# 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 this week
  - HW6, due Monday 5/2 noon
  - **Final Exam on 5/6**
- Readings
  - P&H Ch 5.10
  - *Synthesis Lecture: A Primer on Memory Consistency and Cache Coherence*, 2011 (optional)
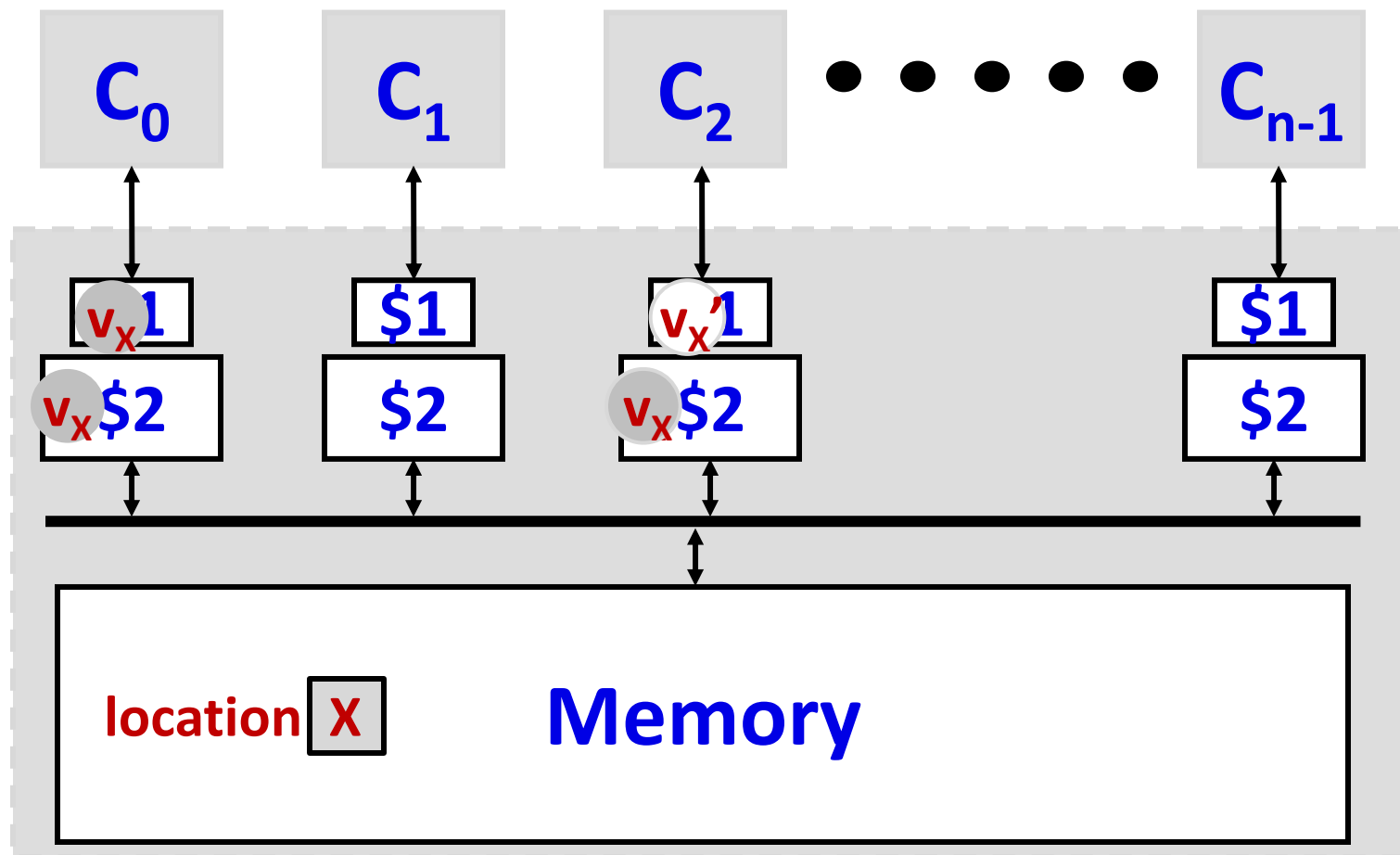
Shared Memory Abstraction

C_0 C_1 C_2 $\bullet\ \bullet\ \bullet\ \bullet\ \bullet$ C_{n-1}
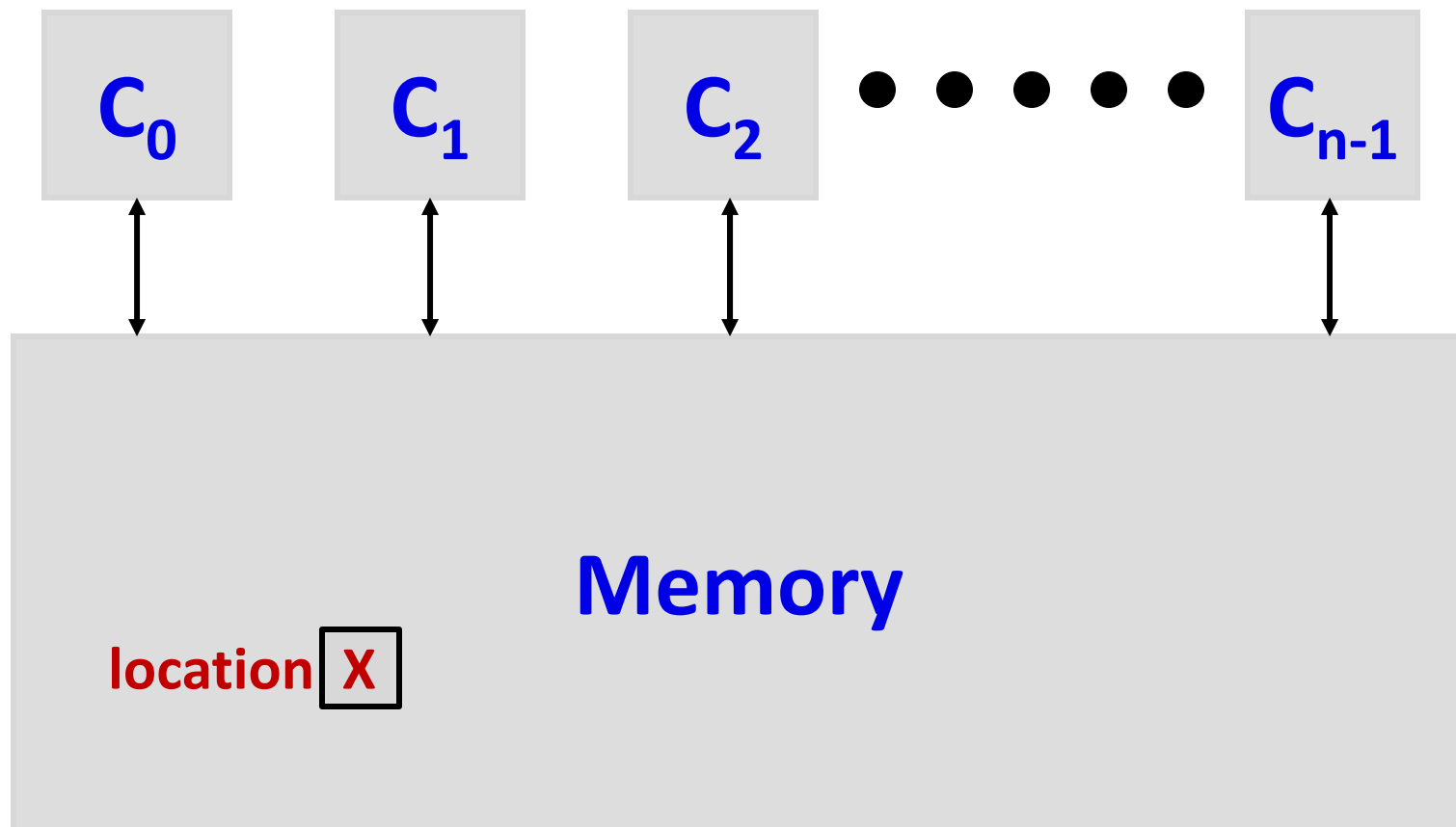
Memory

location X

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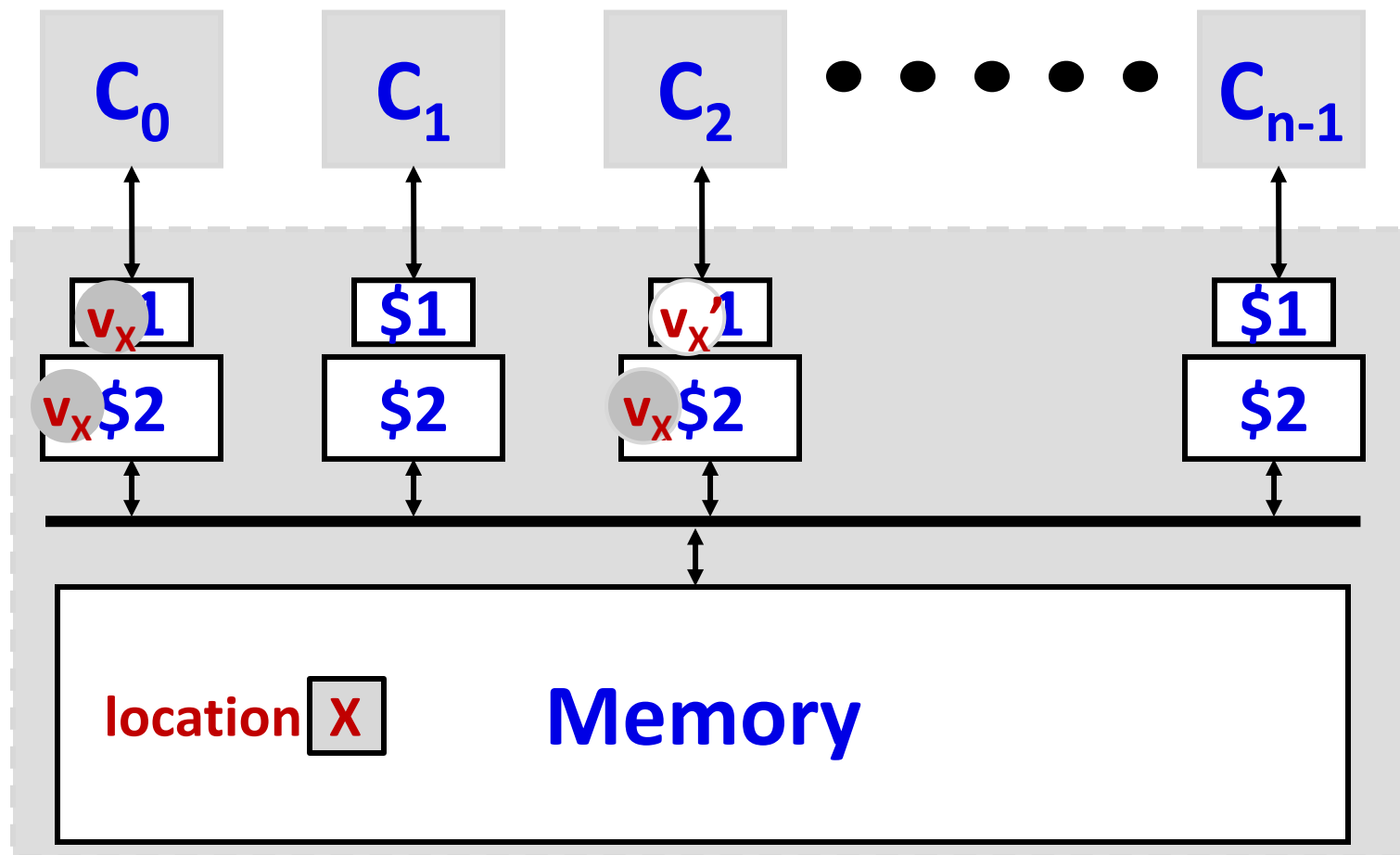
Cache coherence (CC) *maintains the abstraction processors are working directly on location* **X***, despite multiple copies*

# Is the below actually wrong . . .

Who decides what is right and wrong?

Mem Consistency vs Cache Coherence

- Consistency is concerned with <u>all loads and stores on same and different addresses</u>

- Consistency presented to inner level need not be same as presented by outer

  Stricter to weaker is free

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

app

language

ISA/core

cache

"system"
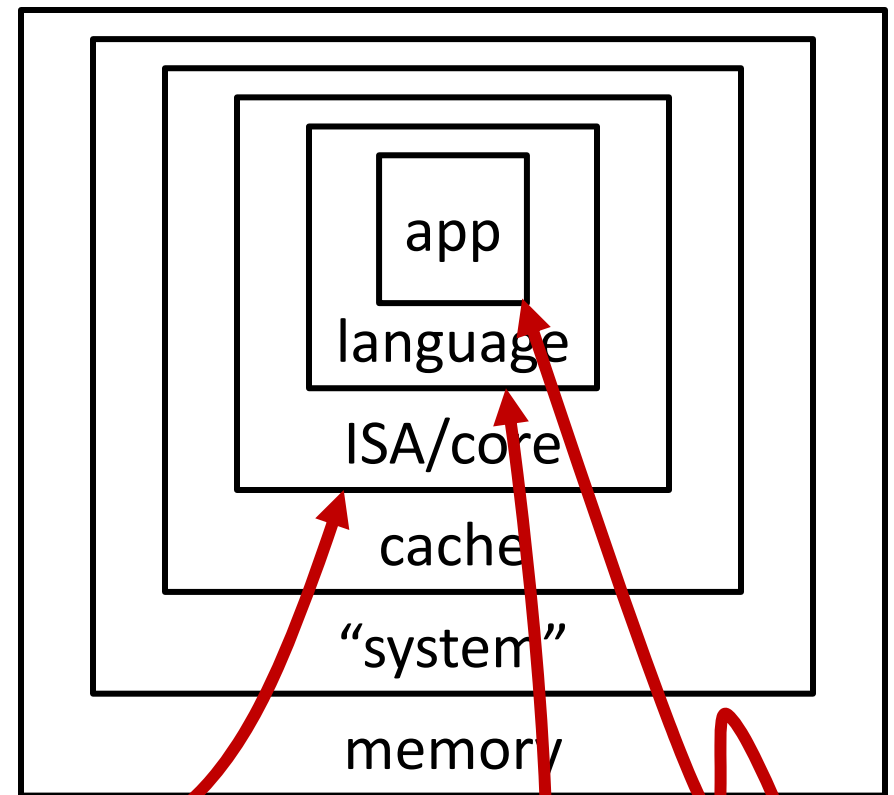
memory

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 <u>at a time(?)</u>
  2. allow multiple copies of a memory location, but they must have the same value <u>at all time(?)</u>

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

"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
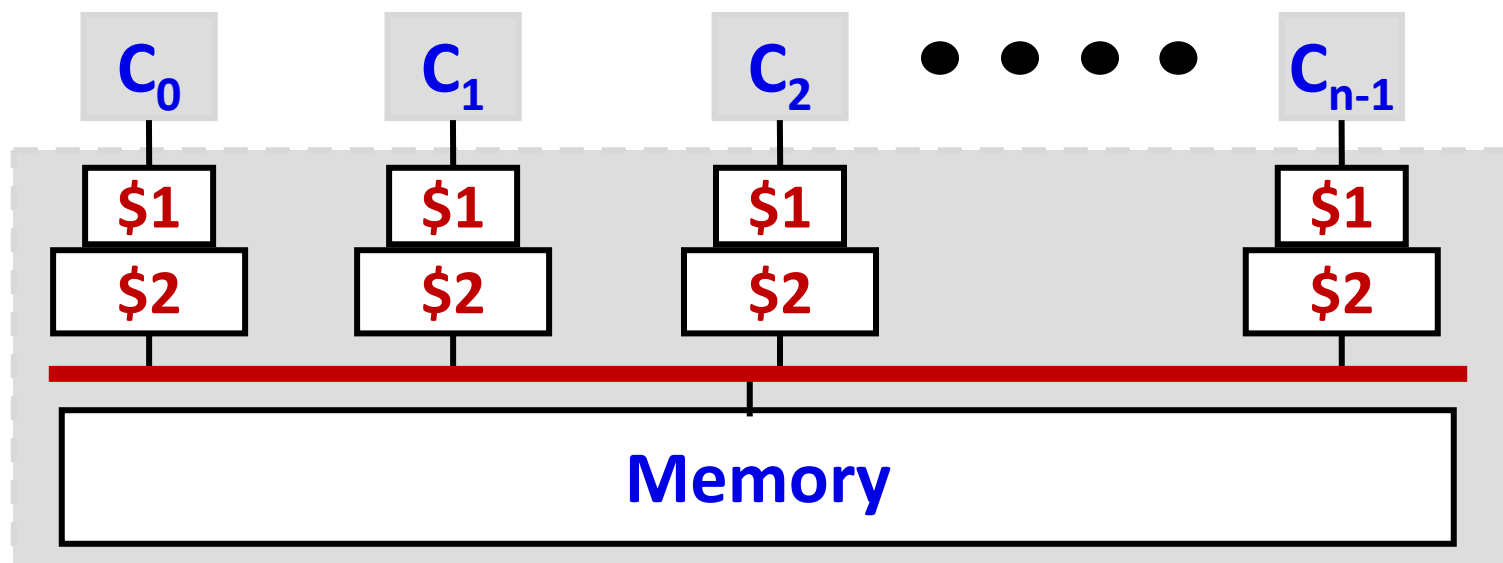
Extreme 1: Multiple Identical Copies

- Multiple write-through caches on a bus write-through BW not scalable
- 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

All cache & mem copies kept "current", but writer sees effect before rest—not SC even if processors in-order
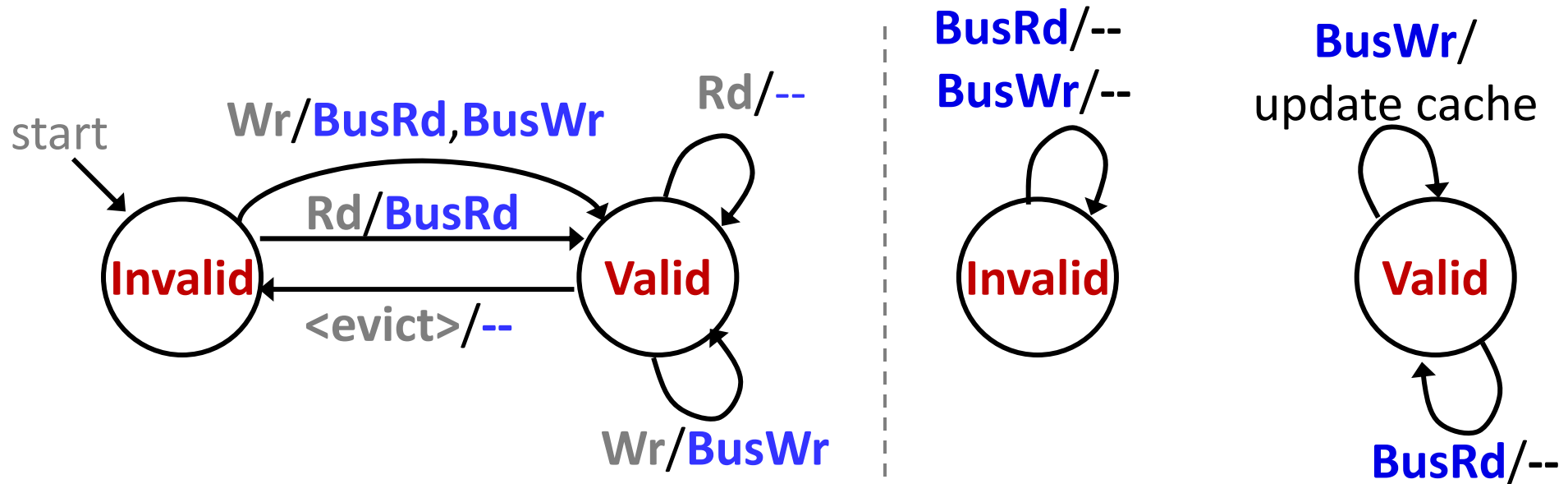
Aside: Strictness of Memory Consistency

- **Clock Synchronized RTL:** most strict; no ambiguity

- **Sequential Consistency (SC):** strictest w/o clock; all threads agree on order of all ld/st of all threads

- **Weak Consistency (WC):** weakest reasonable; each thread enforce only own RAW/WAR/WAW order

- **Processor Consistency (PC):** assume in-order cores, snoopy write-through cache . . . . .$SC >_{strict} PC >_{strict} WC$

T1: store(**X**, 1);	**T2:** store(**Y**, 1);
vy = load(**Y**);	**vx** = load(**X**);

- Initially **X** = 0, **Y** = 0, can **vx**=**vy**=0?

  *hint: what if **X** and **Y** cached at start?*

Protocol Diagram:
Multiple Identical Copies

start

Invalid → **Valid**

Wr/**BusRd,BusWr**

Rd/**BusRd**

<evict>/--

Rd/--

Wr/**BusWr**

BusRd/--
BusWr/--

Invalid

BusWr/
update cache

Valid

BusRd/--

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

copy in mem stale when cached

- 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

If <u>truly</u> only 1 copy, effect of a write is "atomic" to all
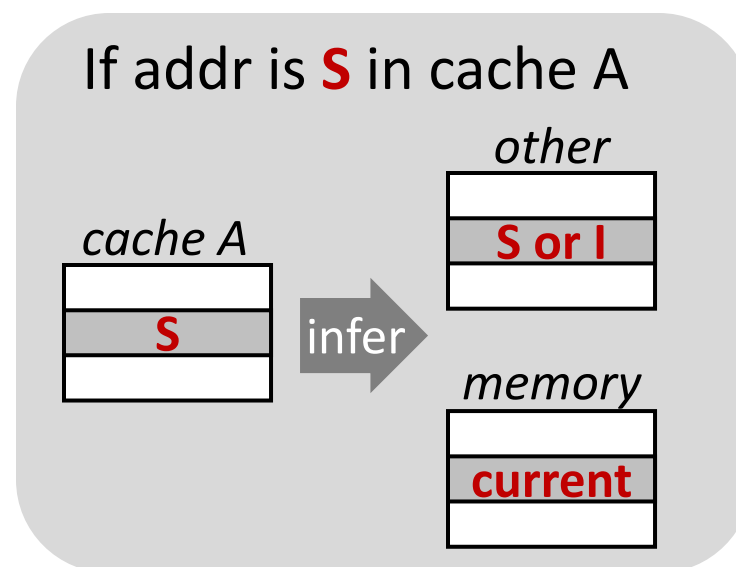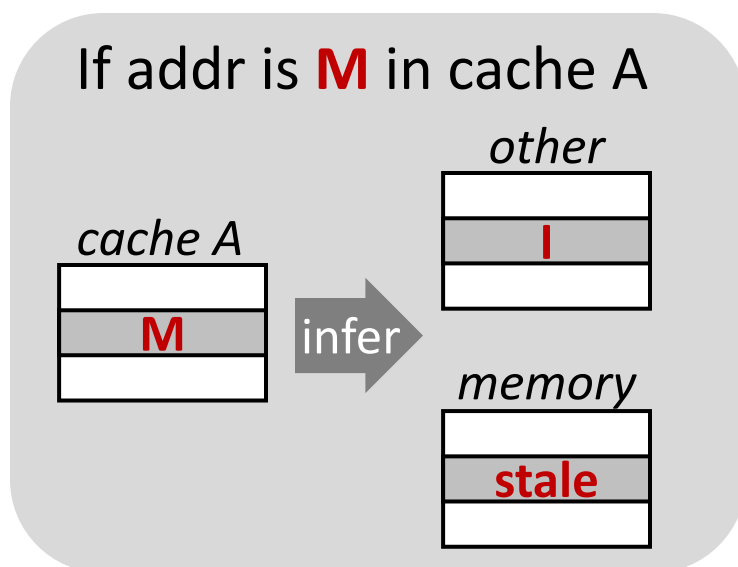
Protocol Diagram: One Copy at a Time
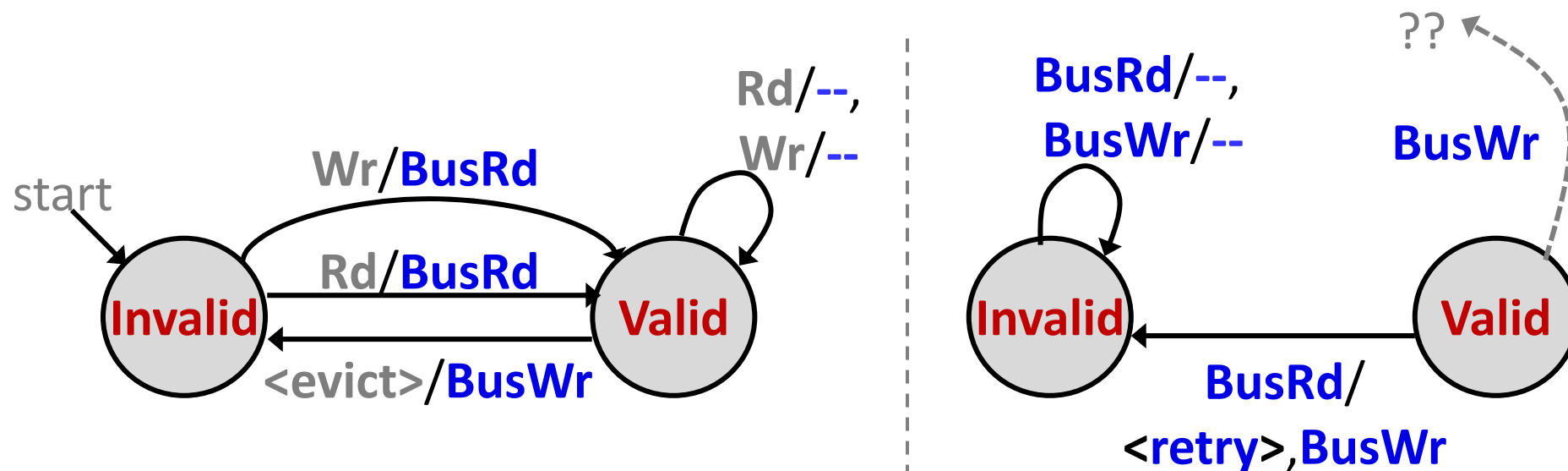
CPU-driven transitions of cacheblock address **X** following processor requests {**Rd**, **Wr**} on **X**

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

"Invalid" means **X** not in cache
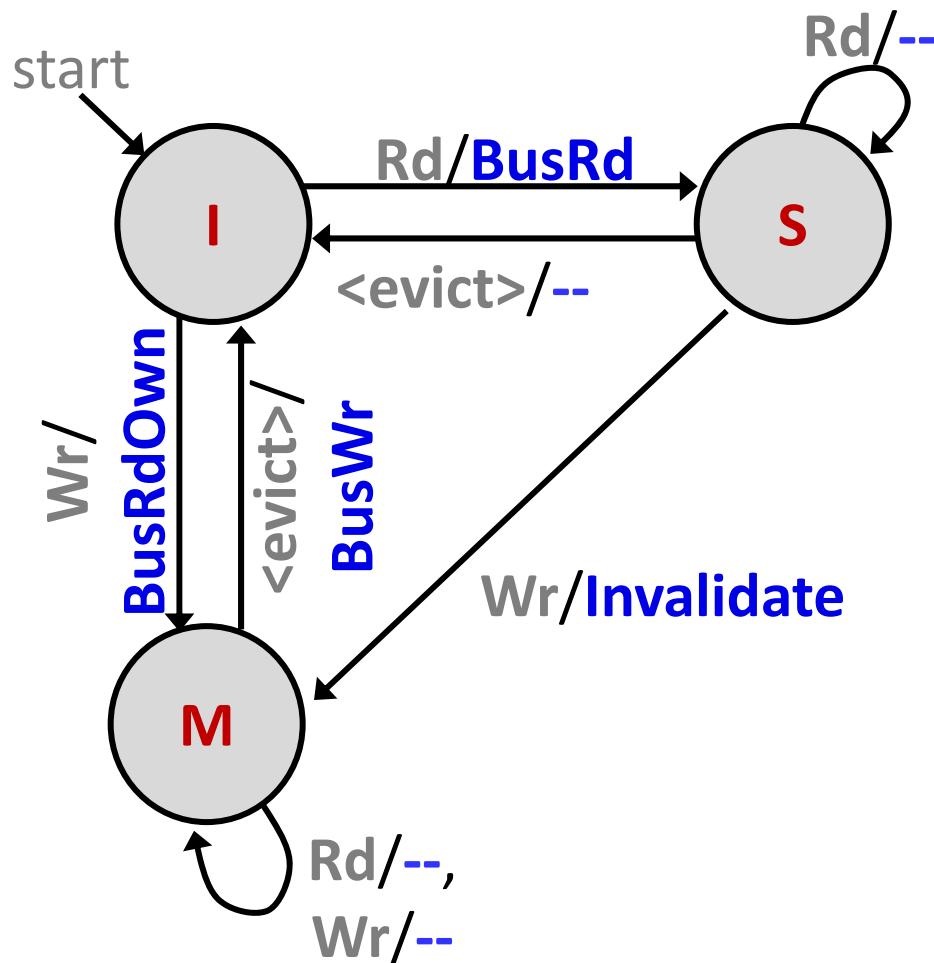
MSI Cache Coherence

- An efficient middle ground for <u>single-writer, multi-reader</u>

  - multiple read-only copies, OR

  - single writable copy

- Instead of simply **Valid**, introduce **Modified** and **Shared** flavors of valid state for differentiation

If addr is **M** in cache A

cache A

M

infer →

other

I

memory

stale

If addr is **S** in cache A

cache A

S

infer →

other

S or I

memory

current

a little bit like dirty not dirty

MSI State Transition Diagram

CPU-driven transitions

start → I

I — Rd/**BusRd** → S

S — <evict>/**--** → I

S: Rd/**--** (self loop)

I ↔ M: Wr/**BusRdOwn** (down), <evict>/**BusWr** (up)

S — Wr/**Invalidate** → M

M: Rd/**--**, Wr/**--** (self loop)

Bus-driven transitions

I: * (self loop)

S — **BusRdOwn**/**--**, **Invalidate**/**--** → I

M — **BusRdOwn**/**<retry>**, **BusWr** → I

M — **BusRd**/**<retry>**, **BusWr** → S

S: **BusRd**/**--** (self loop)
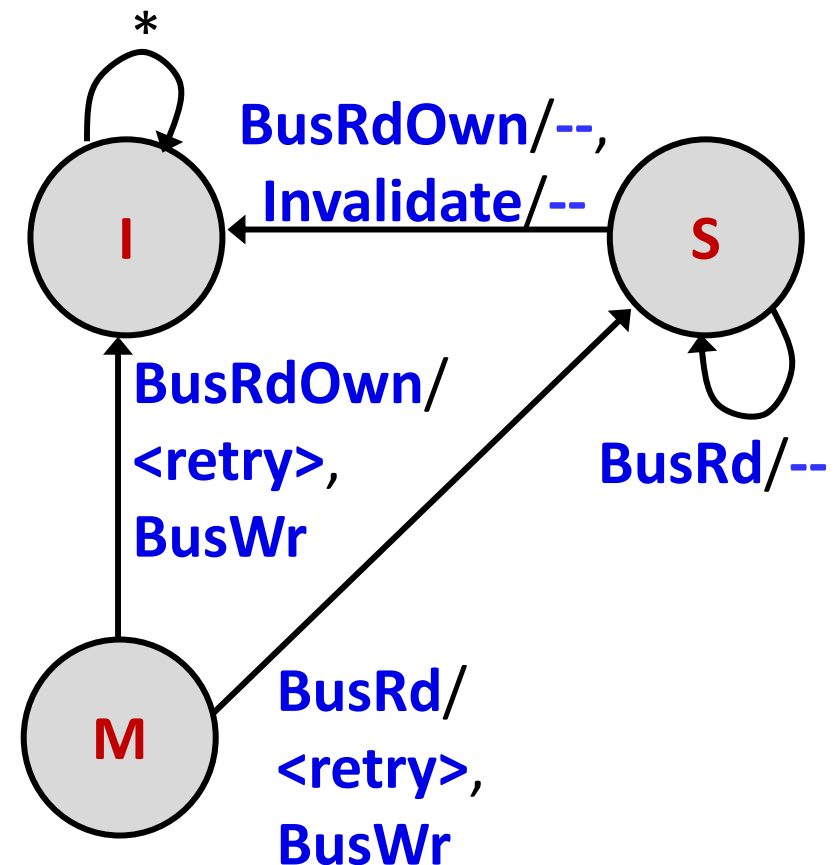
New bus txns **BusRdOwn** and **Invalidate**

Cache-to-Cache Intervention

CPU-driven transitions

Rd/--

start

Rd/BusRd

I

S

<evict>/--

Wr/BusRdOwn

<evict>/BusWr

Wr/Invalidate

M

Rd/--,
Wr/--

Bus-driven transitions

*

BusRdOwn/--,
Invalidate/--

I

S

BusRdOwn/
<intervene>

BusRd/--

M

BusRd/
<intervene>,
BusWr
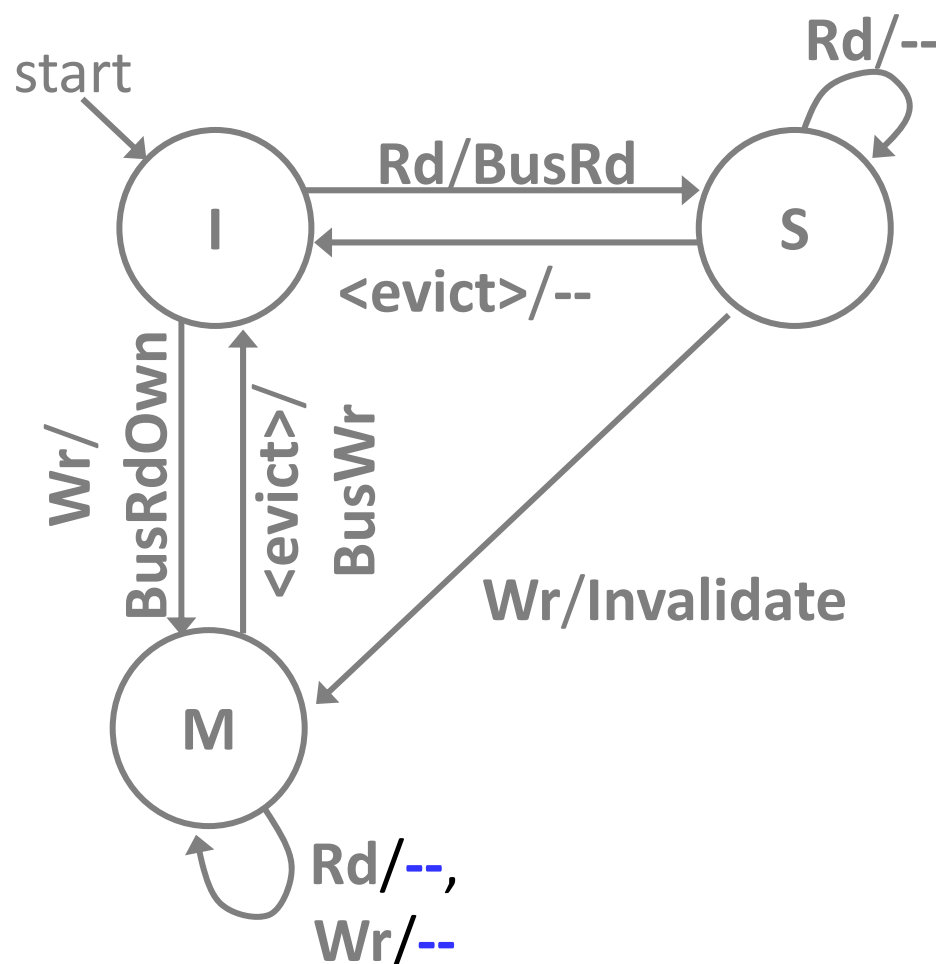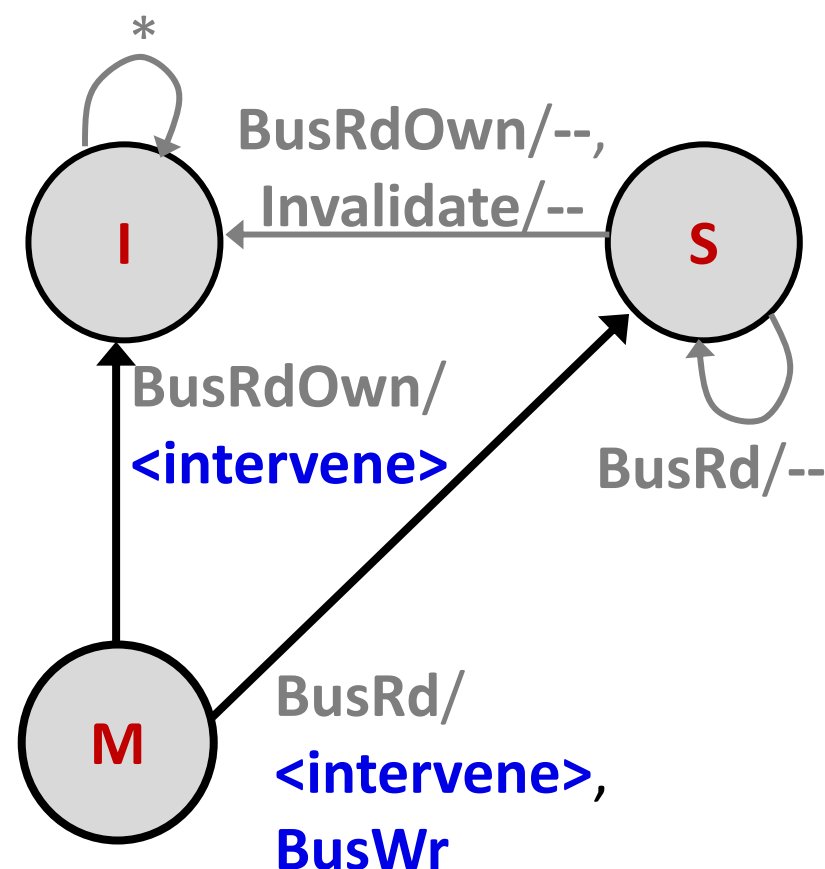
M-copy cache responds in place of DRAM

Nuanced CC States as Optimizations

cache A [**M**] → infer → **other** [**I**] / **memory** [**stale**]
read-write

cache A [**S**] → infer → **other** [**S or I**] / **memory** [**current**]
read-only

cache A [**I**] → infer → **other** [**??**] / **memory** [**??**]
must be valid somewhere
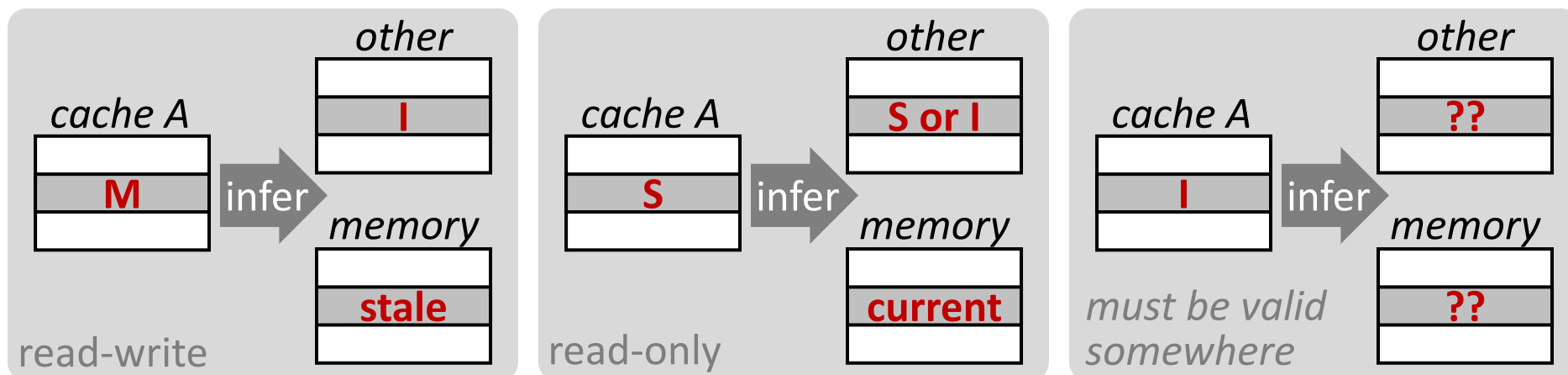
- **Exclusive**, and **Owned** are read-only like **S**, but . . .

cache A [**E**] → infer → **other** [***I***] / **memory** [**current**]

cache A [**O**] → infer → **other** [**S or I**] / **memory** [***stale***]
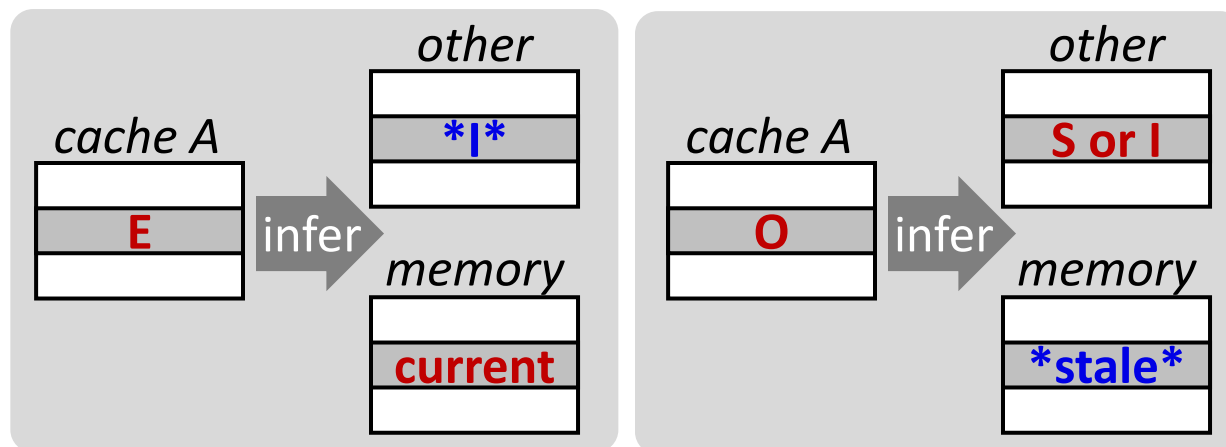
E: silent conversion to **M** or **S** or **I**

O: faster to serve sharers from cache than DRAM

no intelligence attached to DRAM

CC Managed at Block Granularity

- "Embarrassingly parallel" example in HW

```
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 HW, pad psum[] to eliminate **"false sharing"**

Limitations of Snoopy Bus Protocols

- Broadcast bus is not scalable
  - physics dictates big busses expensive and 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?
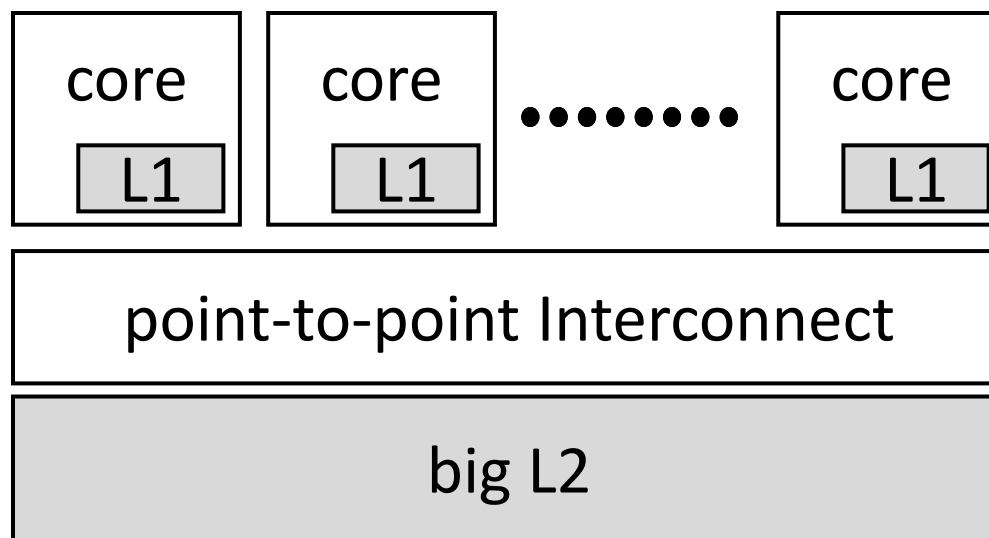
Multicores and Manycores
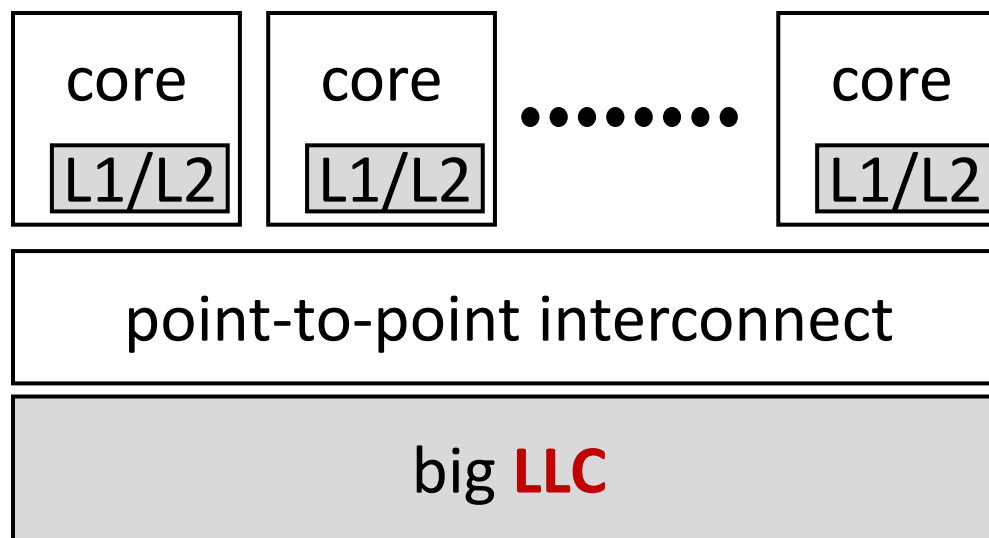
- Private upper-level caches and shared **L**ast-**L**evel **C**ache
- Shared **LLC** typically not inclusive

   total capacity of private caches can add up

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

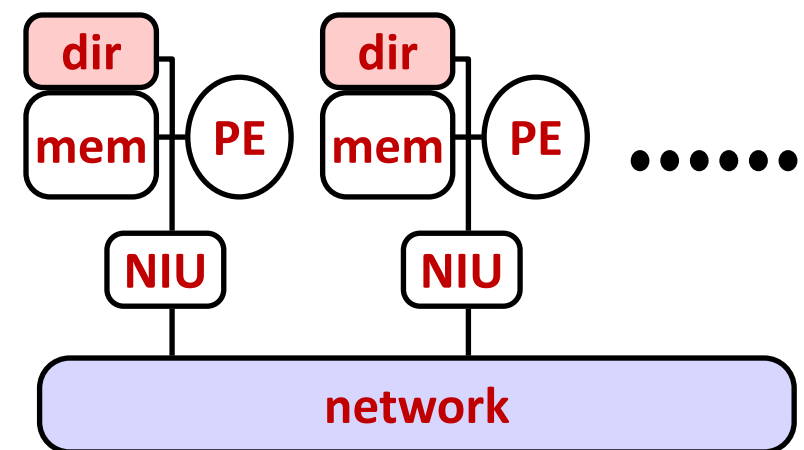# Bookkeeping Instead of Snooping

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
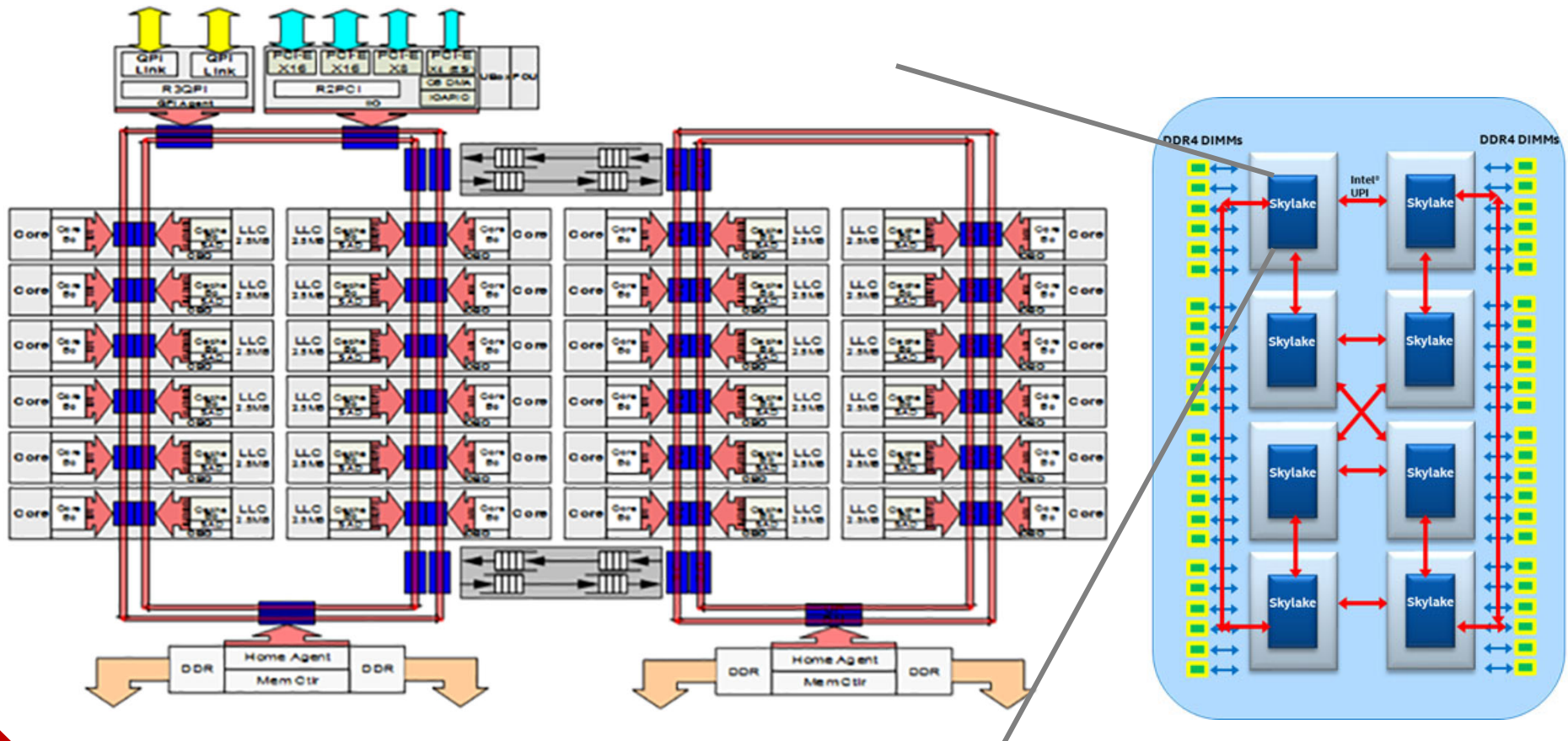
MIMD Shared Memory: Big Irons
Distributed Shared Memory

- 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

18-447-... James C. Hoe, CMU/ECE/CALCM, ©2022

Modern DSM in the small
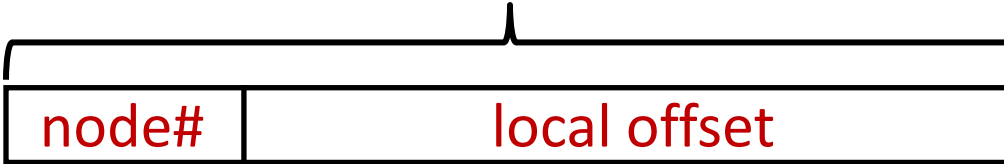
[https://software.intel.com/en-us/articles/intel-xeon-processor-scalable-family-technical-overview]

Global Address Layout

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

Global Physical Address

- Interleaving 1:

node#	local offset

- Interleaving 2:

$offset_{high}$	node#	$offset_{low}$

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

(usually a configurable option)

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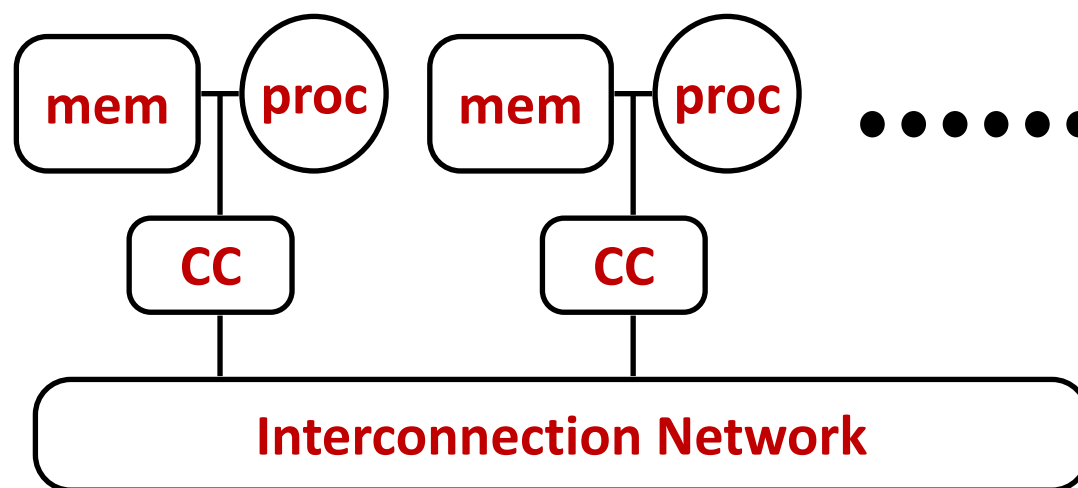
- How to coordinate CC state transitions for large number of far-apart nodes?

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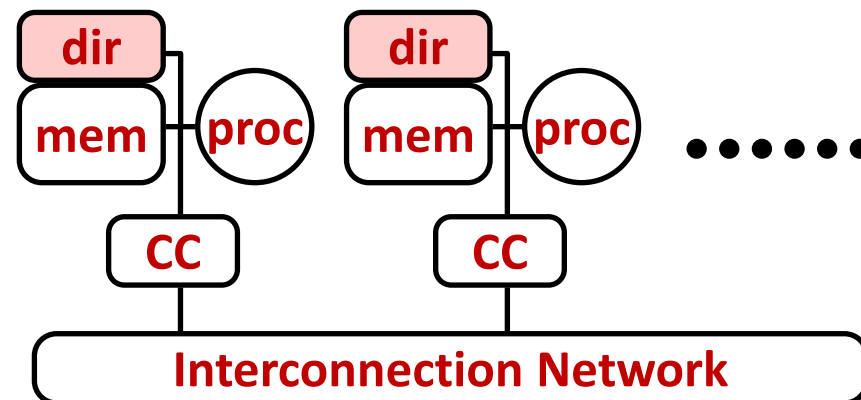
- Avoid unnecessary communication
  - on a cache miss, local CC-controller sends request
    to **home node of address**
  - based on directory information, home-node CC-controller communicates with **only affected nodes**

A Simple Directory Example

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

H	**S**	bit-vector C_i	memory block

⎣____ directory entry ____⎦
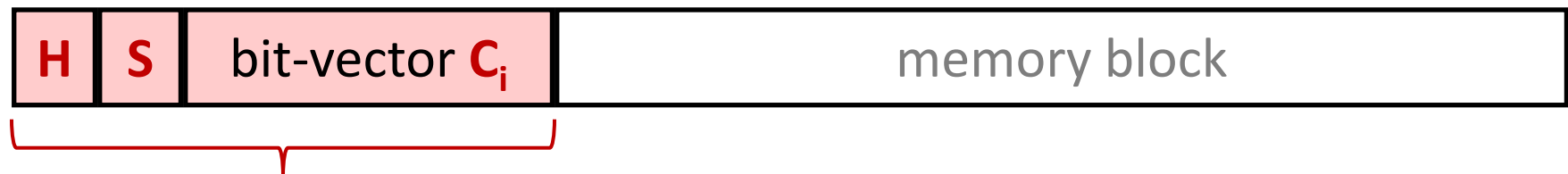
- **H**=1 indicates "at home"; **S**=1 indicates shared
- If **H**=0, C_i bitmaps if node$_i$ has a cached copy
  - uncached (**H**=1, **S**=*): no cached copy exists
  - shared (**H**=0, **S**=1): for all C_i==1, node$_i$ has copy
  - modified (**H**=0, **S**=0): if C_i==1, node$_i$ has only copy

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**

A
X:S

B
X:S

C
X:S

2. S-copy reply

1. **BusRd** *request*

Home

X:H=0;S=1
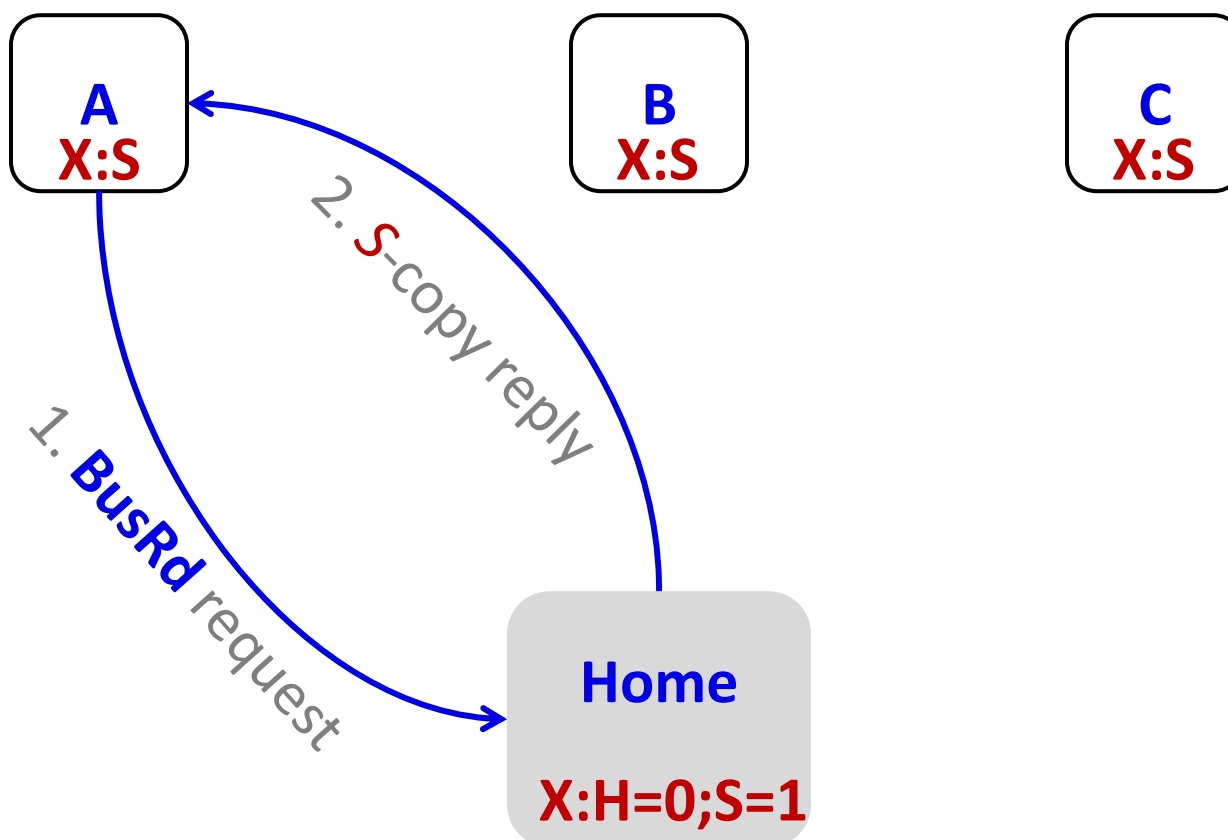
Multi-Hop MSI Protocol Example: Invalidation

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

A
X:M

B
X:S

C
X:S

3. *M*-copy reply

2. **Invalidate** requests

1. **BusRdOwn** request

Home

X:H=0;S=0
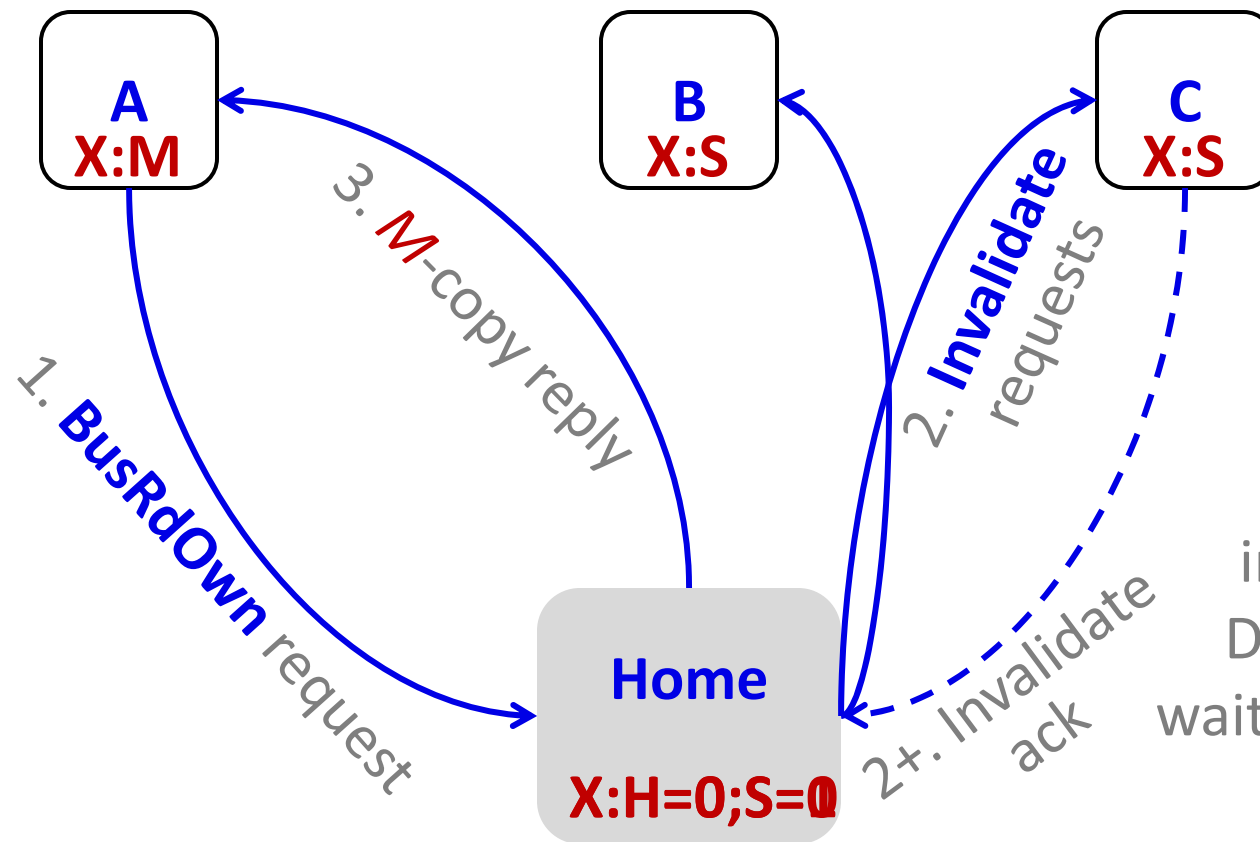
2+. Invalidate ack

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**

A
X:S

B

C
X:S̶M̶

4. *S*-copy reply

2. **Downgrade** request

1. **BusRd** request

3. Downgrade reply (with data)
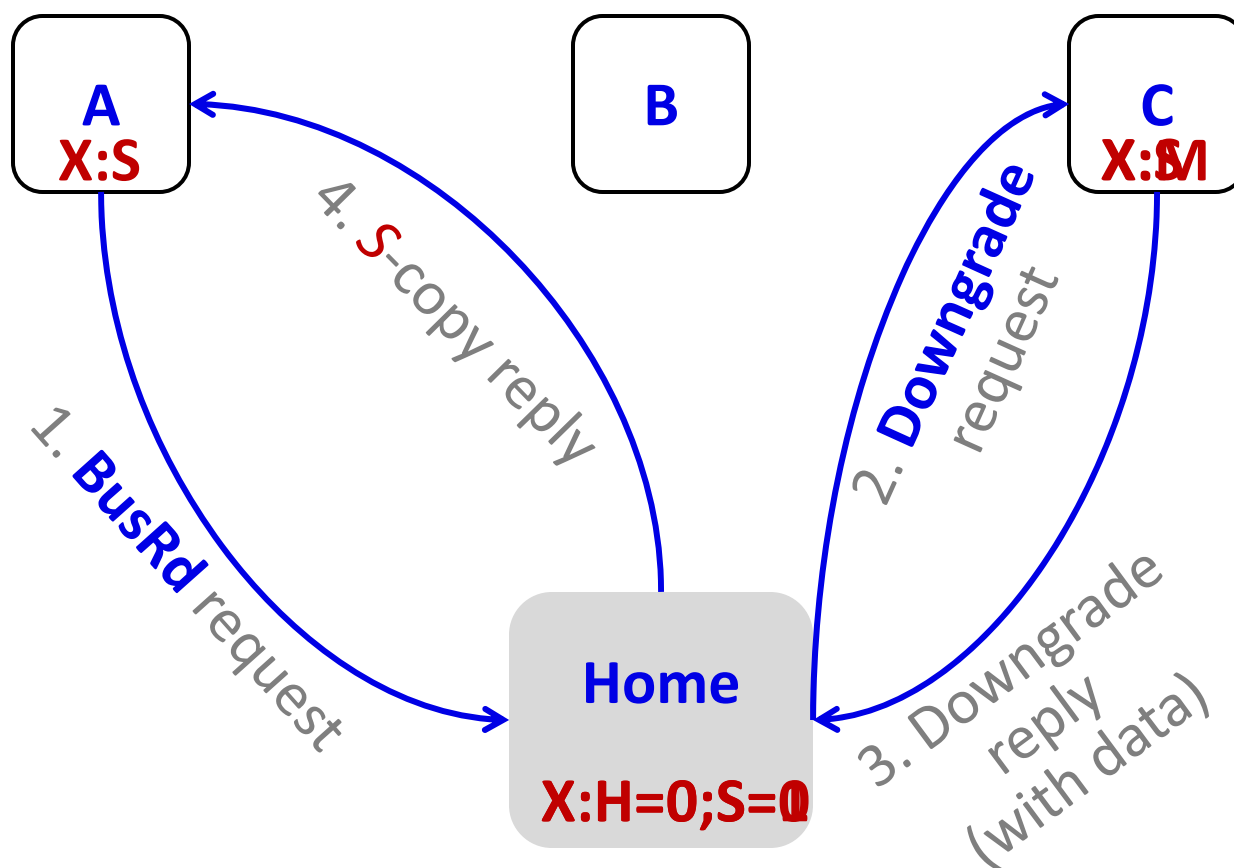
Home
X:H=0;S=0̶1̶

Multi-Hop MSI Protocol Example: Forwarding

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

3. *S*-copy **"forward"**

A
X:S

B

C
X:S

1. **BusRd** request

2. **Downgrade** request

Home

X:H=0;S=1

3. Downgrade reply (with data)
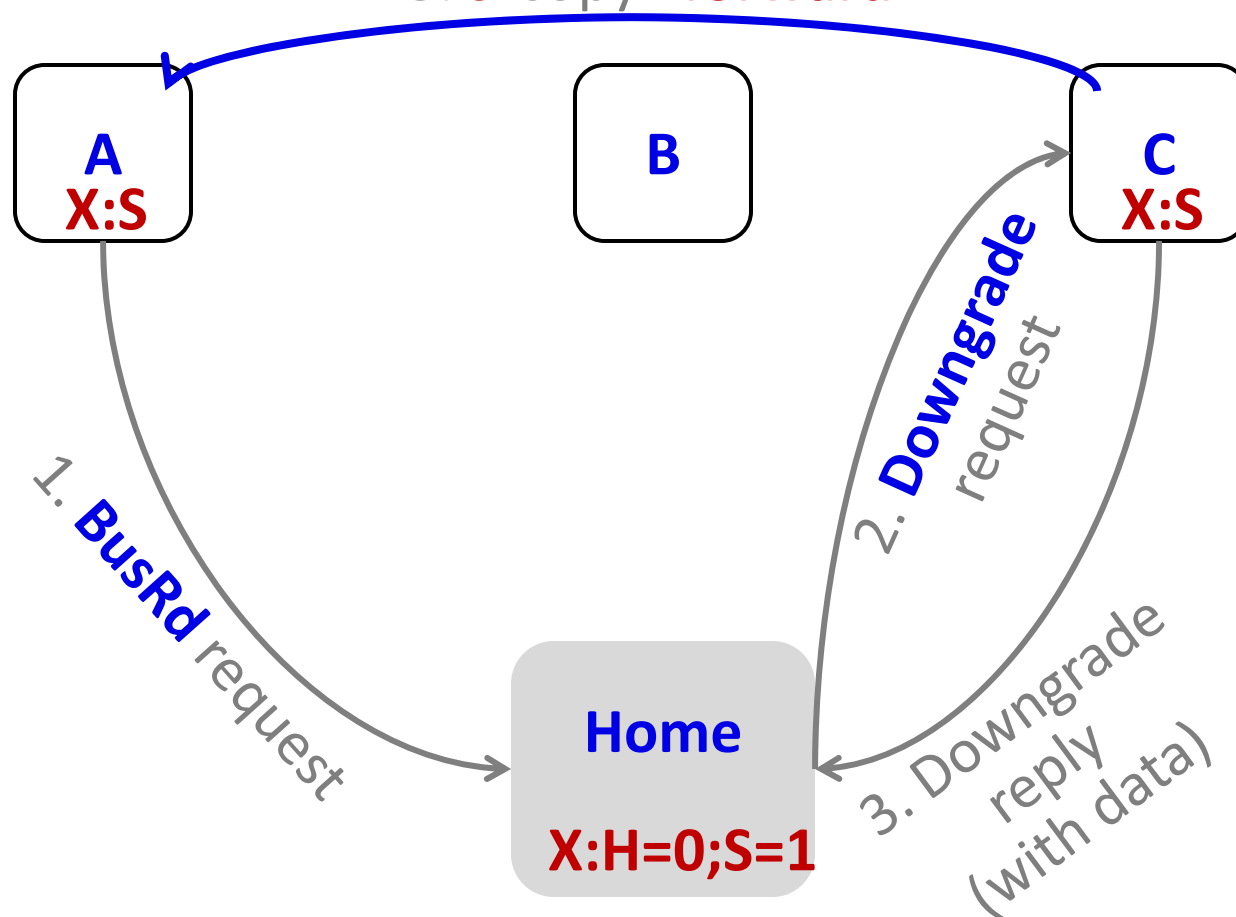
It is much, much harder than it looks

- 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"