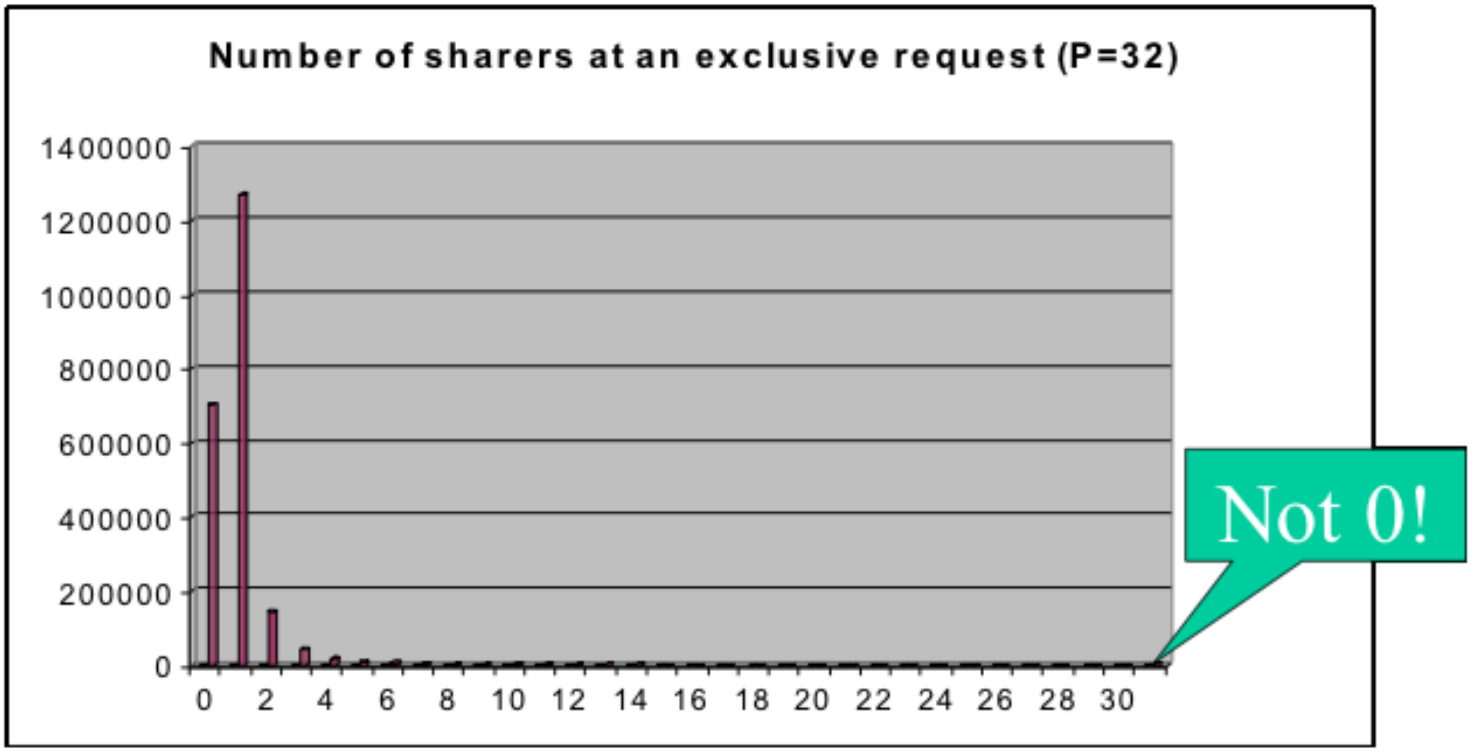
- Few stable states – 2-3 bits usually enough
- Transient states
  - Often 10's of states (+ need to remember node ids, …)
  - Transient state changes frequently, need fast RMW access
  - Design options:
    - Keep in directory: scalable (high concurrency), but slow
    - Keep in separate memory
    - Keep in directory, use cache to accelerate access
    - Keep in protocol controller
      - Transaction State Register File – like MSHRs

Pointer to Exclusive Owner

- Simple node id – $\log_2$ nodes
- Can share storage with sharer list (don't need both…)
- May point to a group of caches that internally maintain coherence (e.g., via snooping)
- May treat local node differently

Sharer List Representation

- Key to scalability – must efficiently represent node subsets
- Observation: most blocks cached by only 1 or 2 nodes
  - But, there are important exceptions (synchronization vars.)



OLTP workload

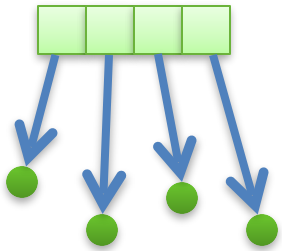
[Data from Nowatzyk]

Idea #1: Sharer Bit Vectors

- One bit per processor / node / cache
  - Storage requirement grows with system size

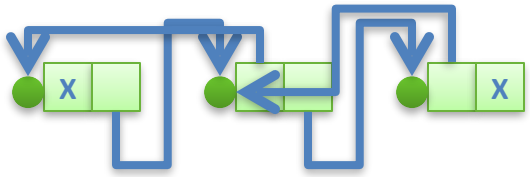
Idea #2: Limited Pointers

- Fixed number (e.g., 4) of pointers to node ids
- If more than *n* sharers:
  - Recycle one pointer (force invalidation)
  - Revert to broadcast
  - Handle in software (maintain longer list elsewhere)



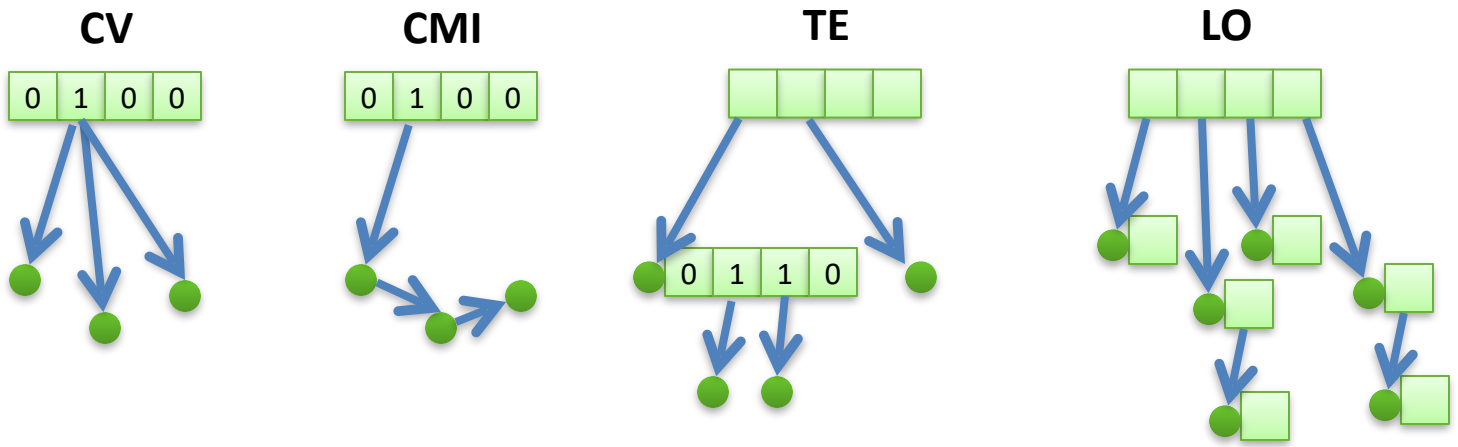
Idea #3: Linked Lists

- Each node has fixed storage for next (prev) sharer
- 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?



Directory representation optimizations

- Coarse Vectors (CV)
- Cruise Missile Invalidations (CMI)
- Tree Extensions (TE)
- List-based Overflow (LO)



Clean Eviction Notification

- Should directory learn when clean blocks are evicted?
- Advantages:
  - Avoids broadcast, frees pointers in limited pointer schemes
  - Avoids unnecessary invalidate messages
- Disadvantages:
  - Read-only data never invalidated (extra evict messages)
  - Notification traffic may be unnecessary
  - New protocol races

Sparse Directories

- Most of memory is invalid; why waste directory storage?
- Instead, use a directory cache
  - Any address w/o an entry is invalid
  - If full, need to evict & invalidate a victim entry
  - Generally needs to be highly associative

Cache Invalidation Patterns

- Hypothesis: On a write to a shared location, # of caches to be invalidated is typically small
- If this isn't true, directory is no better than broadcast/snoop
- Experience tends to validate this hypothesis

Common Sharing Patterns

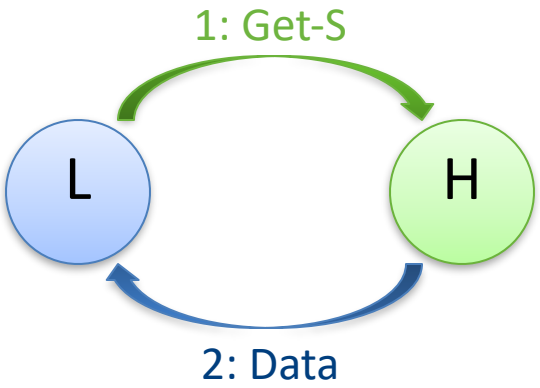
- Code and read-only objects
  - No problem since rarely written
- Migratory objects
  - Even as number of caches grows, only 1-2 invalidations
- Mostly-read objects
  - Invalidations are expensive but infrequent, so OK
- Frequently read/written objects (e.g., task queues)
  - Invalidations frequent, hence sharer list usually small
- Synchronization objects
  - Low-contention locks result in few invalidations
  - High contention locks need to have good coherence performance (e.g. MCS)
- Badly-behaved objects

Designing a Directory Protocol: 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

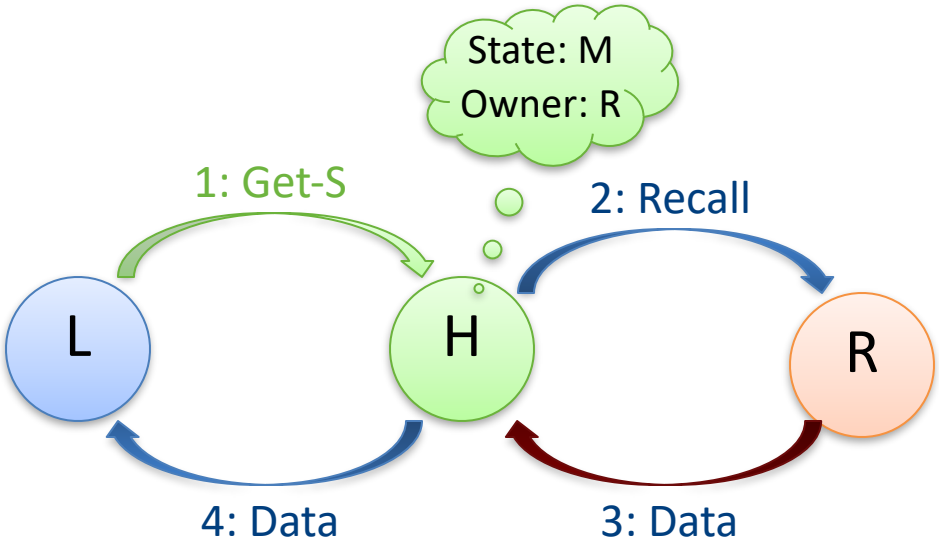
Read Transaction

- L has a cache miss on a load instruction



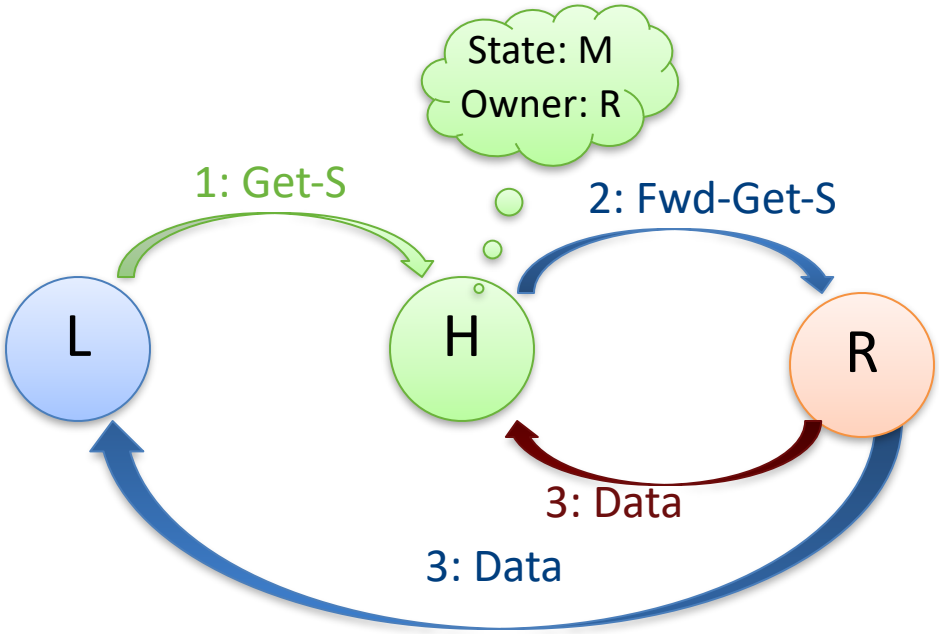
4-hop Read Transaction

- L has a cache miss on a load instruction
  - Block was previously in modified state at R



3-hop Read Transaction

- L has a cache miss on a load instruction
  - Block was previously in modified state at R



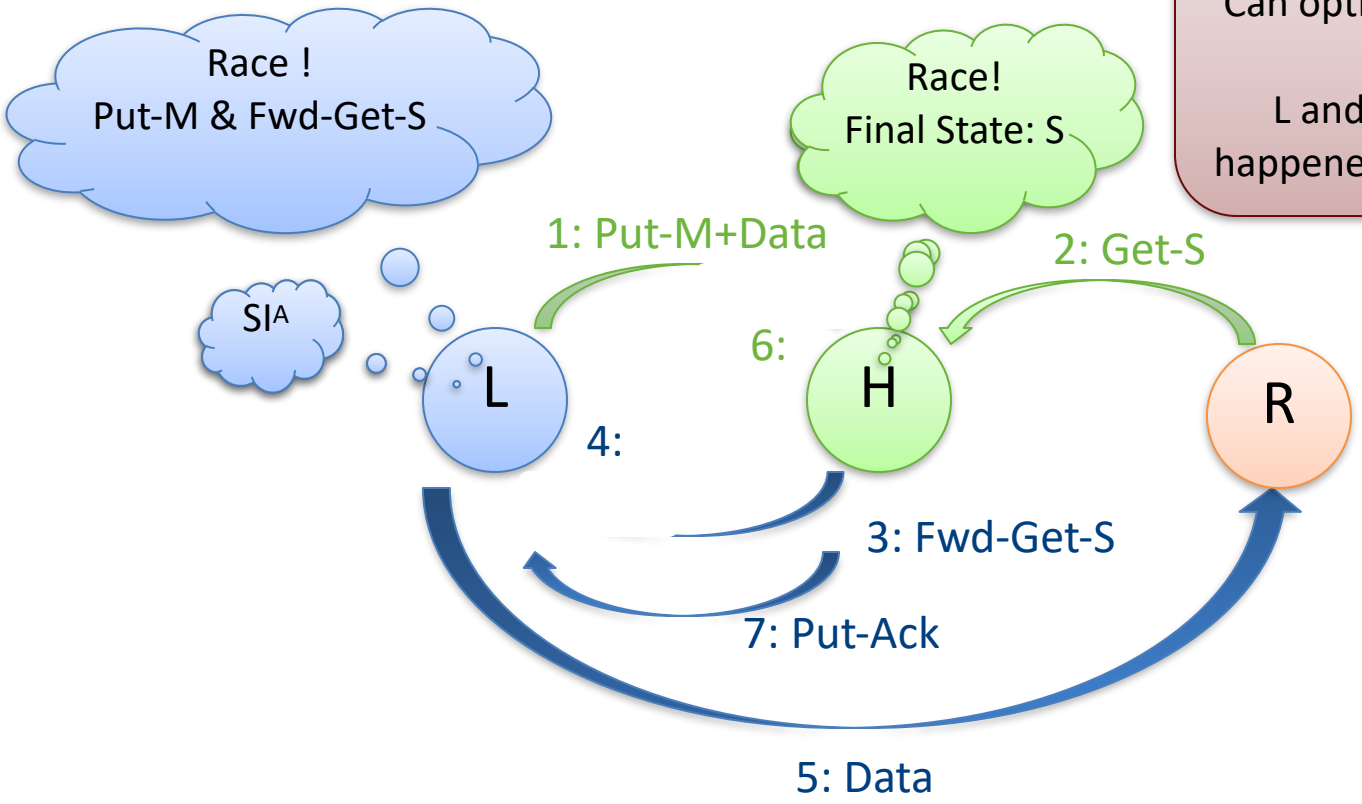
An Example Race: Writeback & Read

- L has dirty copy, wants to write back to H
- R concurrently sends a read to H

To make your head really hurt:

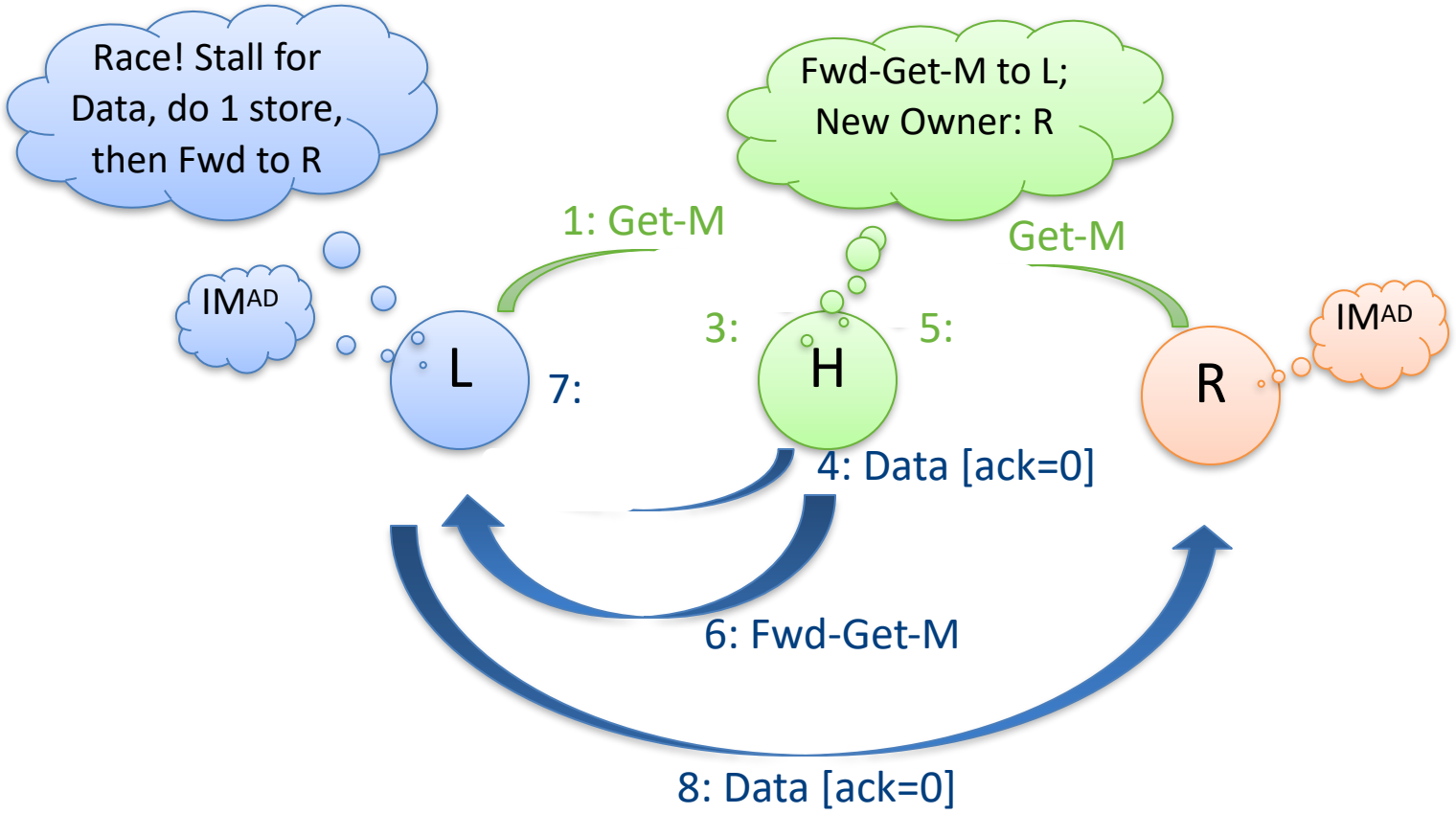
Can optimize away SI^A & Put-Ack!

L and H each know the race happened, don't need more msgs.



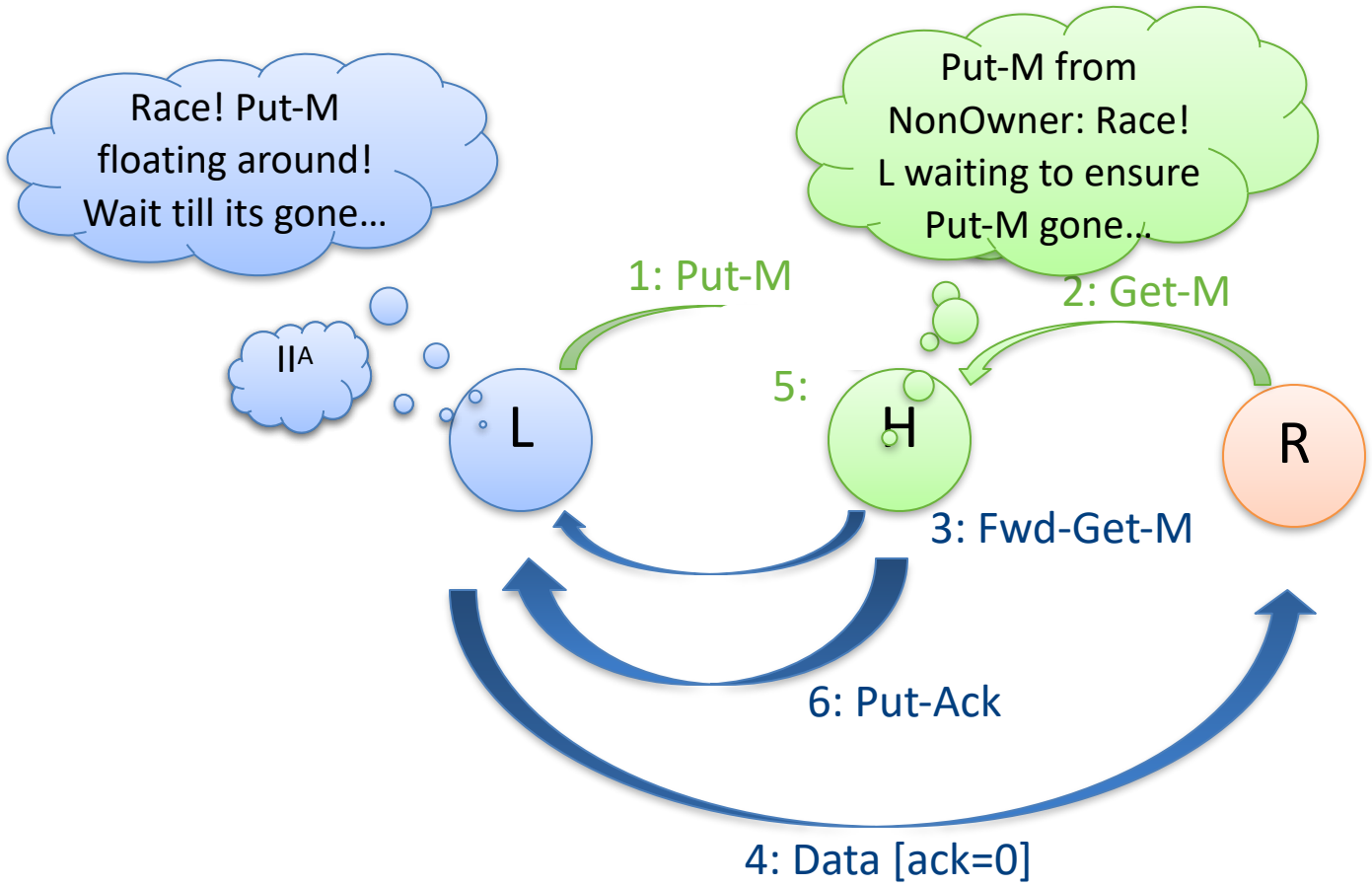
Store-Store Race

- Line is invalid, both L and R race to obtain write permission



Another store-store race

- L evicts dirty copy, R concurrently seeks write permission



Design Principles

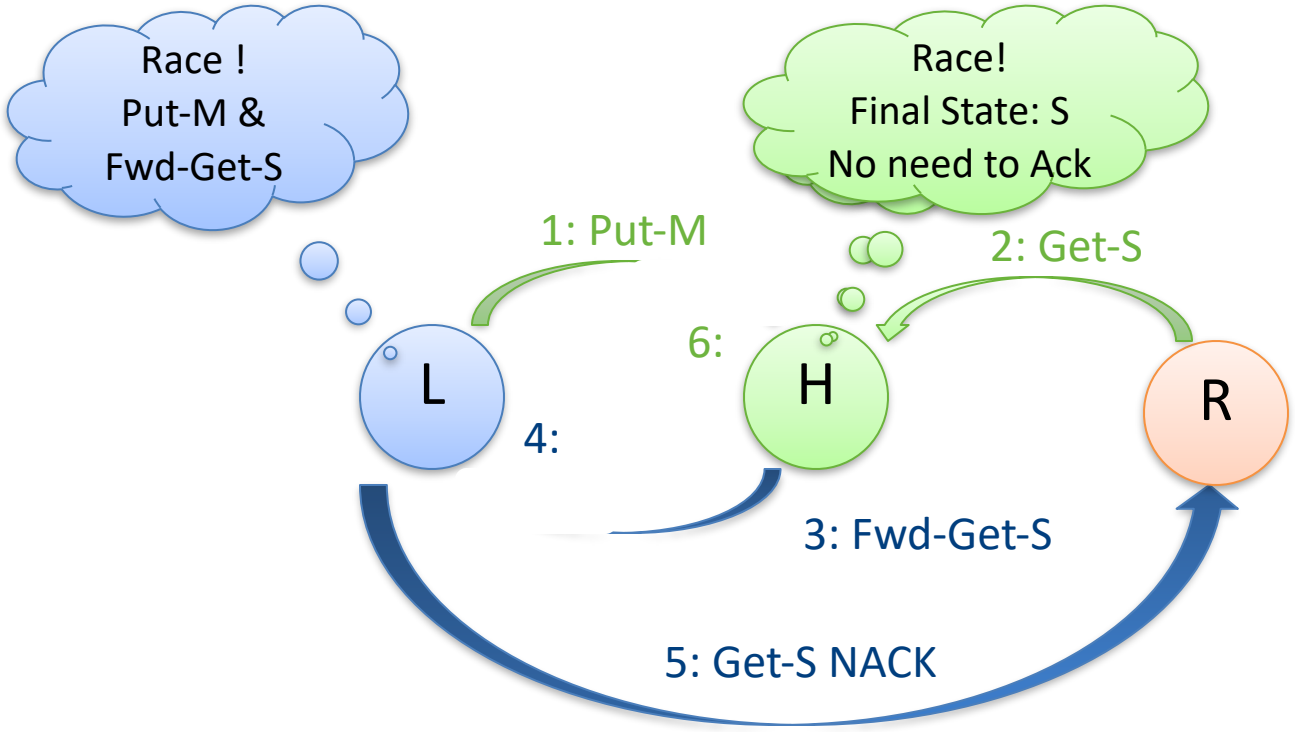
- Think of sending and receiving messages as separate events
- At each "step", consider what new requests can occur
  - E.g., can a new writeback overtake an older one?
- Two messages traversing same direction implies a race
  - Need to consider both delivery orders
    - Usually results in a "branch" in coherence FSM to handle both orderings
  - Need to make sure messages can't stick around "lost"
    - Every request needs an ack; extra states to clean up messages
  - Often, only one node knows how a race resolves
    - Might need to send messages to tell others what to do

CC Protocol Scorecard

- Does the protocol use negative acknowledgments (retries)?
- Is the number of active messages (sent but unprocessed) for one transaction bounded?
- Does the protocol require clean eviction notifications?
- How/when is the directory accessed during transaction?
- How many lanes are needed to avoid deadlocks?

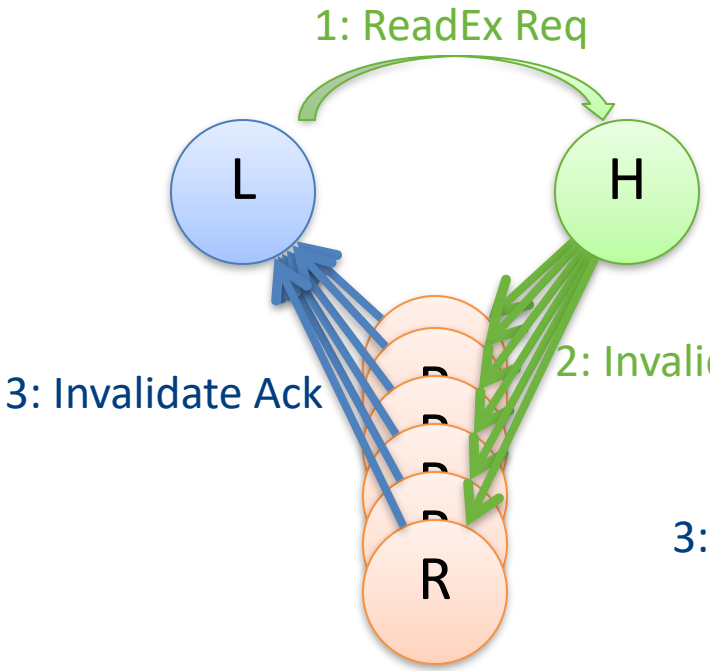
NACKs in a CC Protocol

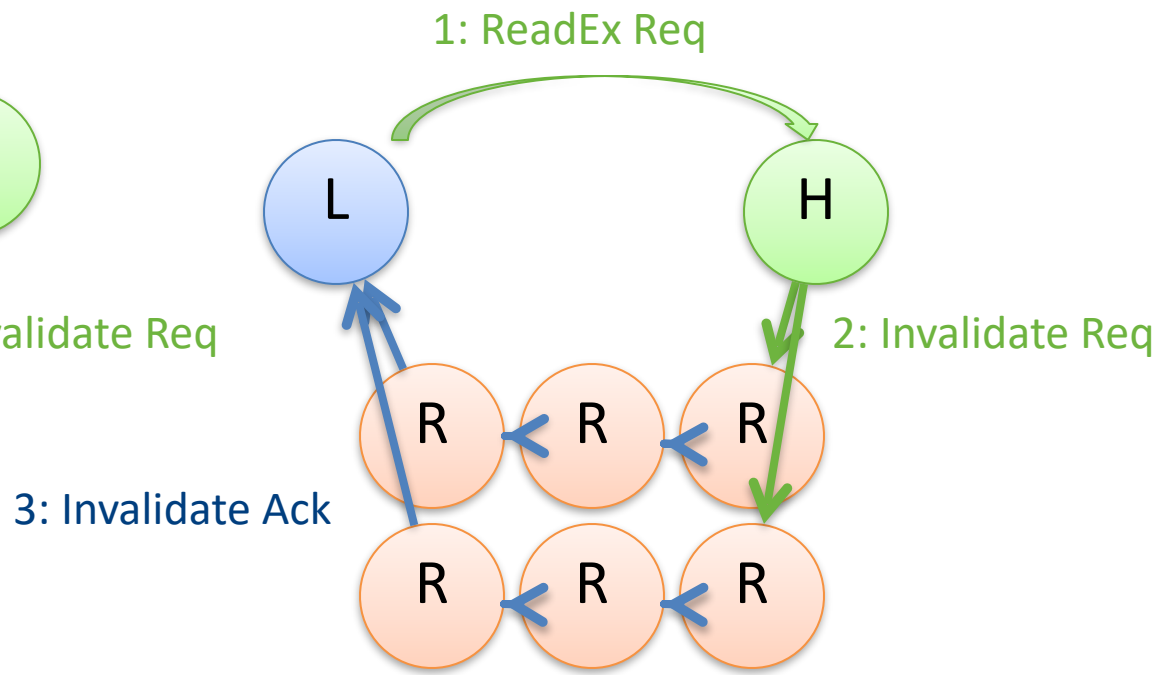
- Issues: Livelock, Starvation, Fairness
- NACKs as a flow control method ("home node is busy")
  - Really bad idea…
- NACKs as a consequence of protocol interaction…



Bounded # Msgs / Transaction

- Scalability issue: how much queue space is needed
- Coarse-vector vs. cruise-missile invalidation



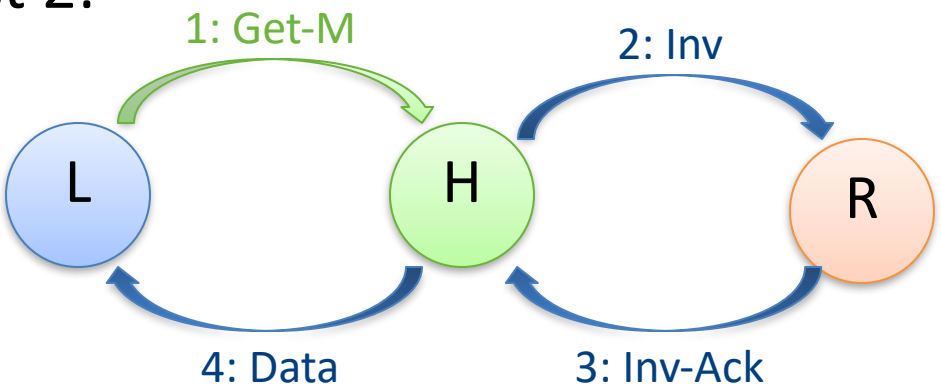


Frequency of Directory Updates

- How to deal with transient states?
  - Keep it in the directory: unlimited concurrency
  - Keep it in a pending transaction buffer (e.g., transaction state register file): faster, but limits pending transactions
- Occupancy free: Upon receiving an unsolicited request, can directory determine final state solely from current state?

Required # of lanes

- Need at least 2:

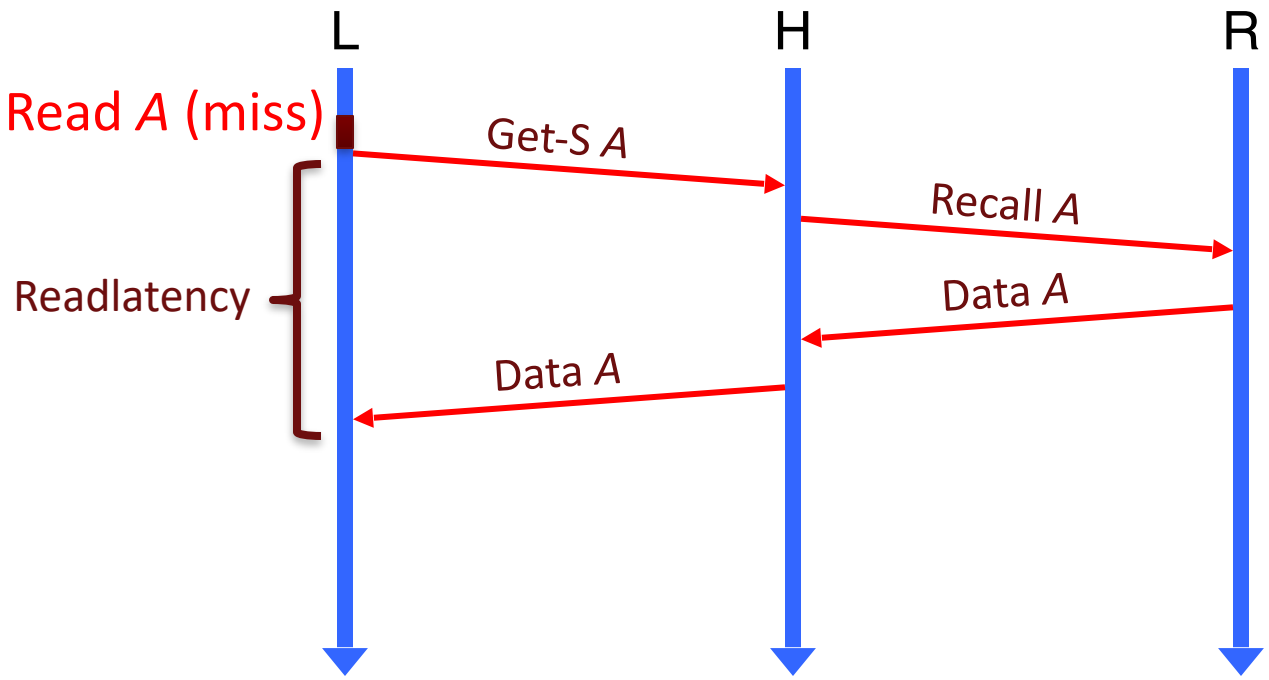


- More may be needed by I/O, complex forwarding
- How to assign lane to message type?
  - Secondary (forced) requests must not be blocked by new requests
  - Replies (completing a pending transaction) must not be blocked by new requests

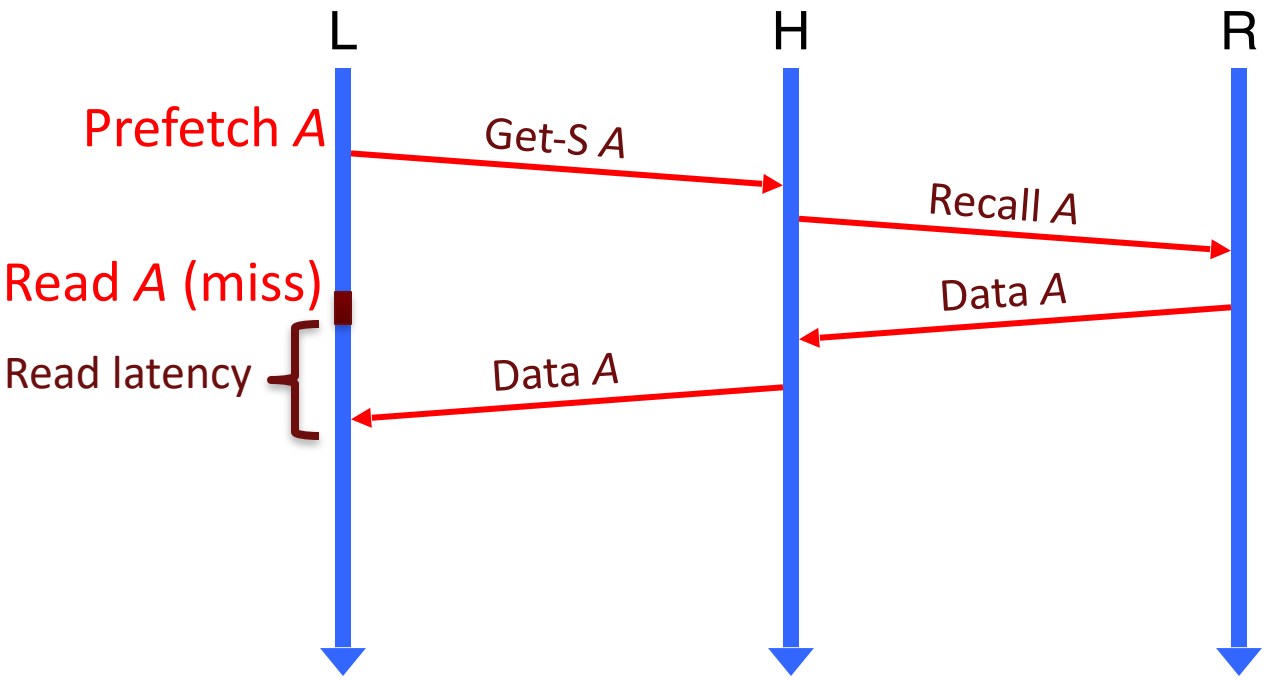
Some more guidelines

- All messages should be ack'd (requests elicit replies)
- Maximum number of potential concurrent messages for one transaction should be small and constant (i.e., independent of number of nodes in system)
- Use context information to avoid NACKs

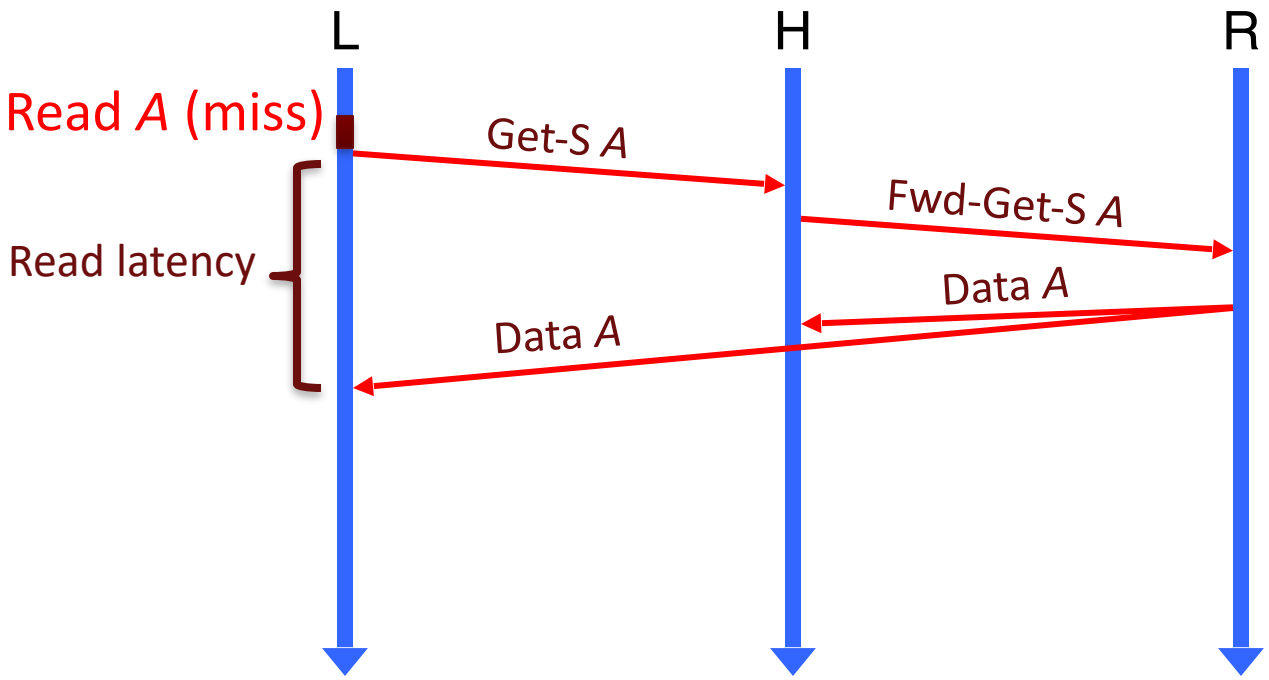
Optimizing coherence protocols



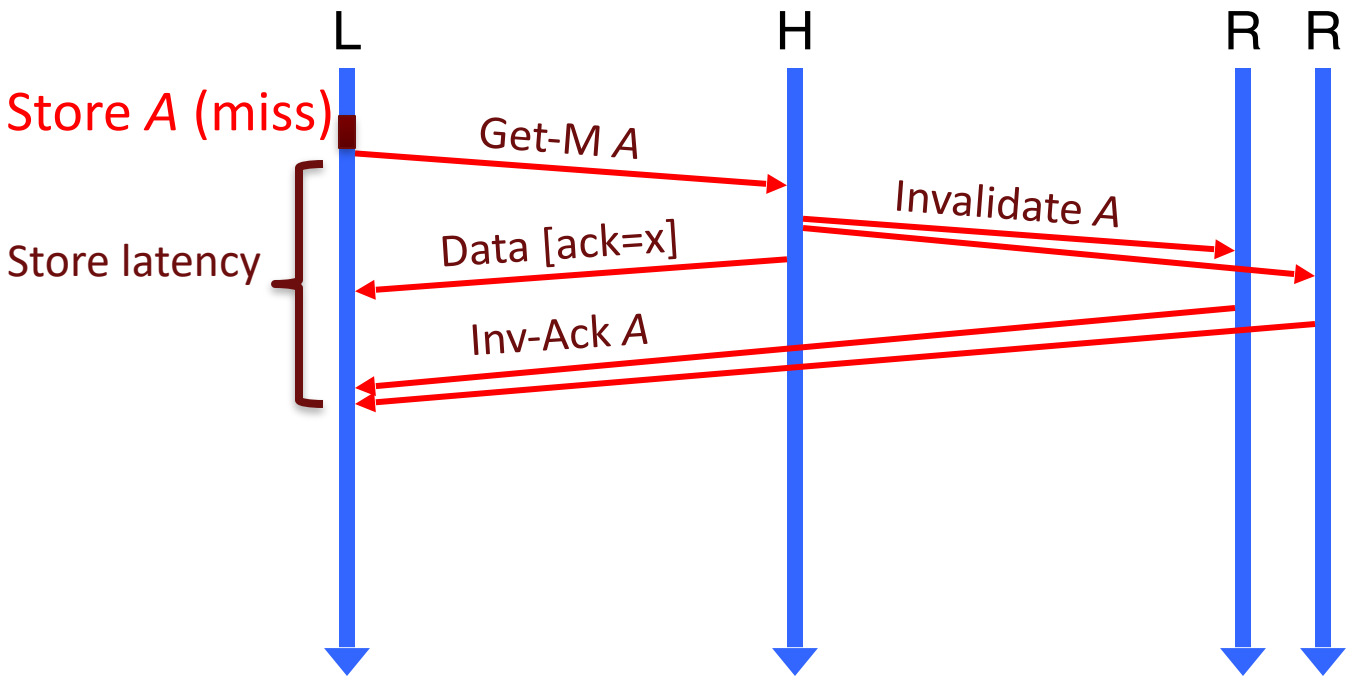
Prefetching



3-hop reads



3-hop writes


L
H
R
R
Store A (miss)
Get-M A
Invalidate A
Store latency
Data [ack=x]
Inv-Ack A

Migratory Sharing

Node 1	Node 2	Node 3
Read X		
Write X		
		Read X
		Write X
	Read X	
	Write X	

- Each Read/Write pair results in read miss + upgrade miss
- Coherence FSM can detect this pattern
  - Detect via back-to-back read-upgrade sequences
  - Transition to “migratory M” state
  - Upon a read, invalidate current copy, pass in “mig E” state

Producer Consumer Sharing

Node 1	Node 2	Node 3
Read X		
Write X		
		Read X
	Read X	
Read X		
Write X		
		Read X
	Read X	

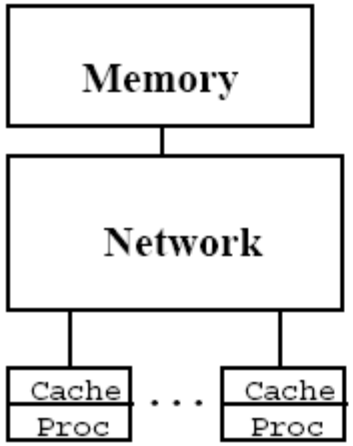
- Upon read miss, downgrade instead of invalidate
  - Detect because there are 2+ readers between writes
  - O state can help reduce number of writebacks
- More sophisticated optimizations
  - Keep track of prior readers
  - Forward data to all readers upon downgrade

Shortcomings of Protocol Optimizations

- Optimizations built directly into coherence state machine
  - Complex! Adds more transitions, races
  - Hard to verify even basic protocols
  - Each optimization contributes to state explosion
  - Can target only simple sharing patterns
  - Can learn only one pattern per address at a time

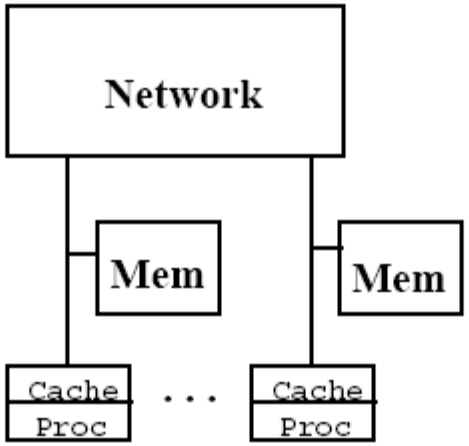
Cache Only Memory Architecture (COMA)

Big Picture



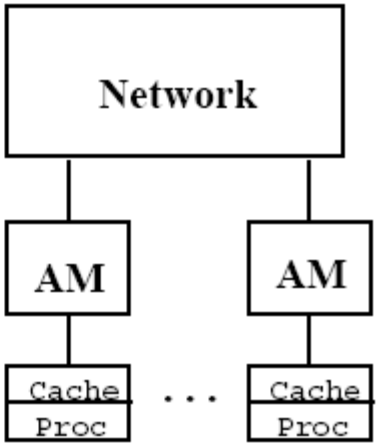
Shared Memory (UMA)

- Centralized shared memory
- Uniform access



Shared Memory (NUMA)

- Distributed Shared memory
- Non-uniform access latency



Cache Only Memory (COMA)

- No notion of "home" node; data moves to wherever it is needed
- Individual memories behave like caches

Cache Only Memory Architecture (COMA)

- Make all memory available for migration/replication
- All memory is DRAM cache called attraction memory

- Example systems
  - Data Diffusion Machine
  - KSR-1 (hierarchical snooping via ring interconnects)
  - Flat COMA (fixed home node for directory, but not data)

- Key questions:
  - How to find data?
  - How to deal with replacements?
  - Memory overhead

COMA Alternatives

- Flat-COMA
  - Blocks (data) are free to migrate
  - Fixed directory location (home node) for a physical address
- Simple-COMA
  - Allocation managed by OS and done at page granularity
- Reactive-NUMA
  - Switches between Simple-COMA and NUMA with remote cache on per-page basis