

18-447 Lecture 24: Cache Coherence

James C. Hoe

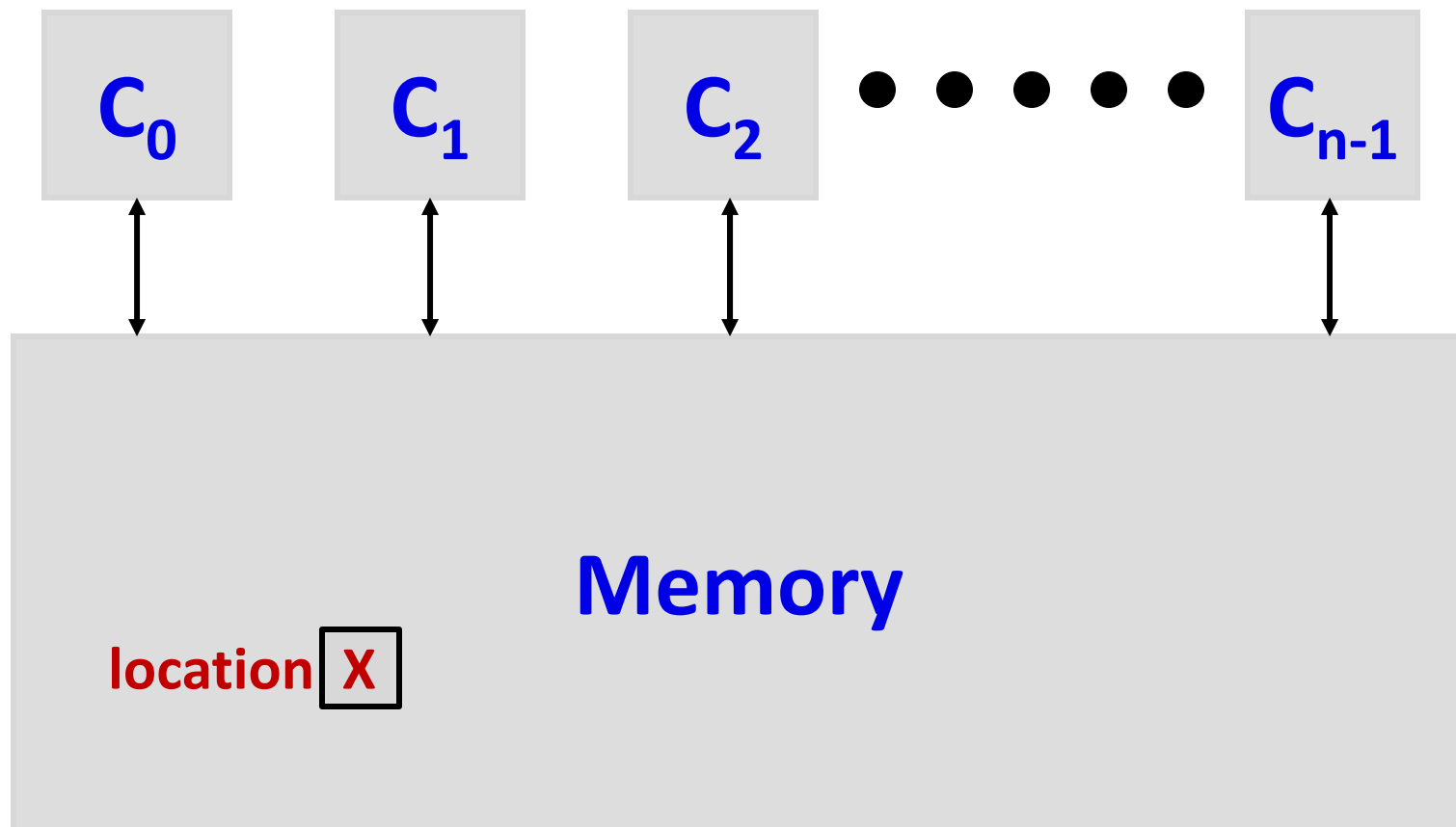
Department of ECE

Carnegie Mellon University

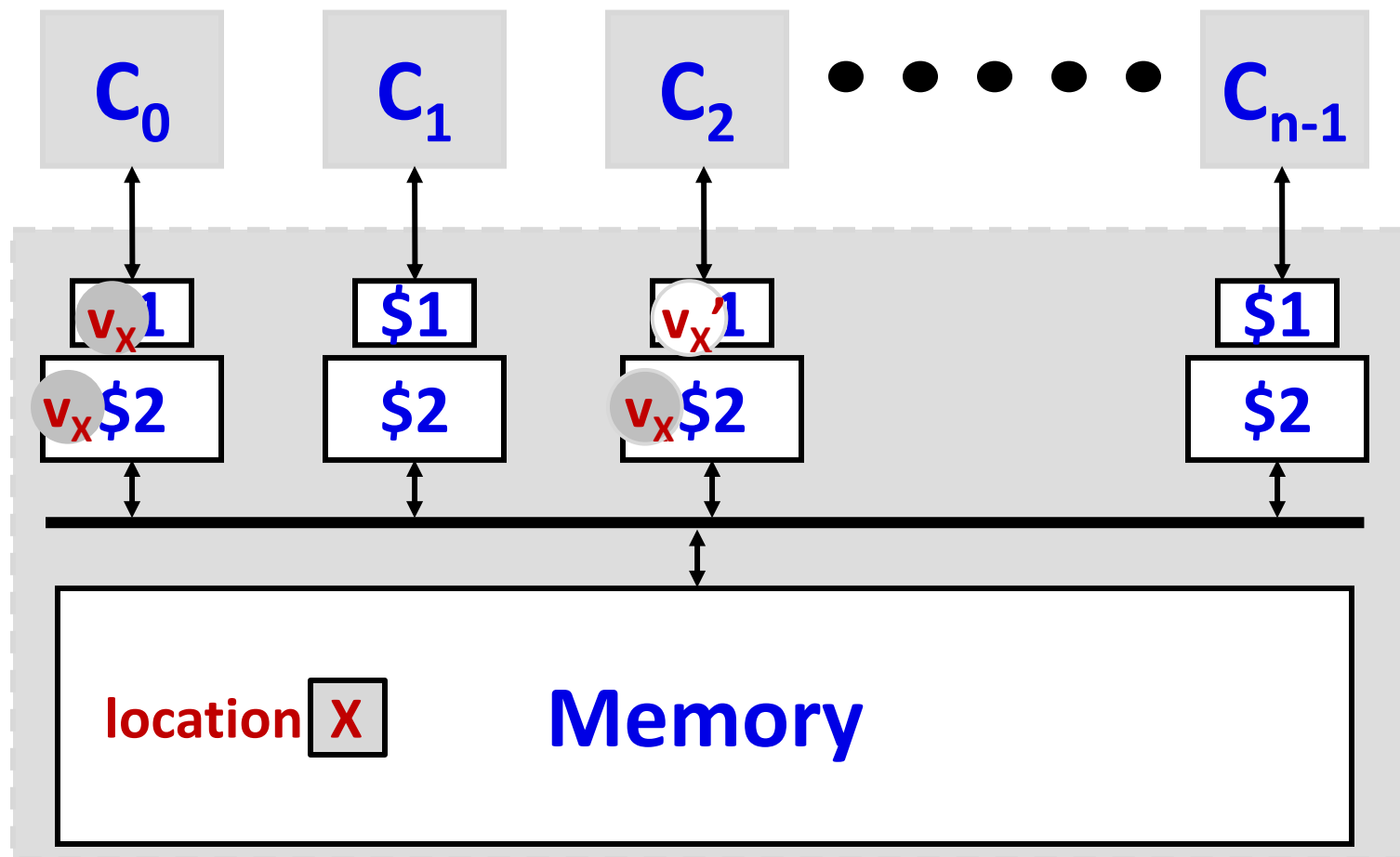
Housekeeping

- Your goal today
 - understand ways to build scalable realizations of shared memory abstraction
- Notices
 - Lab 4: due Friday, 5/7 noon
 - HW5: due Friday, 5/7 noon
 - Midterm 3: Tuesday, 5/11, 5:30~6:25pm
- Readings
 - P&H Ch 5.10
 - *Synthesis Lecture: A Primer on Memory Consistency and Cache Coherence*, 2011 (optional)

Shared Memory Abstraction

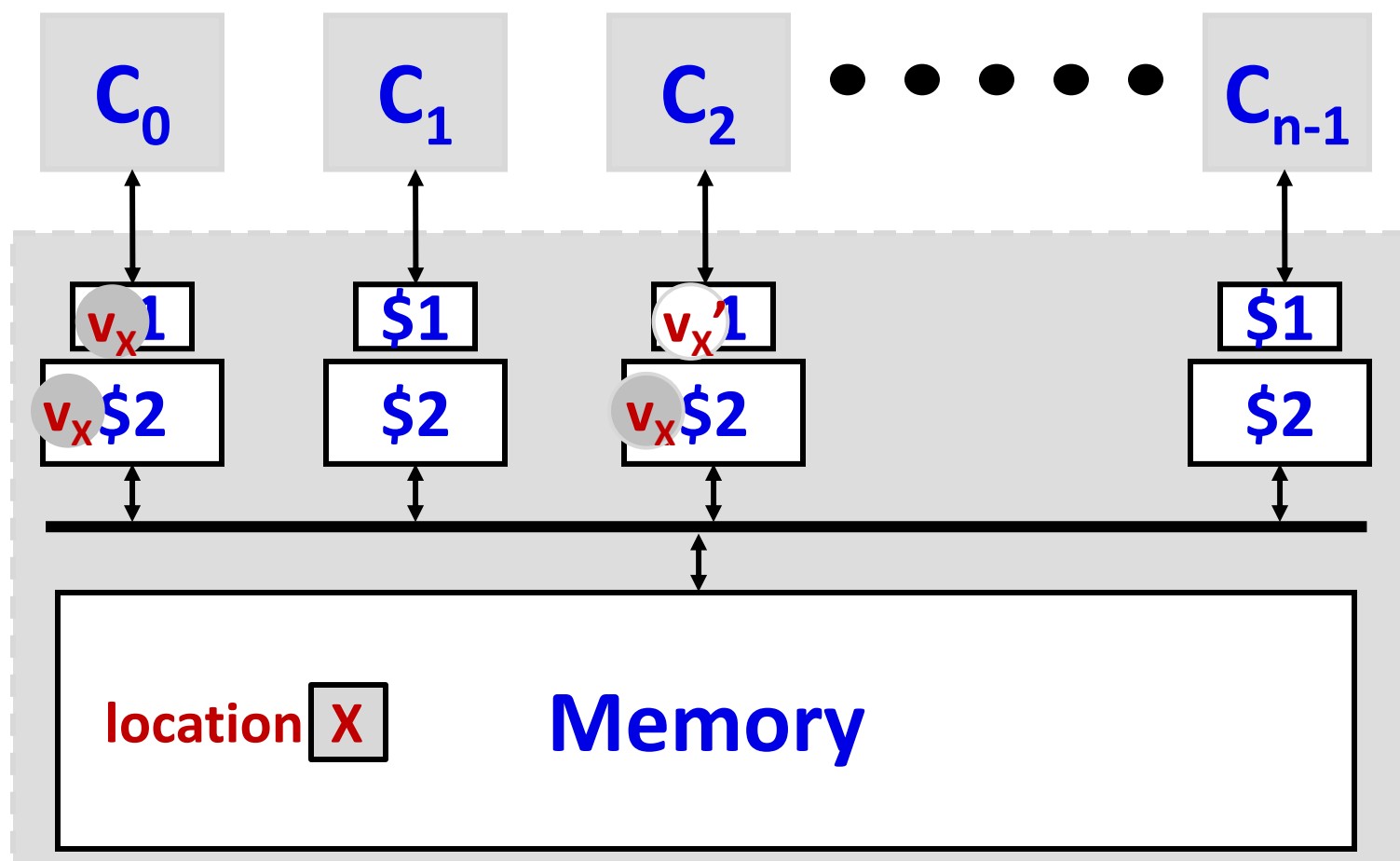


Shared Memory Reality



*Cache coherence (CC) maintains the abstraction processors are working directly on location **X**, despite multiple copies*

Is the below actually wrong . . .



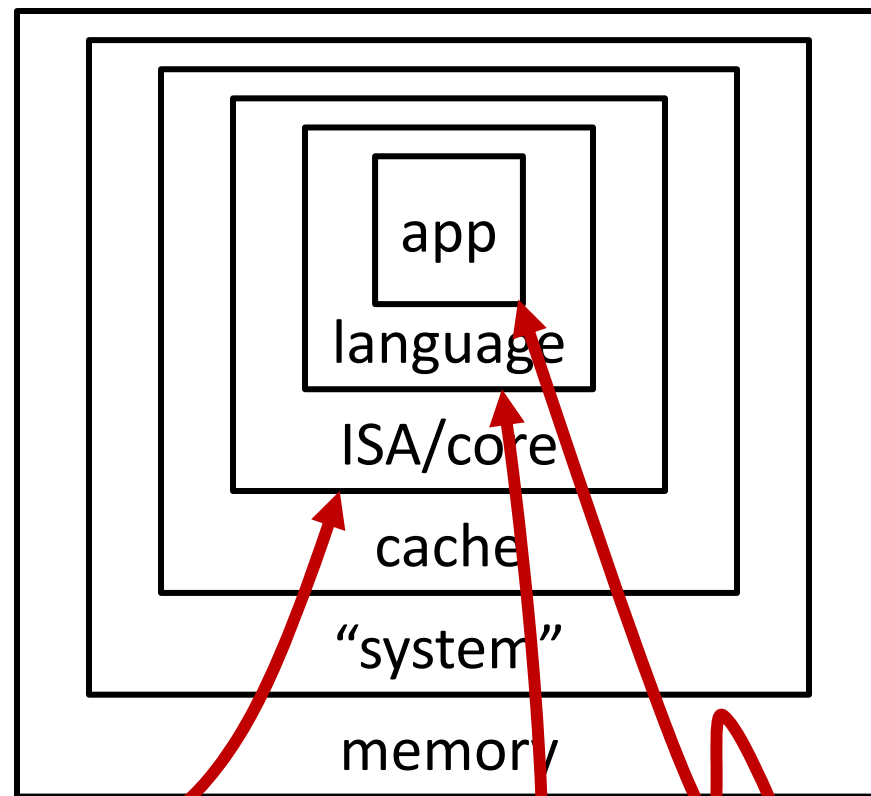
Who decides what is right and wrong?

Mem Consistency vs Cache Coherence

- Consistency is concerned with all loads and stores on same and different addresses
- Consistency presented to inner level need not be same as presented by outer

Stricter to weaker is free

- Per mem location, cache maintains coherence with respect to this consistency model (CC just one part in machinery for consistency)



*Where we were
before (assembly)*

*pthread
gcc -O0
vs -O3*

Extreme Solutions to CC

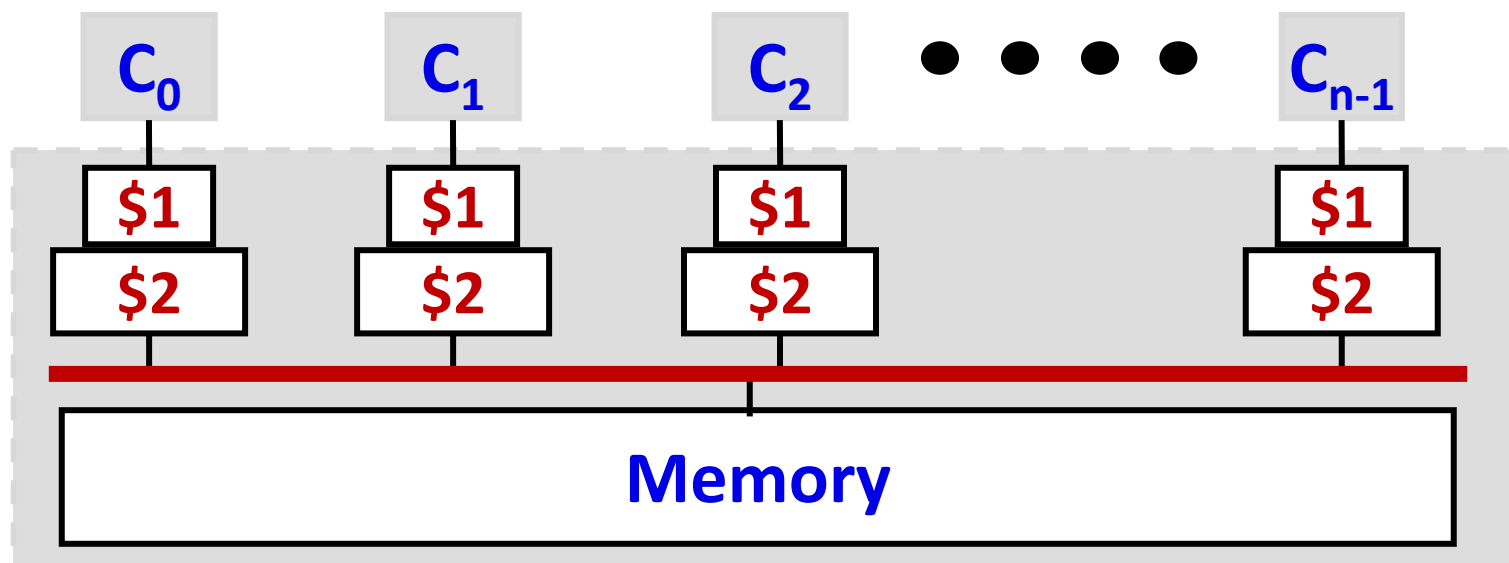
- Problem
 - different cores can hold separate copies of same memory location
 - update to one copy needs to be seen by all
- Extreme solutions to consider first
 0. disallow caching of shared variables
 1. allow only one copy of a memory location at a time
 2. allow multiple copies of a memory location, but they must have the same value at all time

*CC **protocol** is the “rule of conduct”
between caches to enforce a policy*

For simplicity, think SC this point forward

“Snoopy” Protocol for Bus-based Systems

- True bus is a broadcast medium
- Every cache can see (aka snoop) what everyone else does on the bus (reads and writes)
- A cache can even intervene
e.g., one cache could ask another to “retry” a transaction later or respond in place of memory



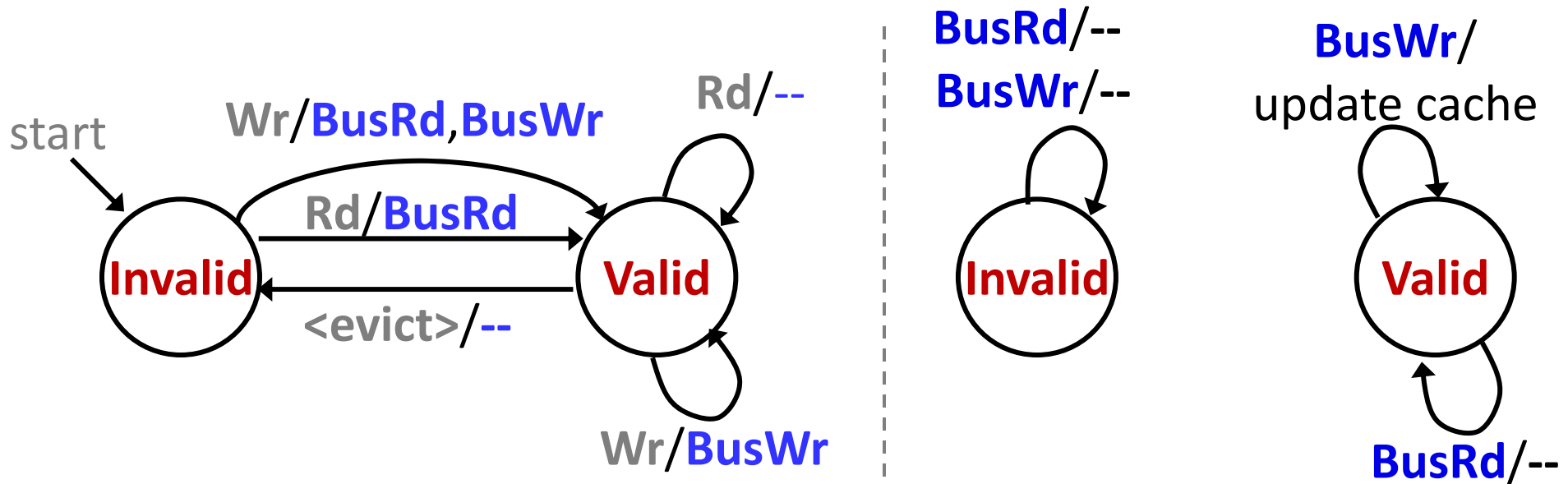
Extreme 1: Multiple Identical Copies

- Multiple write-through caches on a bus
- Processor-side protocol synopsis
 - on read/write miss: issue a memory read txn
 - on read hit: respond directly
 - on write hit: issue a memory write(through) txn
 - on eviction: remove cacheblock silently
- Bus-side protocol synopsis
 - all caches “snoop” for write transactions
 - if write address hits in own cache, update cached copy with new write value

*copy in mem
stays current*

Not scalable due to write-through BW

Protocol Diagram: Multiple Identical Copies



CPU-driven transitions of
cacheblock address **X**
following processor
requests {**Rd**, **Wr**} on **X**

BUS-driven transitions
of cacheblock address **X**
following bus transactions
{**BusRd**, **BusWr**} on **X**

“Invalid” means **X** miss in cache

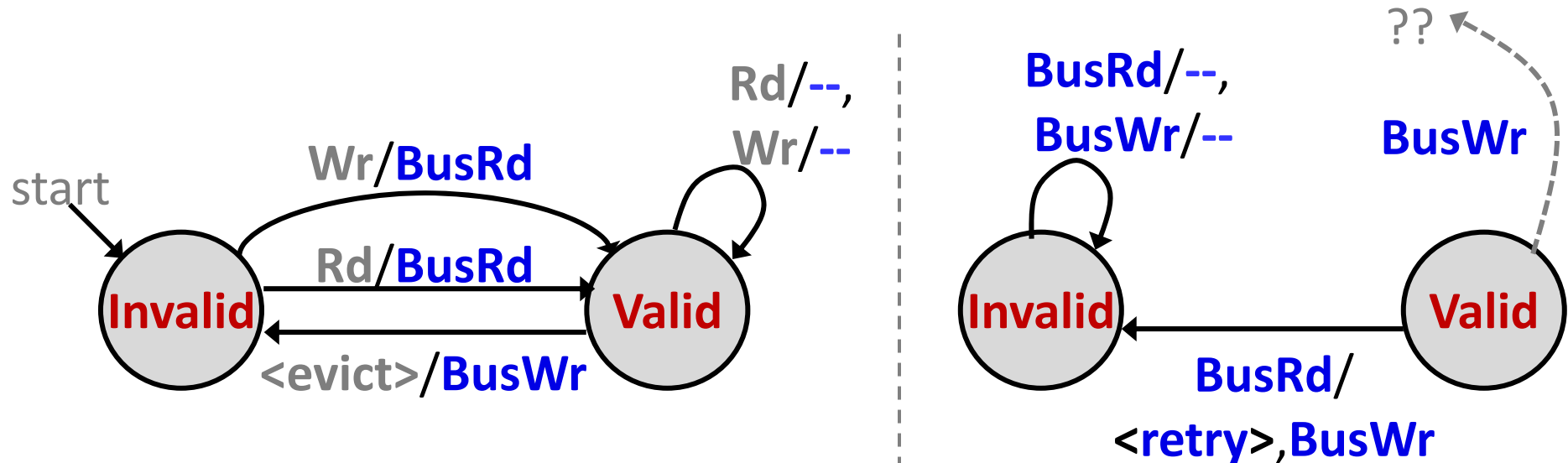
Extreme 2: One Copy at a Time

- Multiple write-back caches on a bus
- Processor-side protocol synopsis
 - on read/write miss: issue a memory read txn
 - on read/write hit: respond directly
 - on eviction: issue a memory write(back) transaction
- Bus-side protocol synopsis
 - all caches “snoop” for read transactions
 - “intervene” if read address hits in cache, **either**
 1. respond with own cached value in place of memory and mark own copy invalid, **OR**
 2. ask requestor to retry later and, in the meantime, evict own cached copy to memory

copy in mem
stale when
cached

No multi-reader support is very limiting

Protocol Diagram: One Copy at a Time



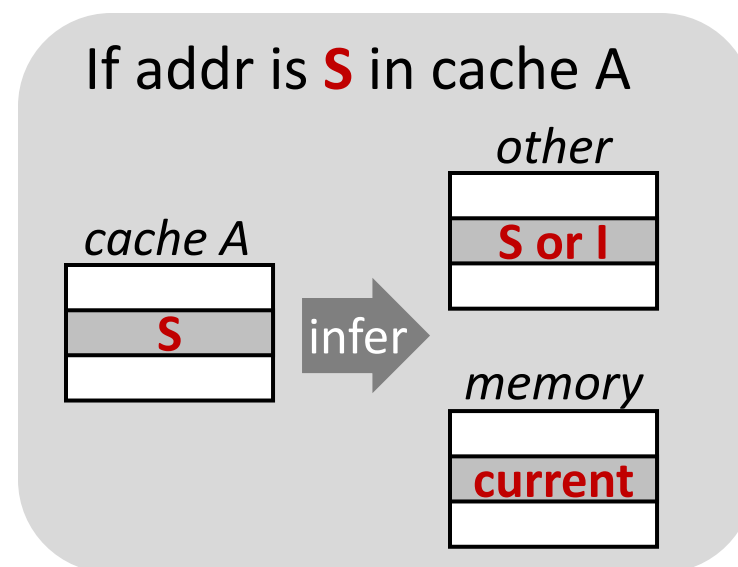
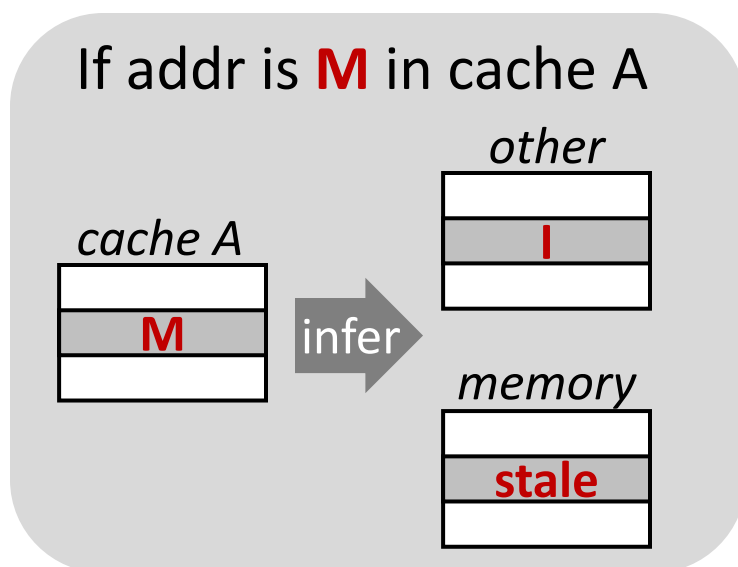
CPU-driven transitions of
cacheblock address **X**
following processor
requests {**Rd**, **Wr**} on **X**

BUS-driven transitions
of cacheblock address **X**
following bus transactions
{**BusRd**, **BusWr**} on **X**

“Invalid” means **X** not in cache

MSI Cache Coherence

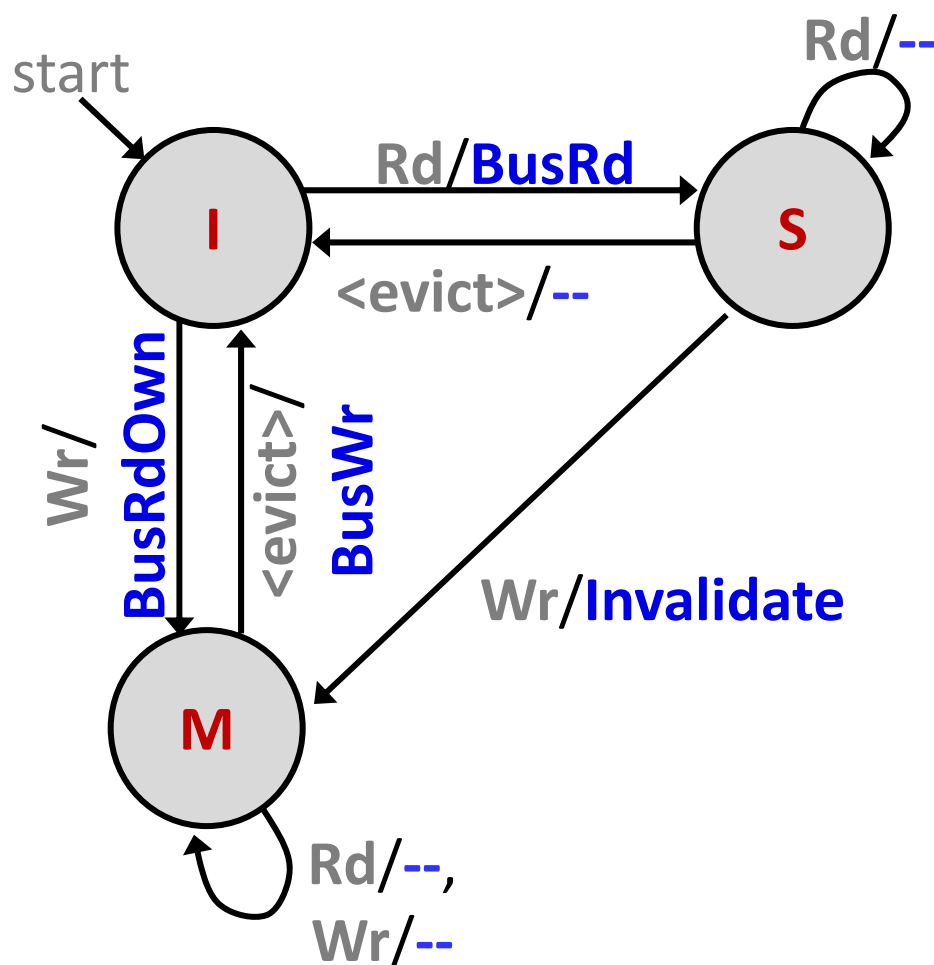
- Efficient middle ground for single-writer, multi-reader
 - multiple read-only copies, OR
 - single writable copy
- Instead of simply **Valid**, introduce **Modified** and **Shared** flavors of valid state for differentiation



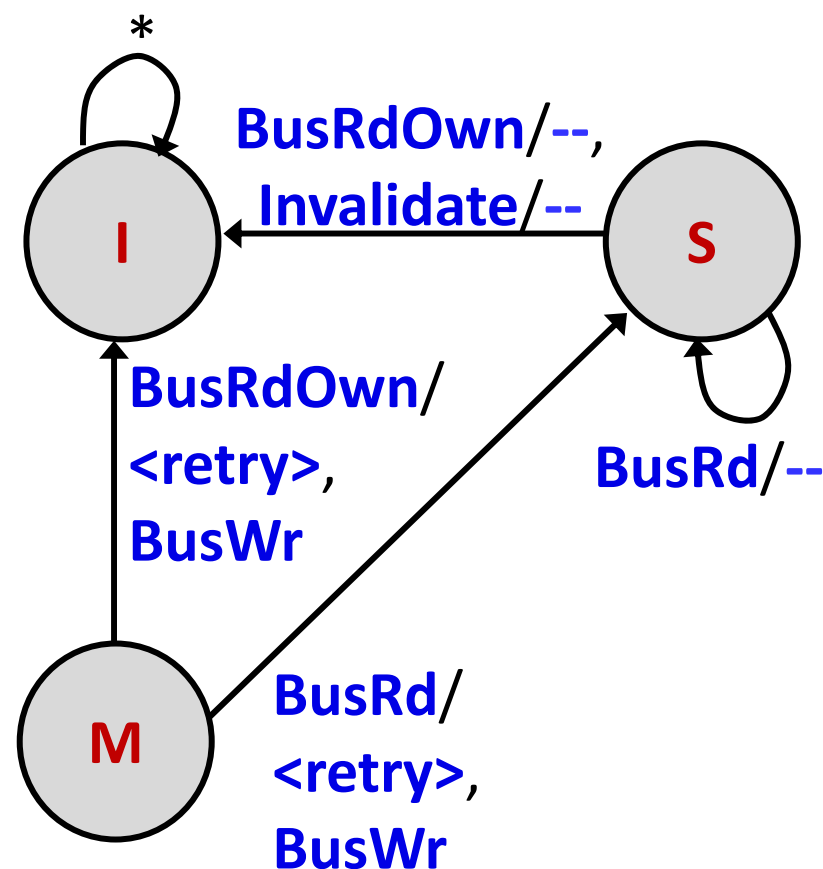
a little bit like dirty not dirty

MSI State Transition Diagram

CPU-driven transitions



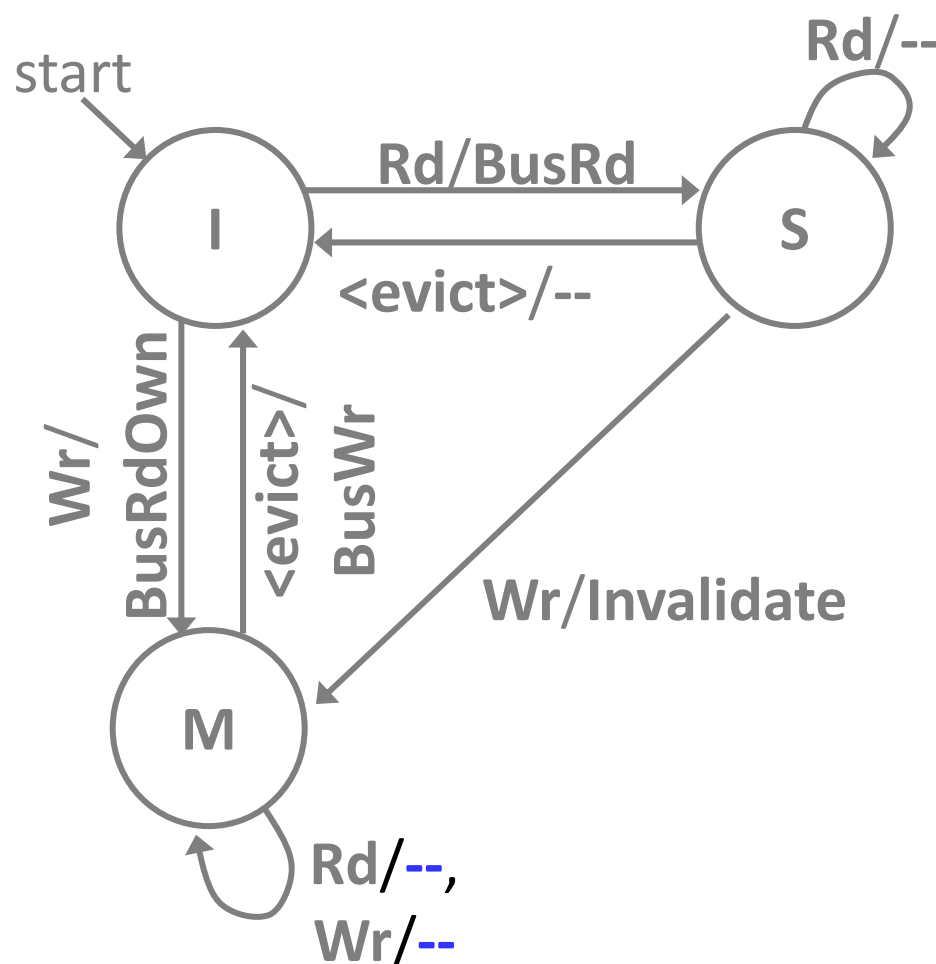
Bus-driven transitions



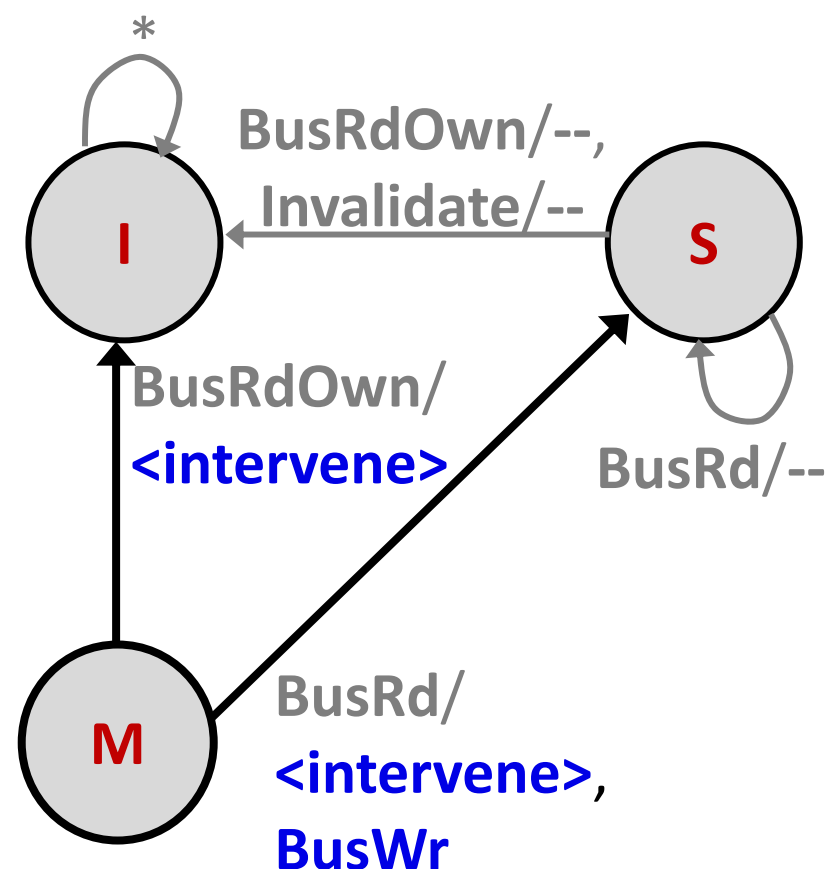
New bus txns **BusRdOwn** and **Invalidate**

Cache-to-Cache Intervention

CPU-driven transitions

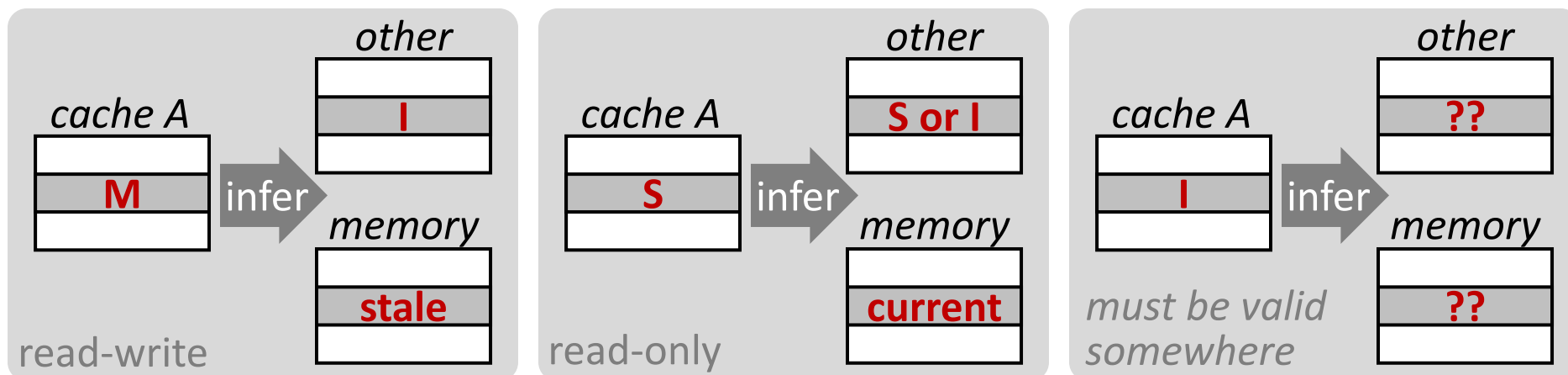


Bus-driven transitions

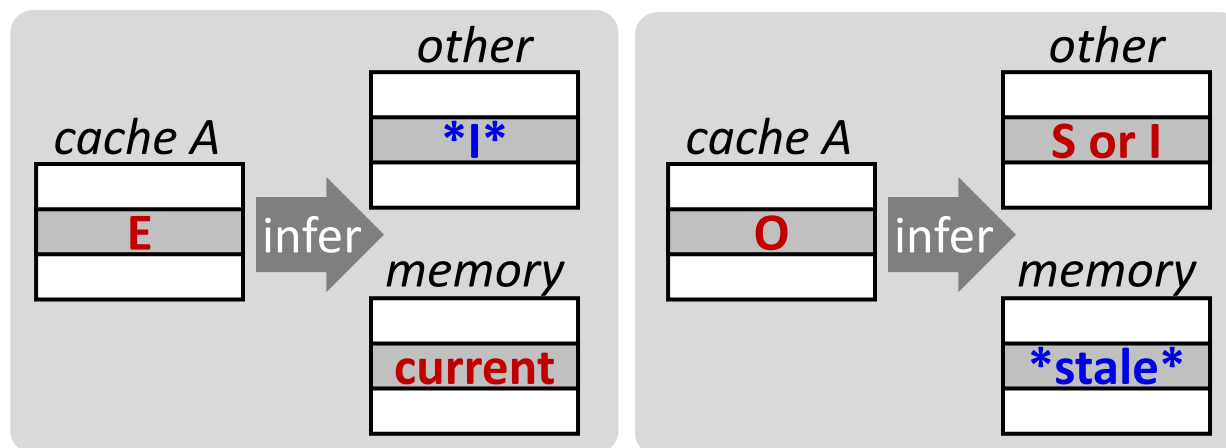


M-copy cache responds in place of DRAM

Nuanced CC States as Optimizations



- **Exclusive**, and **Owned** are read-only like **S**, but . . .



- E**: silent conversion to **M** or **S** or **I**
- O**: faster to serve sharers from cache than DRAM

no intelligence attached to DRAM

CC Managed at Block Granularity

- “Embarrassingly parallel” example in HW5

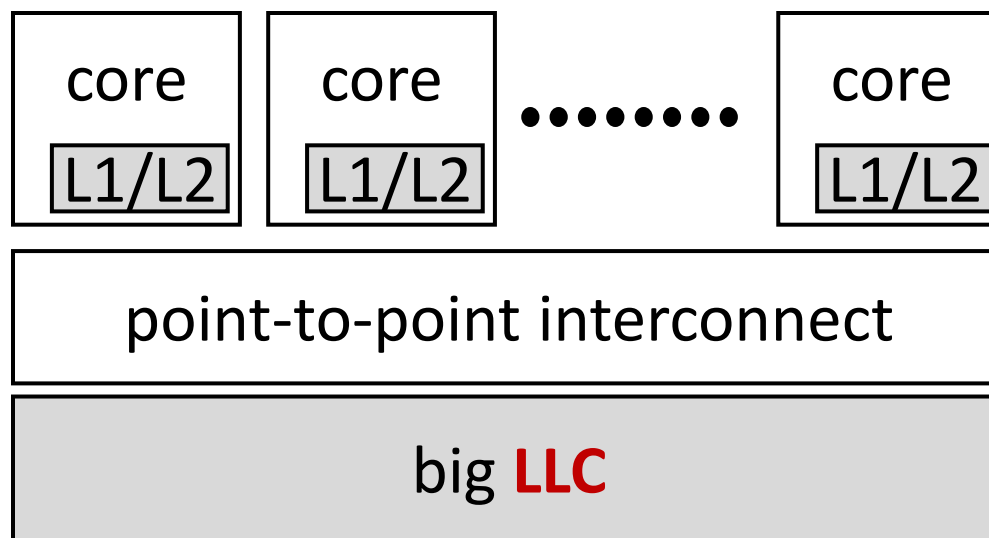
```
void *sumParallel(void *_id) {  
    long id=(long) _id;  
    psum[id]=0;  
    for(long i=0;i<(ARRAY_SIZE/p);i++)  
        psum[id]+=A[id*(ARRAY_SIZE/p) + i];  
}
```

- Threads do not share memory locations in **psum** []
- But, threads do share and contend for cacheblock containing nearby elements of **psum** []
 - cacheblock “ping-pong” between cores hosting threads due to CC
 - for HW5, pad **psum** [] to eliminate “false sharing”

Limitations of Snoopy Bus Protocols

- Broadcast bus is not scalable
 - physics dictates big busses expensive or slow
 - BW is divided by number of processors
- Every bus snoop requires a cache lookup
 - If inclusive hierarchy, snoops only probe lower-level cache (does not compete with processor for L1)*
- Snoopy protocols seem simple but “high-performance” implementations still complicated
 - CPU and bus transactions are not atomic; require intermediate states between **MSI**
 - CC issues intertwined with memory consistency
 - E.g., in **MSI**, can **S**->**M** promote without waiting for invalidate acknowledgement?

Multicores and Manycores

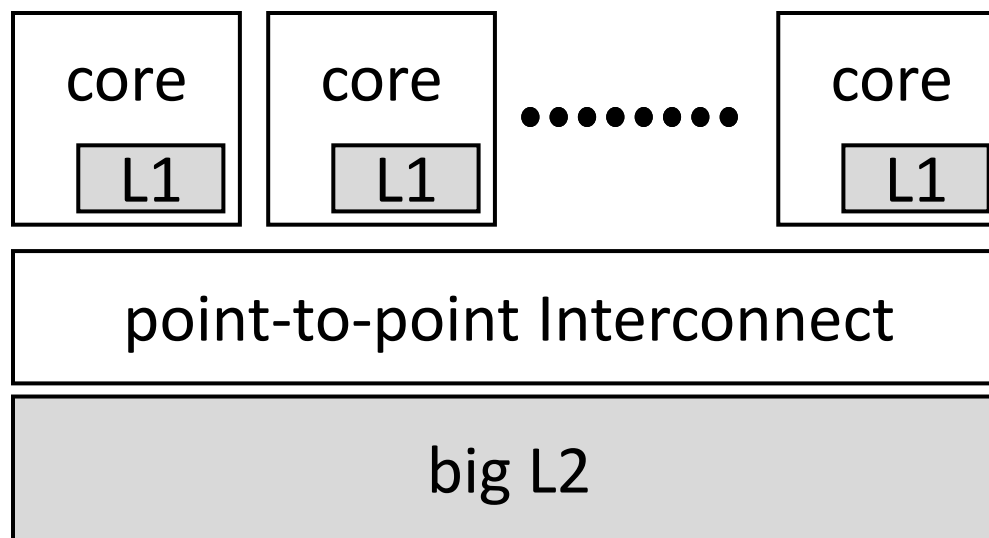


- Private upper-level caches and shared **Last-Level Cache**
- Shared **LLC** typically not inclusive

total capacity of private caches can add up

- Point-to-point interconnect (not a snoopy bus)
connects the private caches to shared **LLC**

Bookkeeping Instead of Snooping



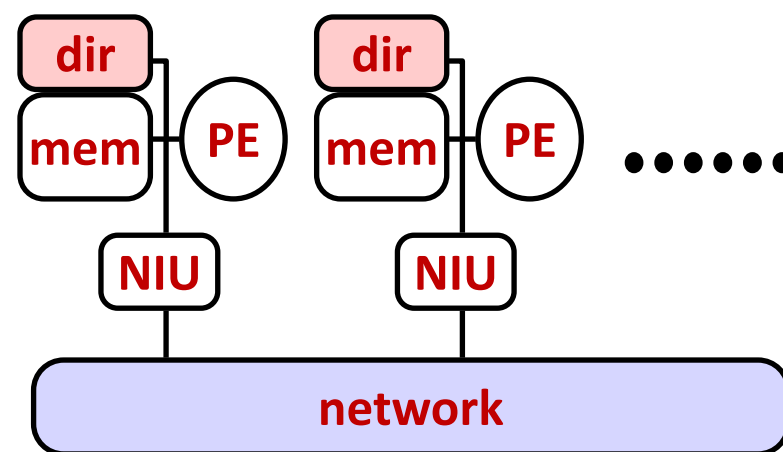
E.g., Piranha [ISCA 2000]

- L2 controller maintains duplicate L1 tags and CC states
- on L1 miss, L2 controller lookup in directory to determine affected L1s and required transitions
- external CC probes consult L2 bookkeeping also

MIMD Shared Memory: Big Irons

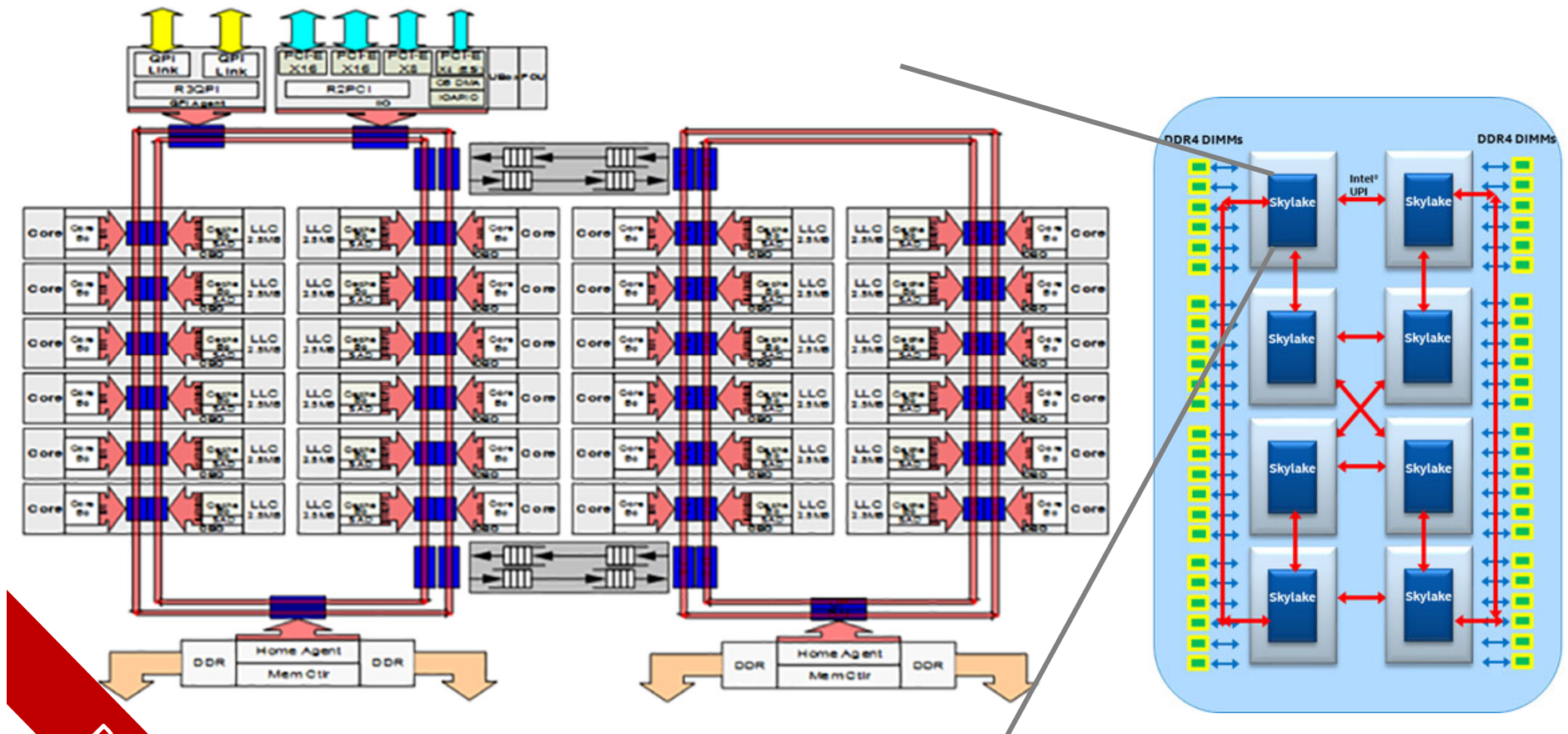
Distributed Shared Memory

- UMA hard to scale due to concentration of BW
- Large scale SMPs have distributed memory with non-uniform memory accesses (NUMA)
 - “local” memory pages (faster to access)
 - “remote” memory pages (slower to access)
 - cache-coherence still possible but complicated
- E.g., SGI Origin 2000
 - upto 512 CPUs and 512GB DRAM (\$40M)
 - 48 128-CPU system was collectively the 2nd fastest computer (3TFLOPS) in 1999



Recall

Modern DSM in the small

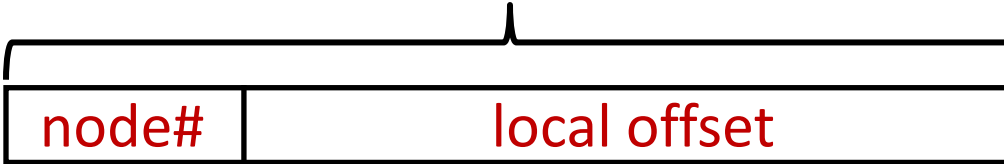



Recall

[<https://software.intel.com/en-us/articles/intel-xeon-processor-scalable-family-technical-overview>]

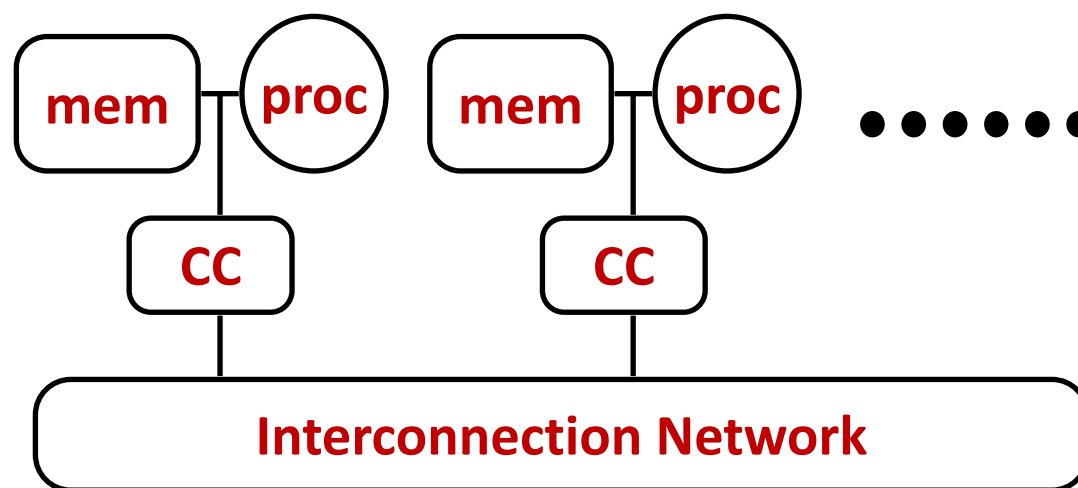
Global Address Layout

- Every memory location has a “home” node
- With respect to a particular processor, every location is either “local” or “remote”

- Interleaving 1: 
- Interleaving 2: 

When accessing nearby memory locations, option
(1) fast for local node; (2) better bandwidth
(usually a configurable option)

Cache-Coherent DSM



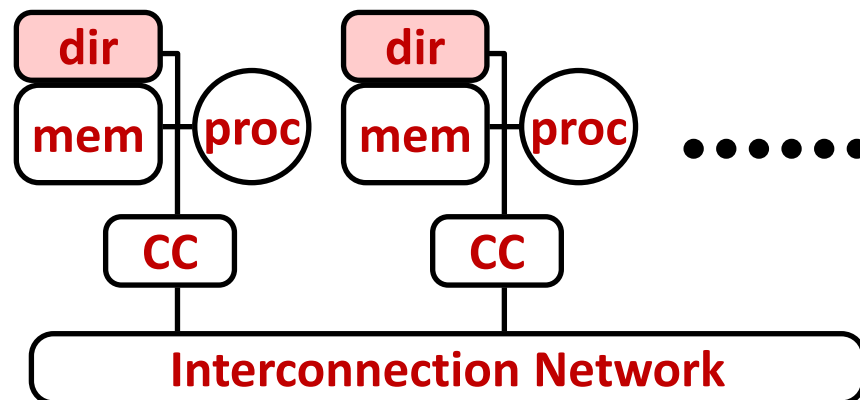
- How to coordinate CC state transitions for large number of far-apart nodes?

Option 1: mimic snooping by exchanging messages with all nodes—*explosion in CC traffic*

Option 2: centrally maintain duplicates of all caches' tags and CC states—*concentration of CC traffic*

Directory-Based Cache Coherence

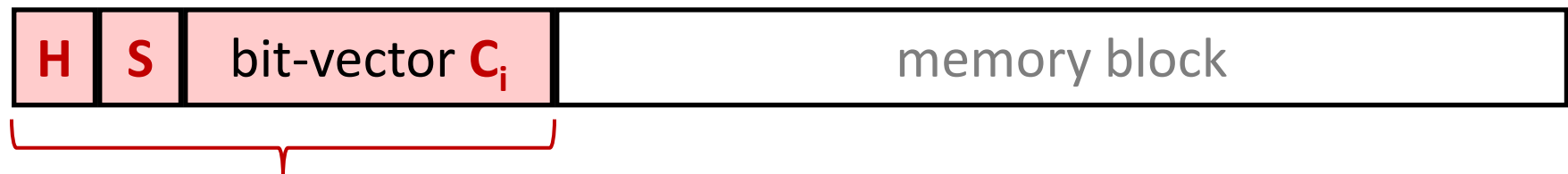
- Distributed bookkeeping
 - keep track for each block in home memory which caches have copies and in what state



- Avoid unnecessary communication
 - on a cache miss, local CC-controller sends request to home node of address
 - based on directory information, home-node CC-controller communicates with only affected nodes

A Simple Directory Example

- Extend every cacheblock-sized memory block with a directory entry



- **H=1** indicates “at home”; **S=1** indicates shared
- If **H=0**, C_i bitmaps if node_{*i*} has a cached copy
 - uncached (**H=1**, **S=***): no cached copy exists
 - shared (**H=0**, **S=1**): for all $C_i==1$, node_{*i*} has copy
 - modified (**H=0**, **S=0**): if $C_i==1$, node_{*i*} has only copy

C_i storage significant for large systems and
upperbounds system size at design time

Directory-Based Cache Coherence

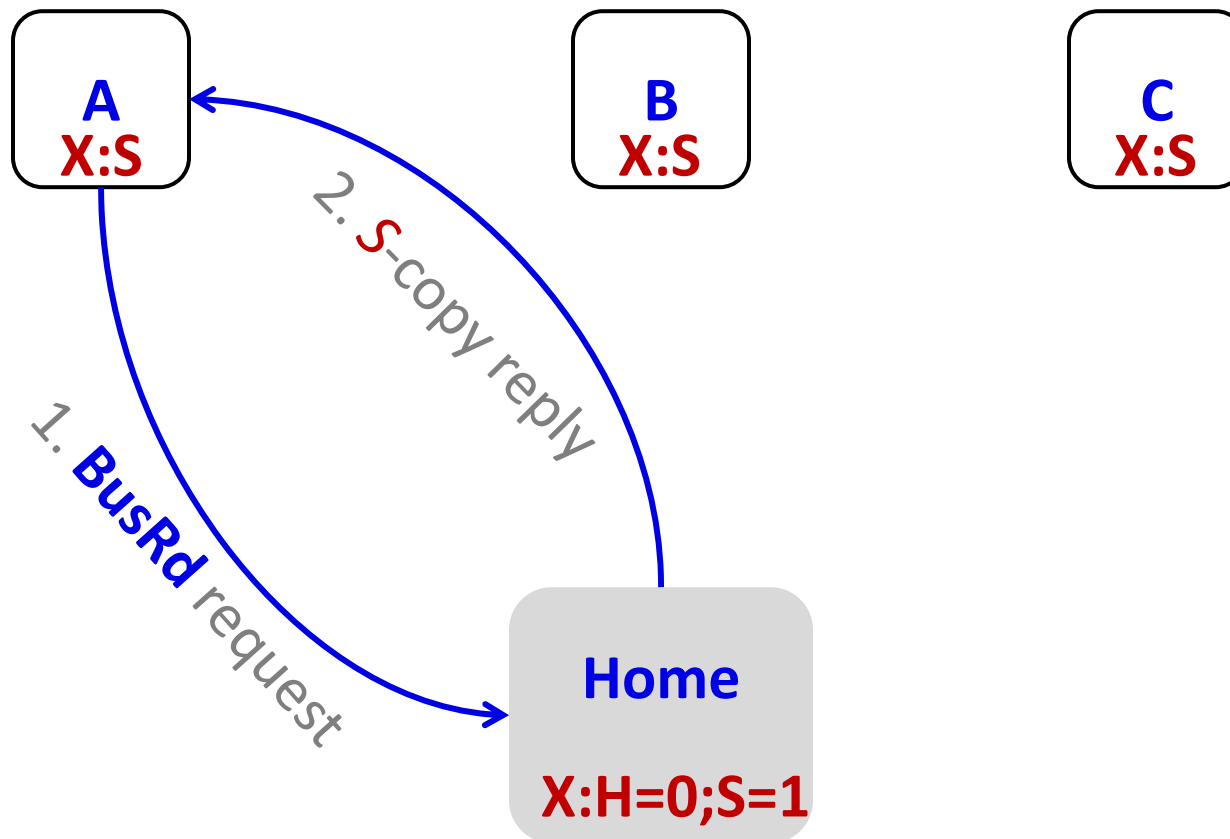
- Based on similar **MSI** states and transitions as snoopy but tracked through point-to-point messages
- E.g., **BusRd** request reaches home from **A** when
 - uncached (**H=1**, **S=***) \Rightarrow **H=0**; **S=1**; **C_A=1**; return **S**-copy
 - shared (**H=0**, **S=1**) \Rightarrow **C_A=1**; return **S**-copy
 - modified (**H=0**, **S=0**) \Rightarrow **1.** ask current owner to
downgrade (**M** \rightarrow **S**) and send
data value back to home
2. **S=1**; **C_A=1**; return **S**-copy

Directory-Based Cache Coherence (continued)

- **BusRdOwn** request reaches home from **A** when
 - uncached (**H=1, S=***) \Rightarrow **H=0, S=0, C_A=1**; return **M**-copy
 - shared (**H=0, S=1**) \Rightarrow
 1. ask all current copy holders to invalidate (*and ack?*)
 2. **S=0; C_A=1; C_{i!=A}=0**;
return **M**-copy
 - modified (**H=0, S=0**):
 1. ask current owner to invalidate and send data value to home
 2. **C_A=1; C_{i!=A}=0**; return **M**-copy

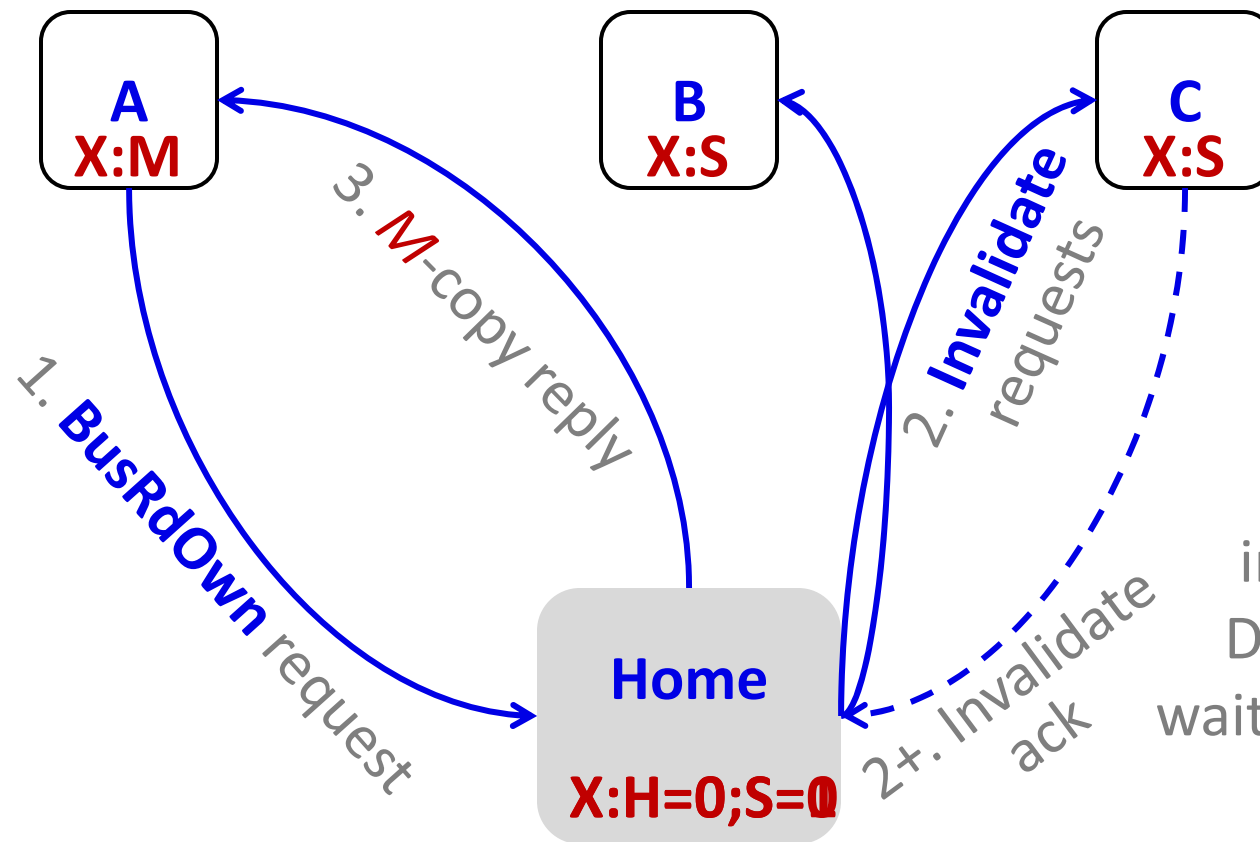
Multi-Hop MSI Protocol Example: Shared Read

- Initially **S**-copy at node-**B/C**; read cache miss at node-**A**



Multi-Hop MSI Protocol Example: Invalidation

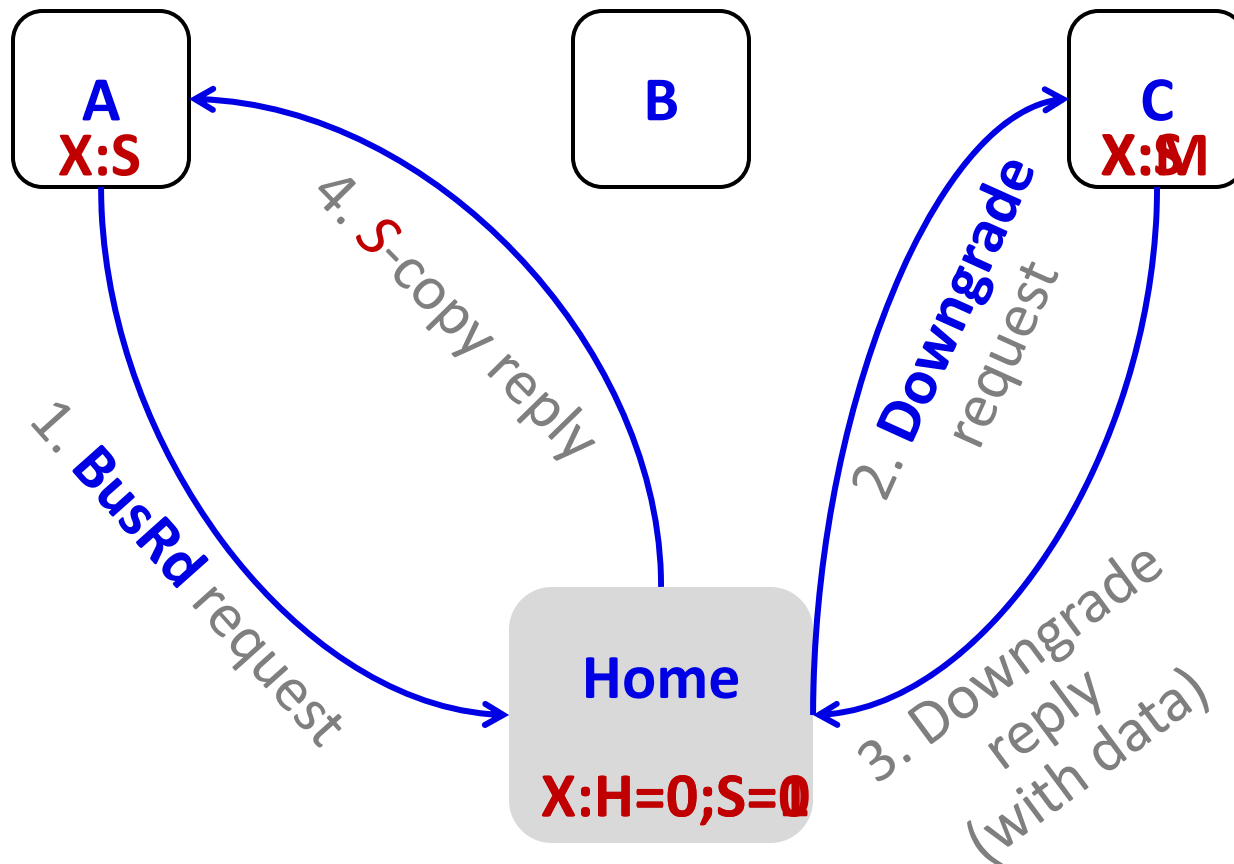
- Initially **S**-copy at node-**B/C**; write cache miss at node-**A**



Do you need
invalidate ack?
Do you need to
wait for invalidate
ack before
returning **M**-copy

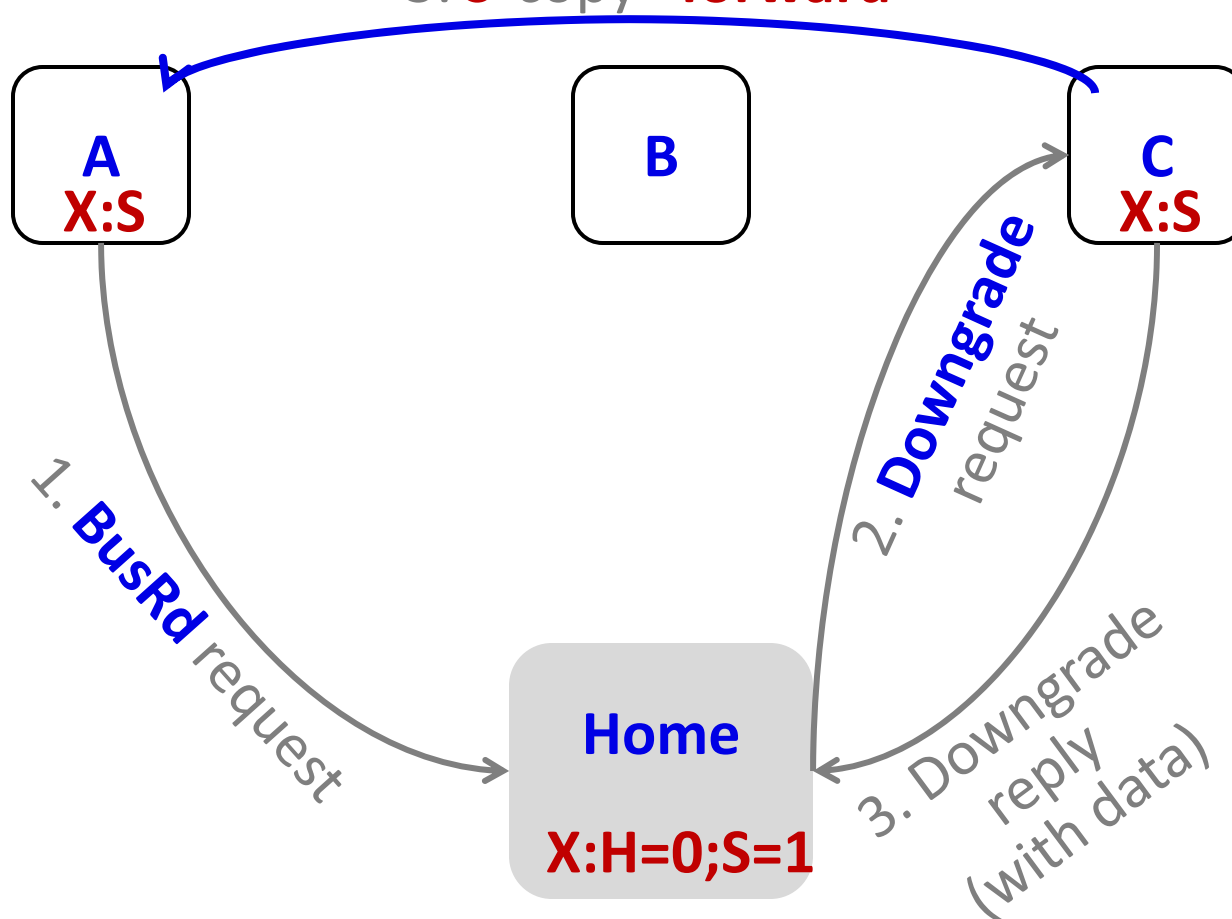
Multi-Hop MSI Protocol Example: Downgrade

- Initially *M*-copy at node-*C*; read cache miss at node-*A*



Multi-Hop MSI Protocol Example: Forwarding

- Initially *M*-copy at node-*C*; read cache miss at node-*A*
3. *S*-copy “forward”



It is much, much harder than it looks

- CC state information not always current
 - home doesn't know when a cache invalidates a block spontaneously (e.g. on replacement)
 - home could send requests when no-longer apply
- CC transitions not atomic
 - another bus request can arrive while an earlier one is still being serviced
 - if not careful, dependencies can lead to deadlocks
- CC transactions are distributed and concurrent
 - no single point of serialization for different addr
 - subtle interplay with memory consistency

Everything today is simplified “intro”