

**Lecture 10:**

# **Snooping-Based Cache Coherence**

---

**Parallel Computer Architecture and Programming  
CMU 15-418/15-618, Fall 2023**

# Learning Objectives

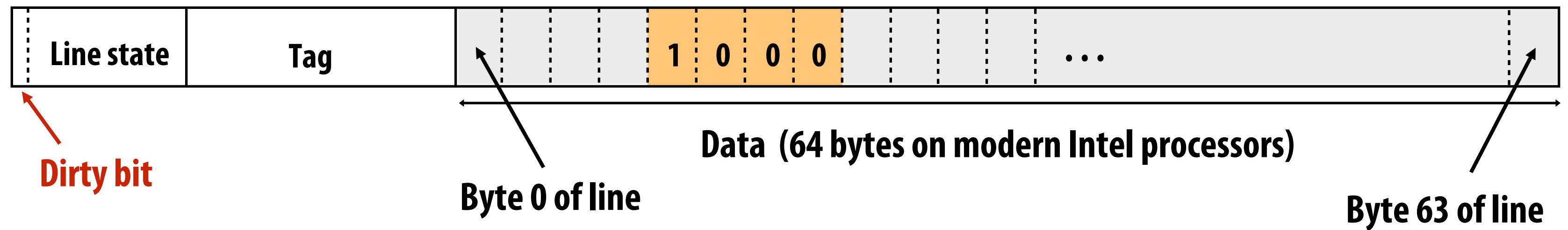
- **Why do caches make memory accesses in a parallel system difficult?**
- **How does hardware provide coordination to preserve software expectations of memory?**
- **When are the hardware abstractions exposed to software?**

# Cache design review

Let's say your code executes `int x = 1;`

(Assume for simplicity `x` corresponds to the address `0x12345604` in memory... it's not stored in a register)

One cache line:



## ■ Review:

- What is the difference between a write back and a write-through cache?
- What about a allocate vs. write-no-allocate cache?

# Review: behavior of write-allocate, write-back cache on a write miss (uniprocessor case)

Example: processor executes `int x = 1;`

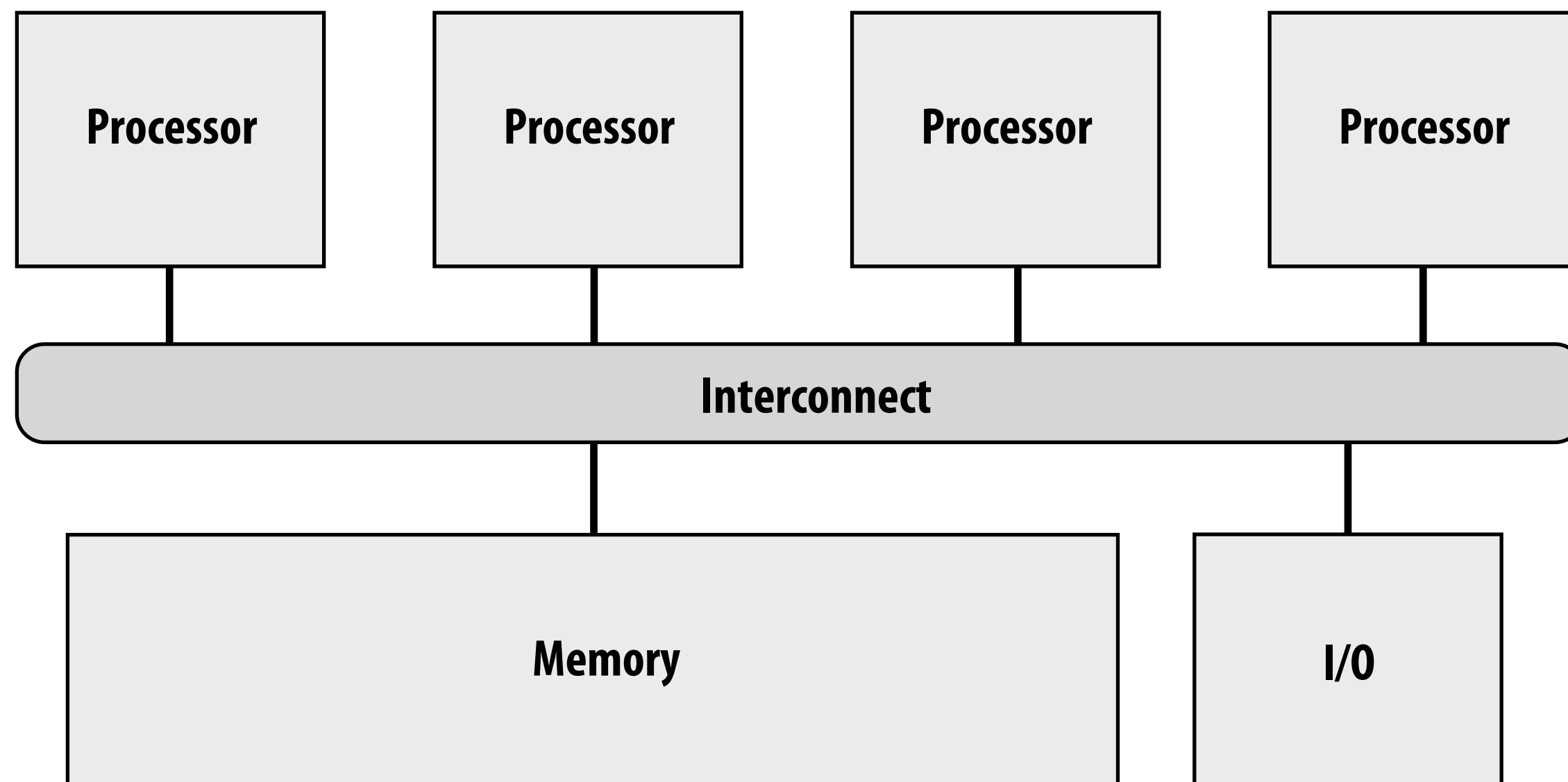
1. Processor performs write to address that is not resident in cache
2. Cache selects location to place line in cache, if there is a dirty line currently in this location, the dirty line is written out to memory
3. Cache loads line from memory (“allocates line in cache”)
4. 32 bits of cache line are updated
5. Cache line is marked as dirty



Dirty bit

# A shared memory multi-processor

- **Processors read and write to shared variables**
  - **More precisely: processors issue load and store instructions**
- **A reasonable expectation of memory is:**
  - **Reading a value at address  $X$  should return the last value written to address  $X$  by any processor**

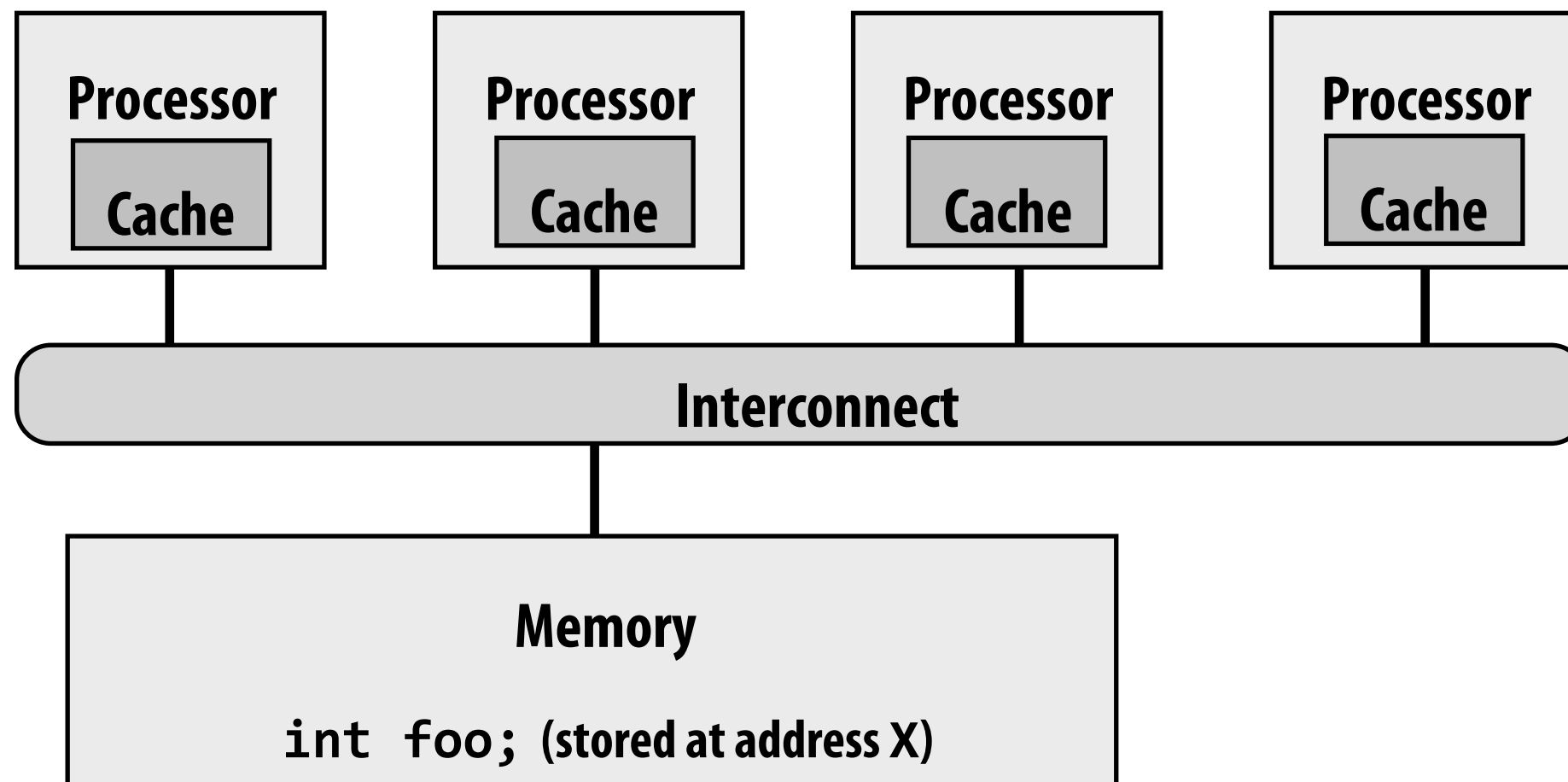


**(A simple view of four processors and their shared address space)**

# The cache coherence problem

Modern processors replicate contents of memory in local caches

**Problem: processors can observe different values for the same memory location**



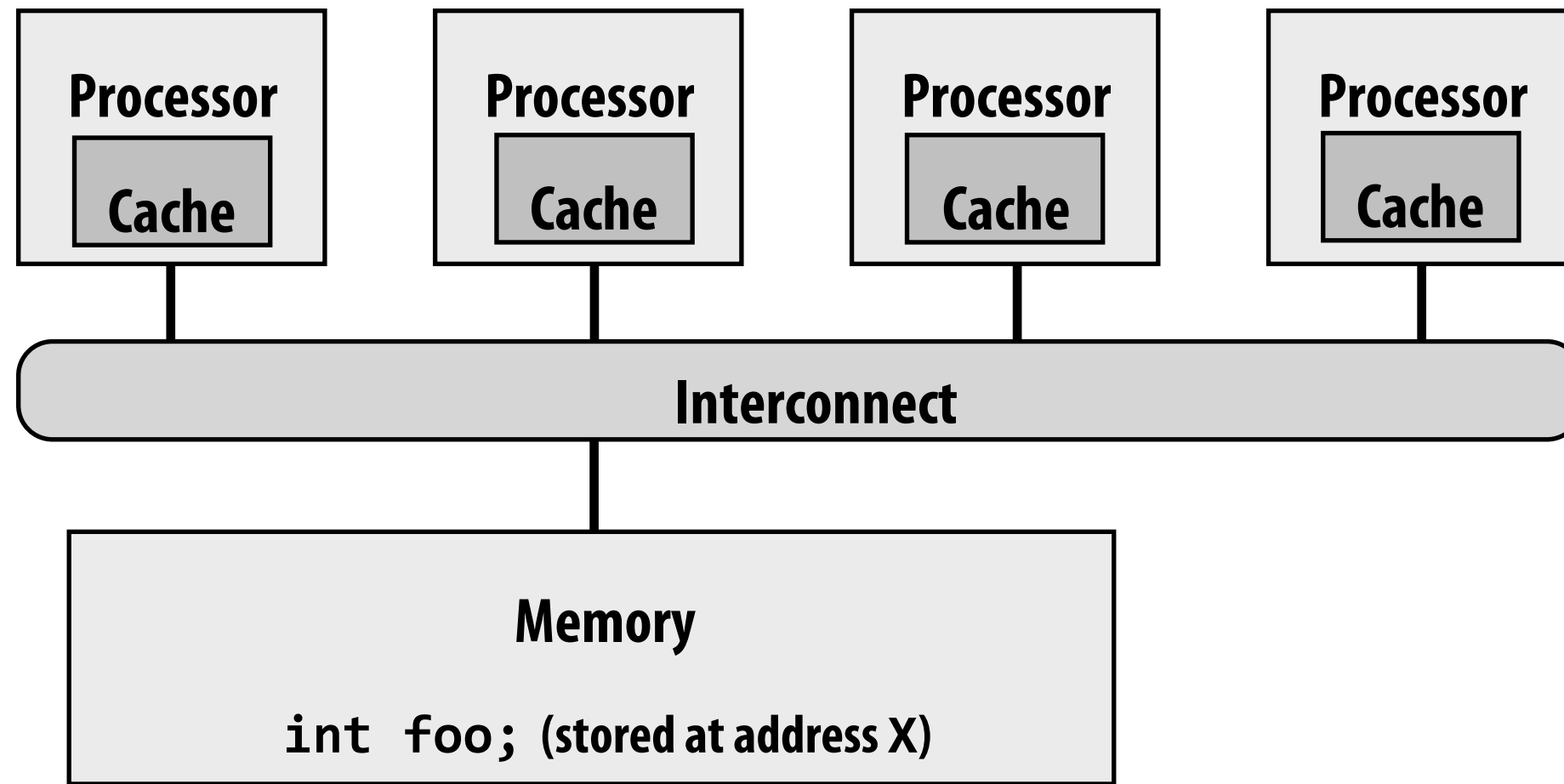
The chart at right shows the value of variable `foo` (stored at address `X`) in main memory and in each processor's cache

Assume the initial value stored at address `X` is 0

Assume write-back cache behavior

Action	P1 \$	P2 \$	P3 \$	P4 \$	mem[X]
					0
P1 load X	0 miss				0
P2 load X	0	0 miss			0
P1 store X	1	0			0
P3 load X	1	0	0 miss		0
P3 store X	1	0	2		0
P2 load X	1	0 hit	2		0
P1 load Y (assume this load causes eviction of X)		0	2		1

# The cache coherence problem



Is this a mutual exclusion problem?

Can you fix the problem by adding locks to your program?

**NO!**

This is a problem created by replicating the data stored at address X in local caches (a hardware implementation detail)

The chart at right shows the value of variable foo (stored at address X) in main memory and in each processor's cache

Assume the initial value stored at address X is 0

Assume write-back cache behavior

Action	P1 \$	P2 \$	P3 \$	P4 \$	mem[X]
					0
P1 load X	0 miss				0
P2 load X	0	0 miss			0
P1 store X	1	0			0
P3 load X	1	0	0 miss		0
P3 store X	1	0	2		0
P2 load X	1	0 hit	2		0
P1 load Y (assume this load causes eviction of X)		0	2		1

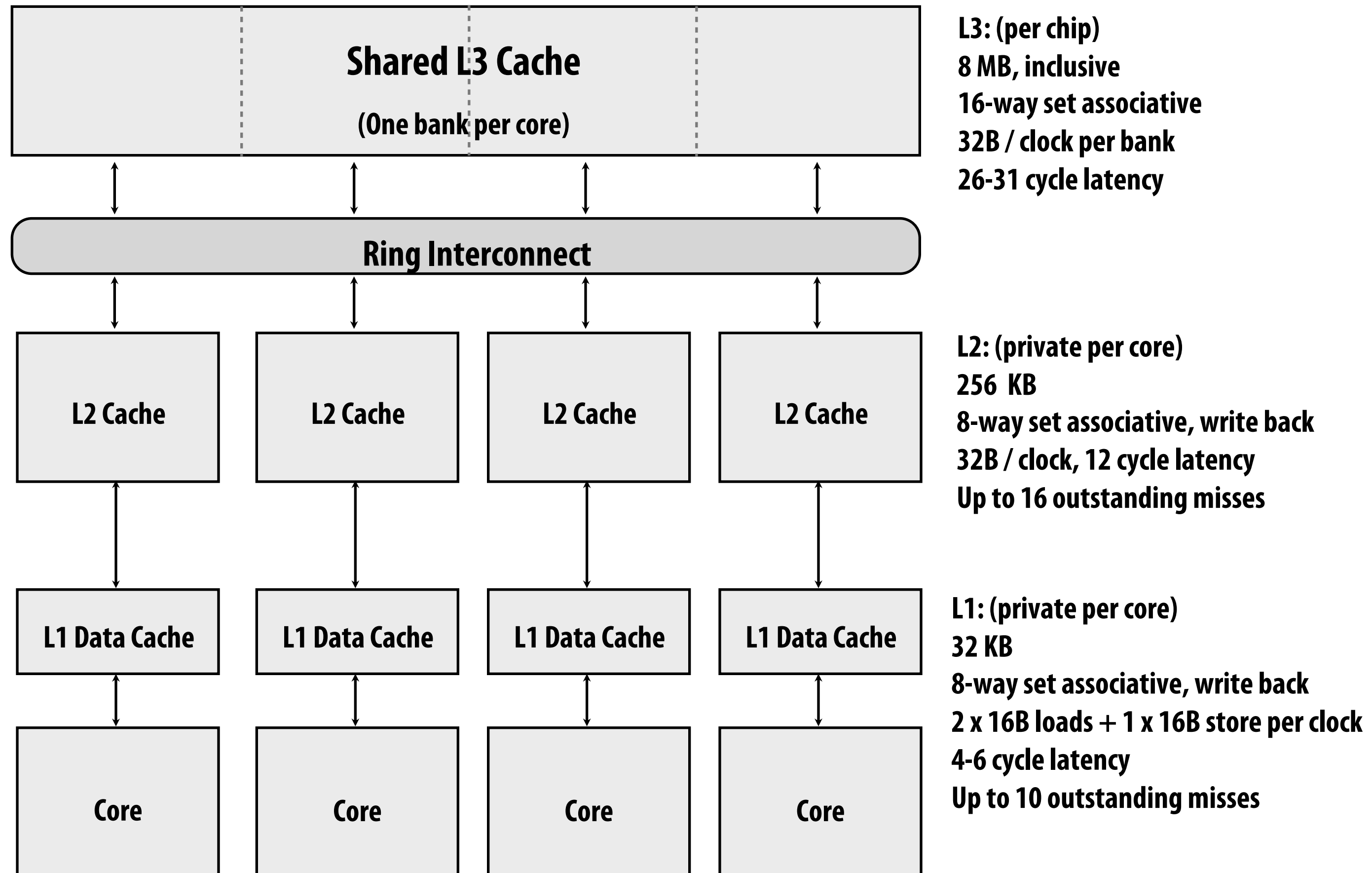
# The memory coherence problem

- **Intuitive behavior for memory system: reading value at address  $X$  should return the last value written to address  $X$  *by any processor*.**
- **Memory coherence problem exists because there is both global storage (main memory) and per-processor local storage (processor caches) implementing the abstraction of a single shared address space.**



# Cache hierarchy of Intel Haswell CPU (2013)

64 byte cache line size

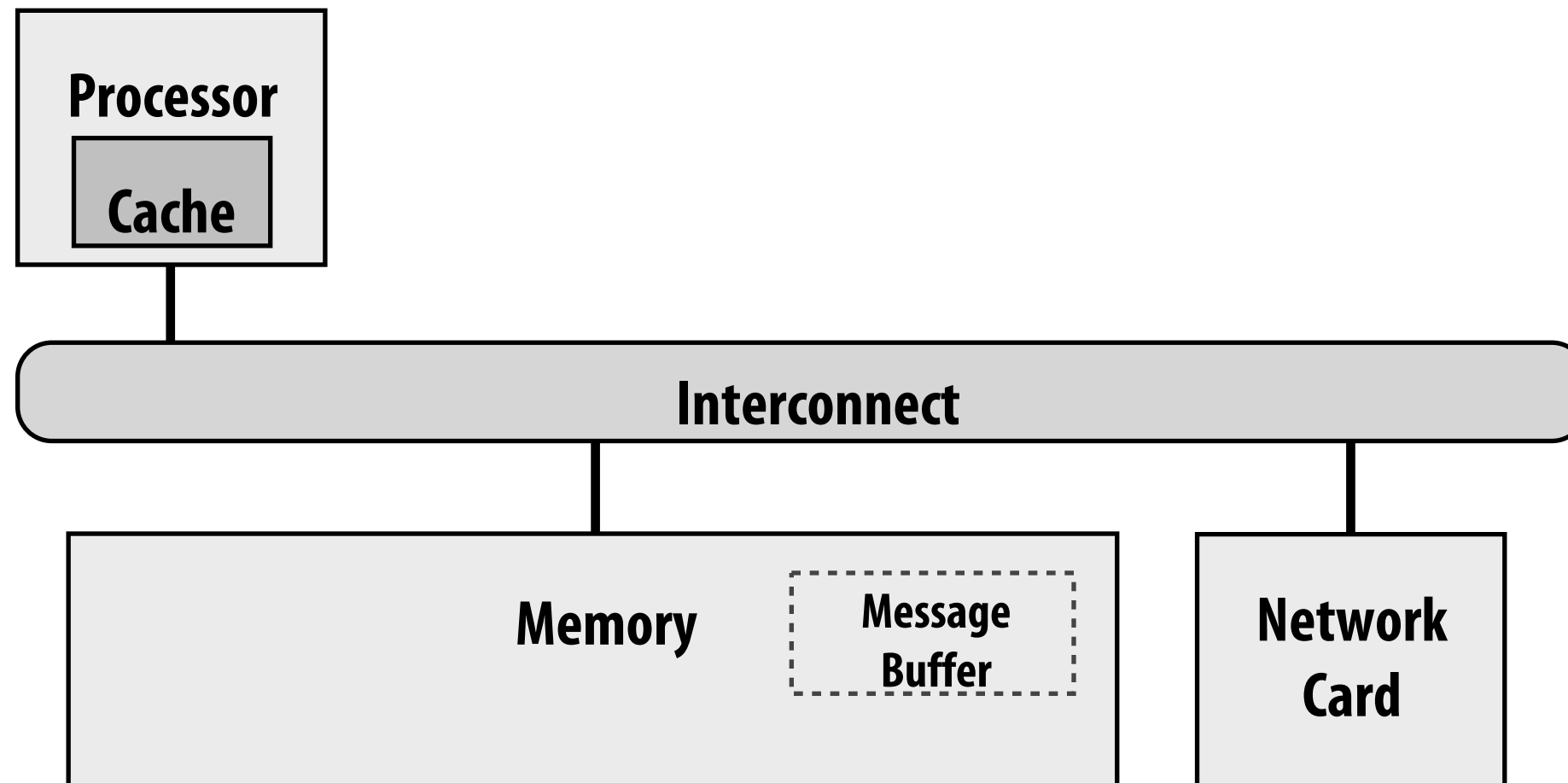


# Intuitive expectation of shared memory

- **Intuitive behavior for memory system: reading value at address  $X$  should return the last value written to address  $X$  by any processor.**
- **On a uniprocessor, providing this behavior is fairly simple, since writes typically come from one client: the processor**
  - **Exception: device I/O via direct memory access (DMA)**

# Coherence is an issue in a single CPU system

Consider I/O device performing DMA data transfer



Case 1:

Processor writes to buffer in main memory

Processor tells network card to async send buffer

**Problem: network card may transfer stale data if processor's writes (reflected in cached copy of data) are not flushed to memory**

Case 2:

Network card receives message

Network card copies message in buffer in main memory using DMA transfer

Card notifies CPU msg was received, buffer ready to read

**Problem: CPU may read stale data if addresses updated by network card happen to be in cache**

- **Common solutions:**
  - CPU writes to shared buffers using uncached stores (e.g., driver code)
  - OS support:
    - Mark virtual memory pages containing shared buffers as not-cachable
    - Explicitly flush pages from cache when I/O completes
- In practice, DMA transfers are infrequent compared to CPU loads and stores (so these heavyweight software solutions are acceptable)

# Problems with the intuition

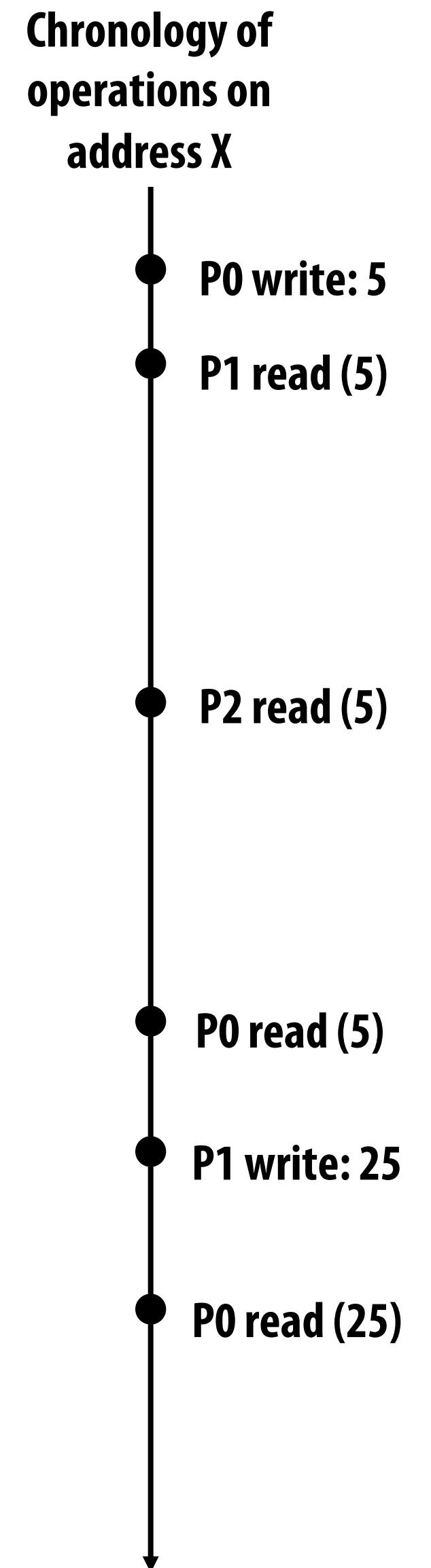
- **Intuitive behavior: reading value at address  $X$  should return the last value written to address  $X$  by any processor.**
- **What does “last” mean?**
  - **What if two processors write at the same time?**
  - **What if a write by P1 is followed by a read from P2 so close in time that it is impossible to communicate the occurrence of the write to P2 in time?**
- **In a sequential program, “last” is determined by program order (not time)**
  - **Holds true within one thread of a parallel program**
  - **But we need to come up with a meaningful way to describe order across threads in a parallel program**

# Definition: coherence

A memory system is coherent if:

**The results of a parallel program's execution are such that for each memory location, there is a hypothetical serial order of all program operations (executed by all processors) to the location that is consistent with the results of execution, and:**

- 1. Memory operations issued by any one processor occur in the order issued by the processor**
- 2. The value returned by a read is the value written by the last write to the location... as given by the serial order**



# Definition: coherence (said differently)

A memory system is coherent if:

- 1. A read by processor P to address X that follows a write by P to address X, should return the value of the write by P** *(assuming no other processor wrote to X in between)*
- 2. A read by processor P1 to address X that follows a write by processor P2 to X returns the written value... if the read and write are “sufficiently separated” in time** *(assuming no other write to X occurs in between)*
- 3. Writes to the same address are serialized: two writes to address X by any two processors are observed in the same order by all processors.**

*(Example: if values 1 and then 2 are written to address X, no processor observes X having value 2 before value 1)*

**Condition 1: obeys program order (as expected of a uniprocessor system)**

**Condition 2: “write propagation”:** Notification of a write must eventually get to the other processors. Note that precisely when information about the write is propagated is not specified in the definition of coherence.

**Condition 3: “write serialization”**

# Write serialization

**Writes to the same location are serialized: two writes to address  $X$  by any two processors are observed in the same order by all processors.**

*(Example: if a processor observes  $X$  having value 1 and then 2, then no processor observes  $X$  having value 2 before it has value 1)*

**Example: P1 writes value  $a$  to  $X$ . Then P2 writes value  $b$  to  $X$ .**

**Consider situation where processors P3 and P4 observe different order of writes:**

**Order observed by P3**

**ld  $X$   $\rightarrow$  load returns "a"**

**$\vdots$**

**ld  $X$   $\rightarrow$  load returns "b"**

**Order observed by P4**

**ld  $X$   $\rightarrow$  load returns "b"**

**$\vdots$**

**ld  $X$   $\rightarrow$  load returns "a"**

**In terms of the first coherence definition: there is no global ordering of loads and stores to  $X$  that is in agreement with results of this parallel program.**

**(you cannot put the two memory operations involving  $X$  on a single timeline and have both processor's observations agree with the timeline)**

# Implementing coherence

## ■ **Software-based solutions**

- **OS uses page-fault mechanism to propagate writes**
- **Can be used to implement memory coherence over clusters of workstations**
- **We won't discuss these solutions**

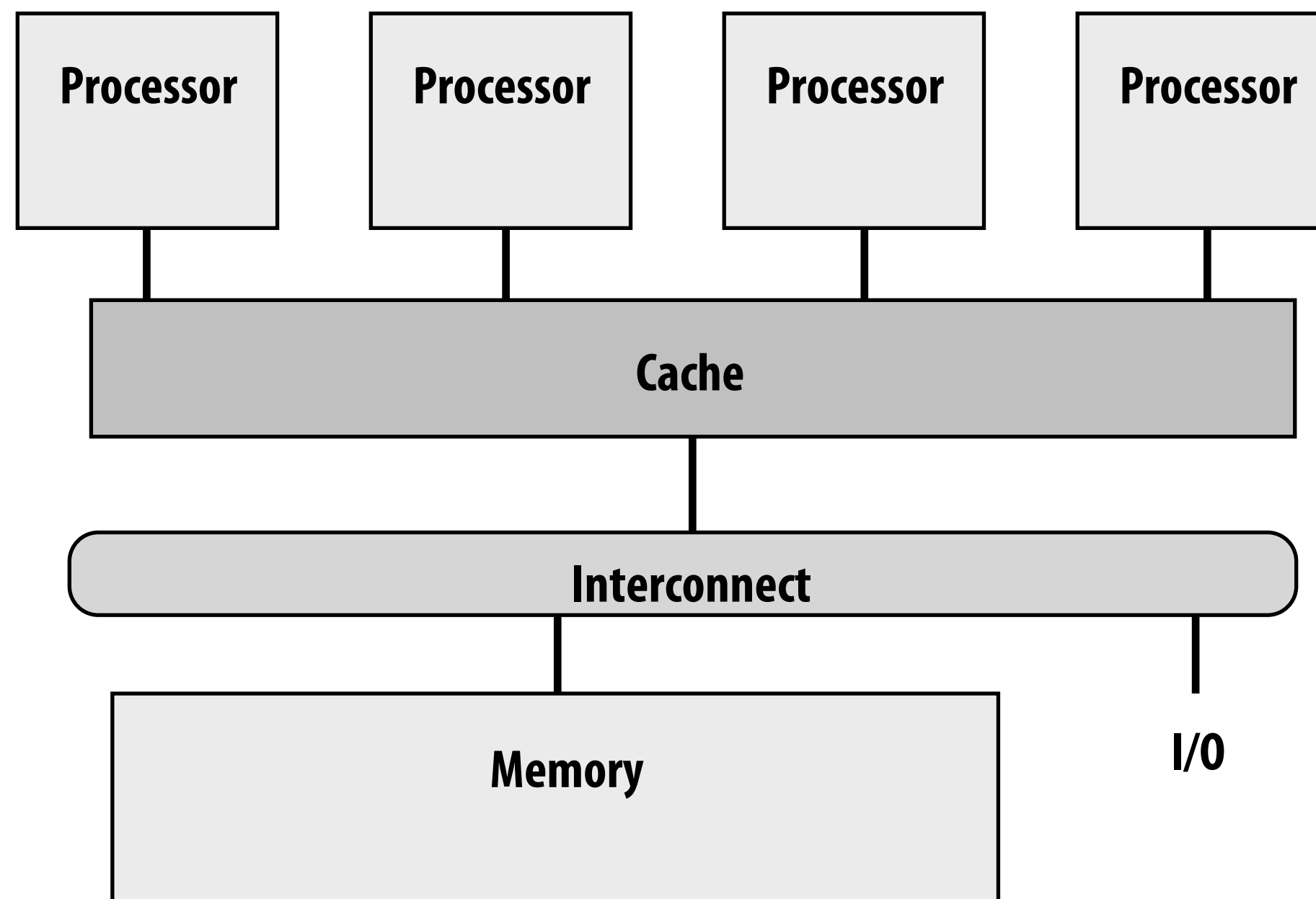
## ■ **Hardware-based solutions**

- **"Snooping"-based coherence implementations (today)**
- **Directory-based coherence implementations (next class)**



# Shared caches: coherence made easy

- **One single cache shared by all processors**
  - Eliminates problem of replicating state in multiple caches
- **Obvious scalability problems (since the point of a cache is to be local and fast)**
  - Interference / contention due to many clients
- **But shared caches can have benefits:**
  - Facilitates fine-grained sharing (overlapping working sets)
  - Loads/stores by one processor might pre-fetch lines for another processor



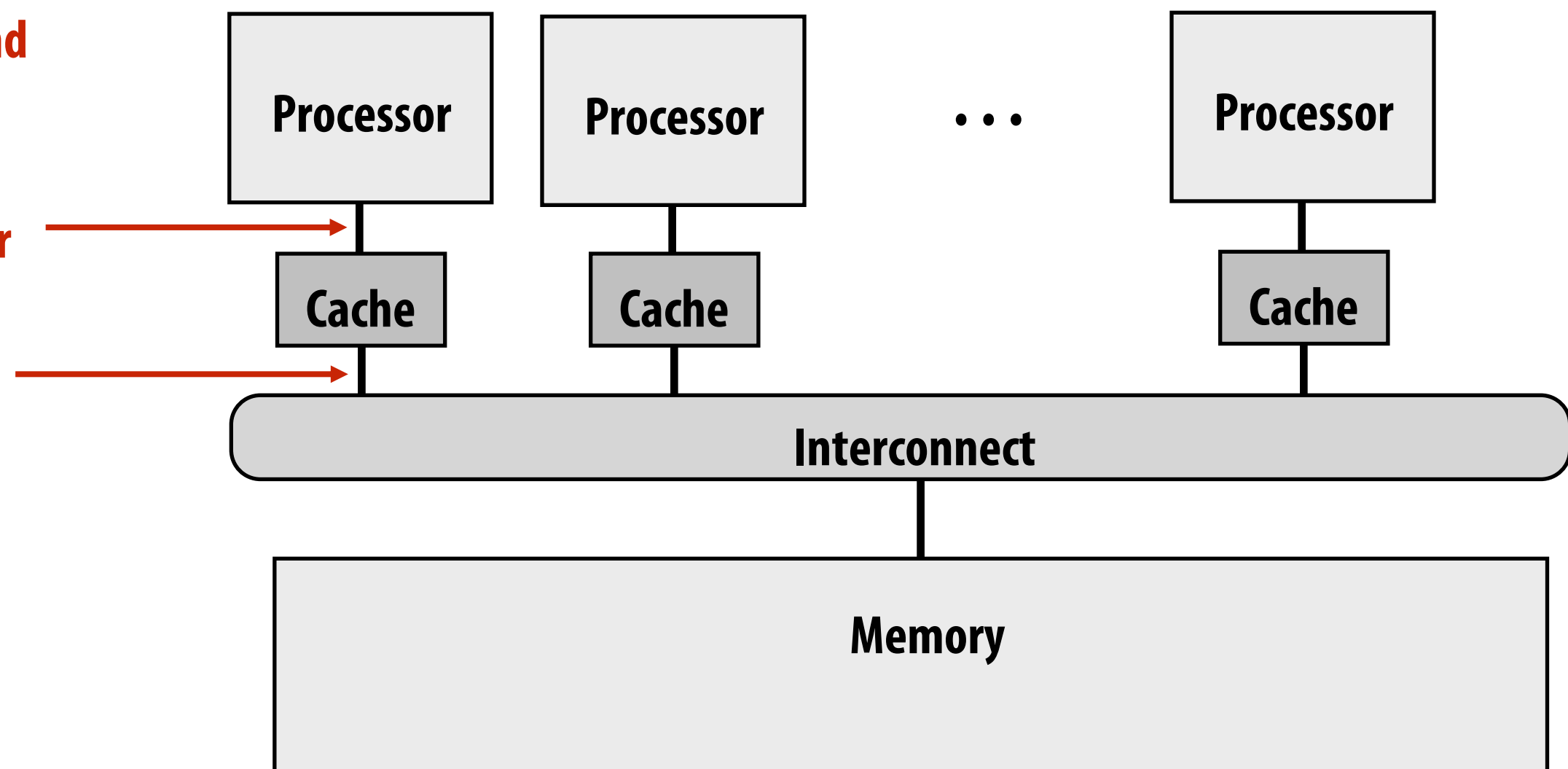
# Snooping cache-coherence schemes

- Main idea: all coherence-related activity is broadcast to all processors in the system (more specifically: to the processor's cache controllers)
- Cache controllers monitor ("they snoop") memory operations, and react accordingly to maintain memory coherence

Notice: now cache controller must respond to actions from "both ends":

1. LD/ST requests from its local processor

2. Coherence-related activity broadcast over the chip's interconnect



# Very simple coherence implementation

Let's assume:

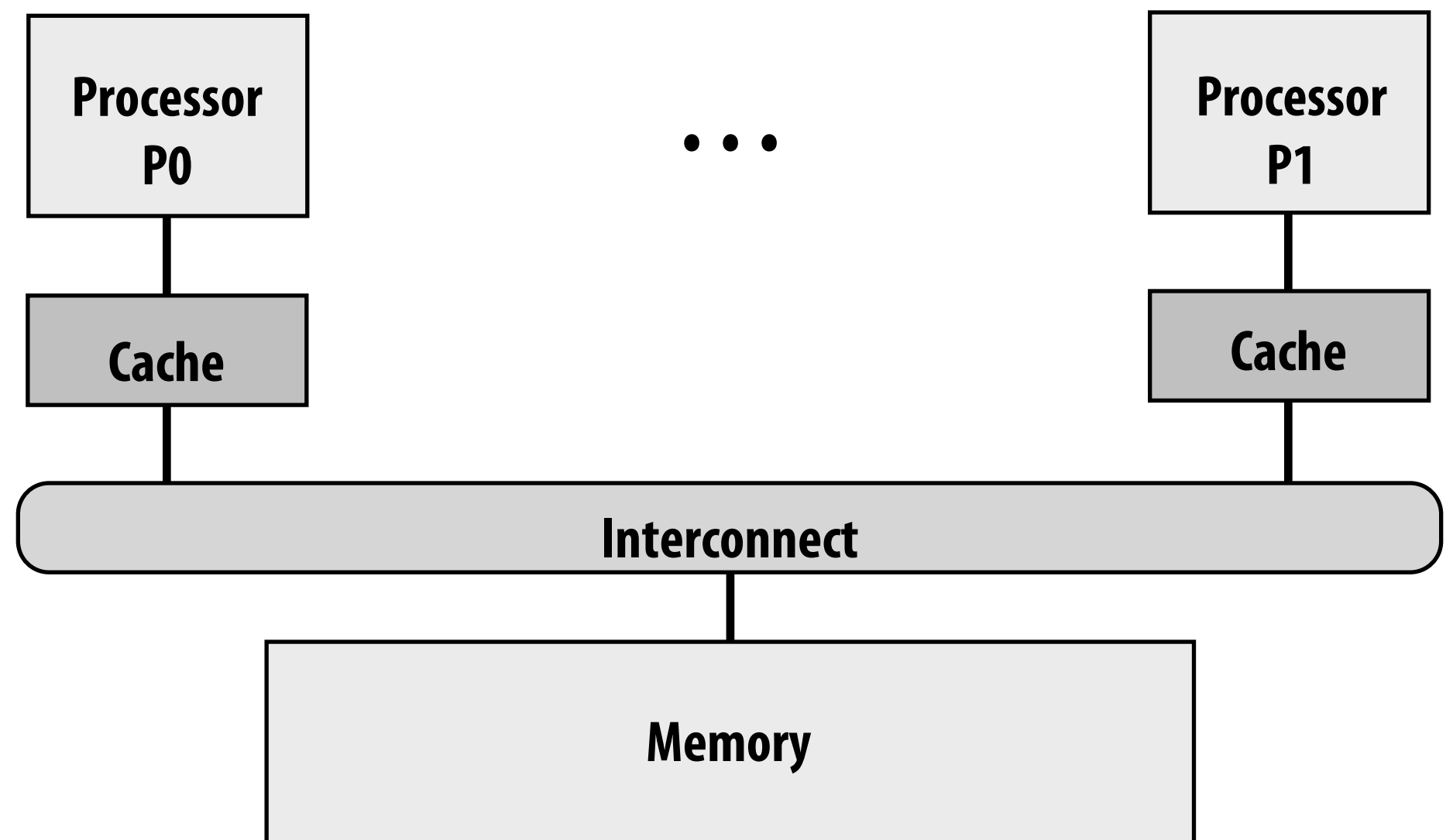
1. **Write-through** caches

2. Granularity of coherence is cache line

Upon write, cache controller broadcasts  
invalidation message

As a result, the next read from other  
processors will trigger cache miss

(processor retrieves updated value from memory due to  
write-through policy)

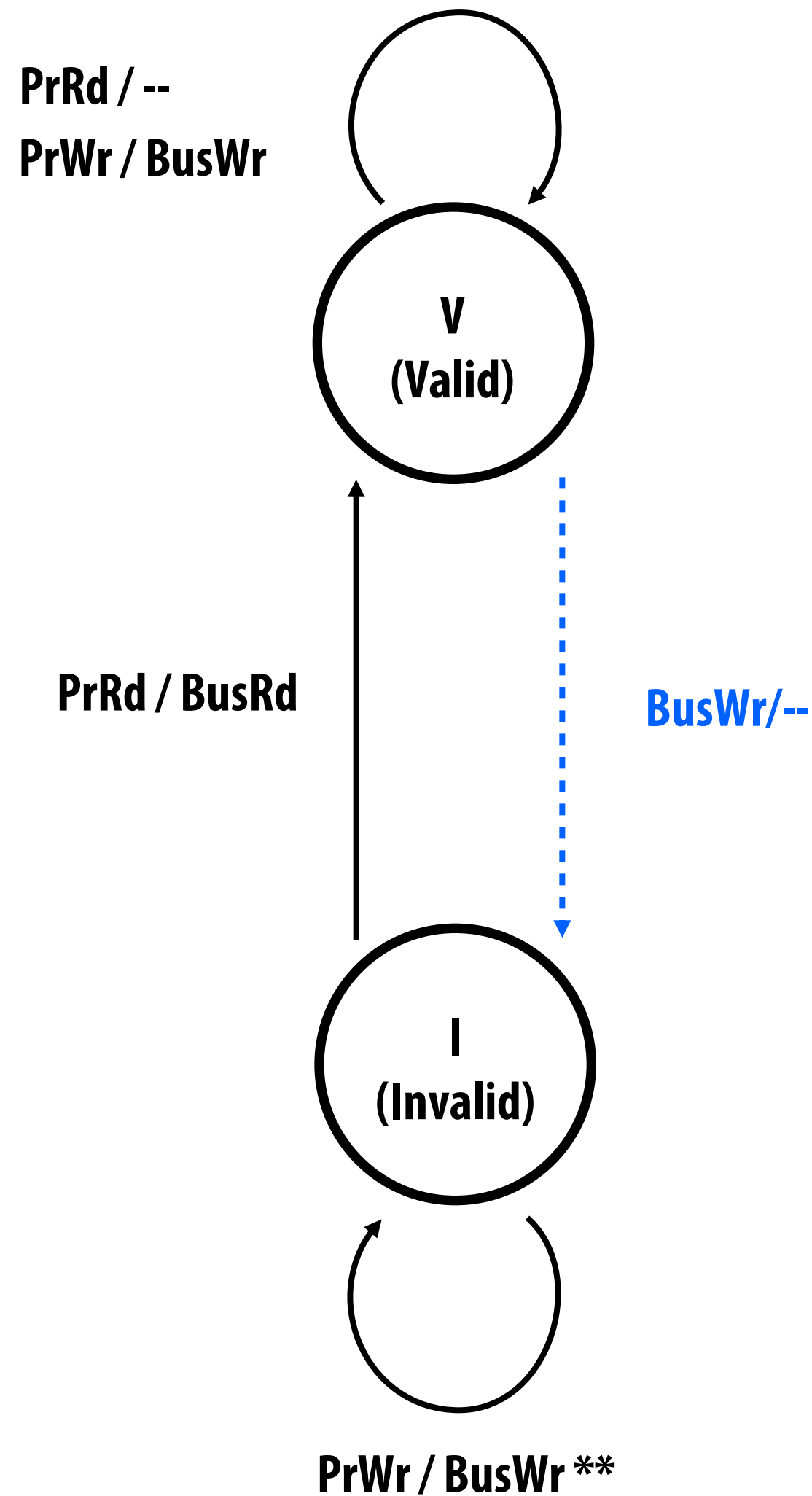


Action	Interconnect activity	P0 \$	P1 \$	mem location X
				0
P0 load X	cache miss for X	0		0
P1 load X	cache miss for X	0	0	0
P0 write 100 to X	<b>invalidation for X</b>	100		100
P1 load X	cache miss for X	100	100	100

# A clarifying note

- **The logic we are about to describe is performed by each processor's cache controller in response to:**
  - **Loads and stores by the local processor**
  - **Messages it receives from other caches**
  
- **If all cache controllers operate according to this described protocol, then coherence will be maintained**
  - **The caches “cooperate” to ensure coherence is maintained**

# Write-through invalidation: state diagram



- **Two cache line states (same as meaning of invalid in uniprocessor cache)**
  - Invalid (I)
  - Valid (V)
- **Two processor operations (triggered by local processor)**
  - PrRd (read)
  - PrWr (write)
- **Two bus transactions (from remote caches)**
  - BusRd (another processor intends to read line)
  - BusWr (another processor intends to write to line)\*

## Notation:

A / B: if event A is observed by cache controller, then action B is taken

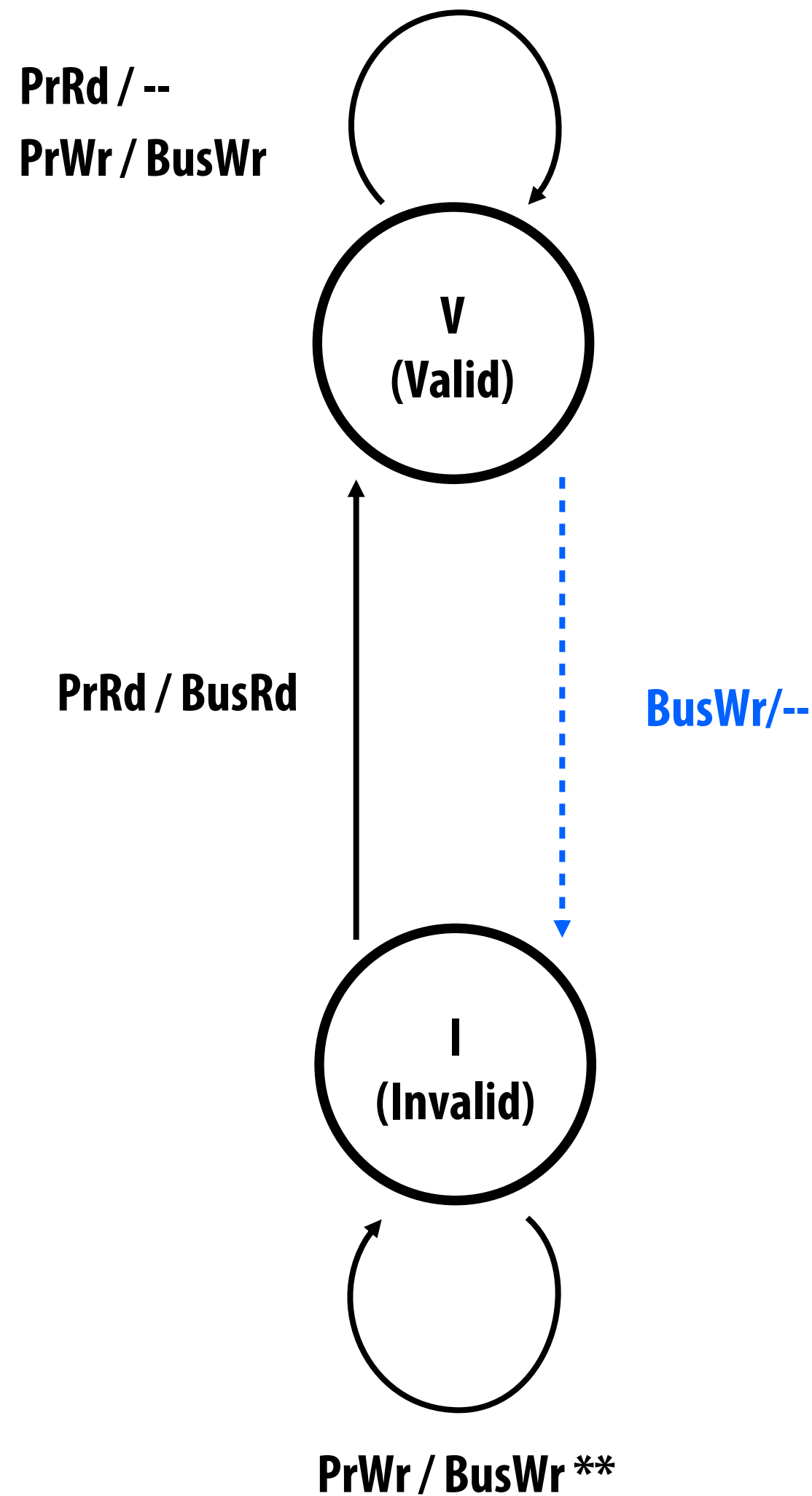
-----> Remote processor (coherence) initiated transaction

————> Local processor initiated transaction

\*\* Assumes write no-allocate policy (for simplicity)

\* BusWr also called BusRdX

# Write-through invalidation: state diagram



## Requirements of the interconnect:

1. All write transactions visible to all cache controllers
2. All write transactions visible to all cache controllers in the same order

## Simplifying assumptions here:

1. Interconnect and memory transactions are atomic
2. Processor waits until previous memory operations is complete before issuing next memory operation
3. Invalidation applied immediately as part of receiving invalidation broadcast

A / B: if action A is observed by cache controller, action B is taken

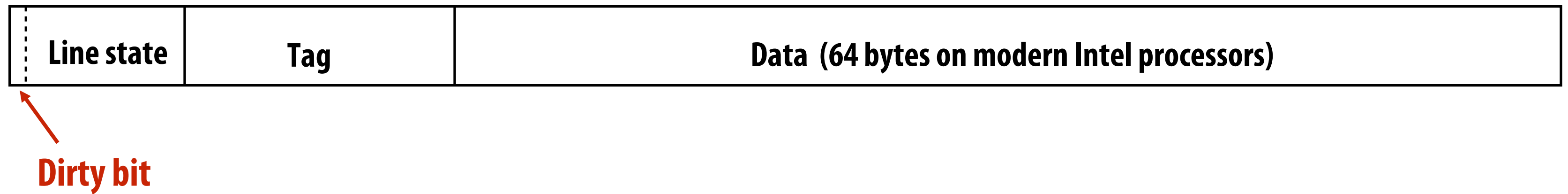
-----> Remote processor (coherence) initiated transaction

-----> Local processor initiated transaction

# Write-through policy is inefficient

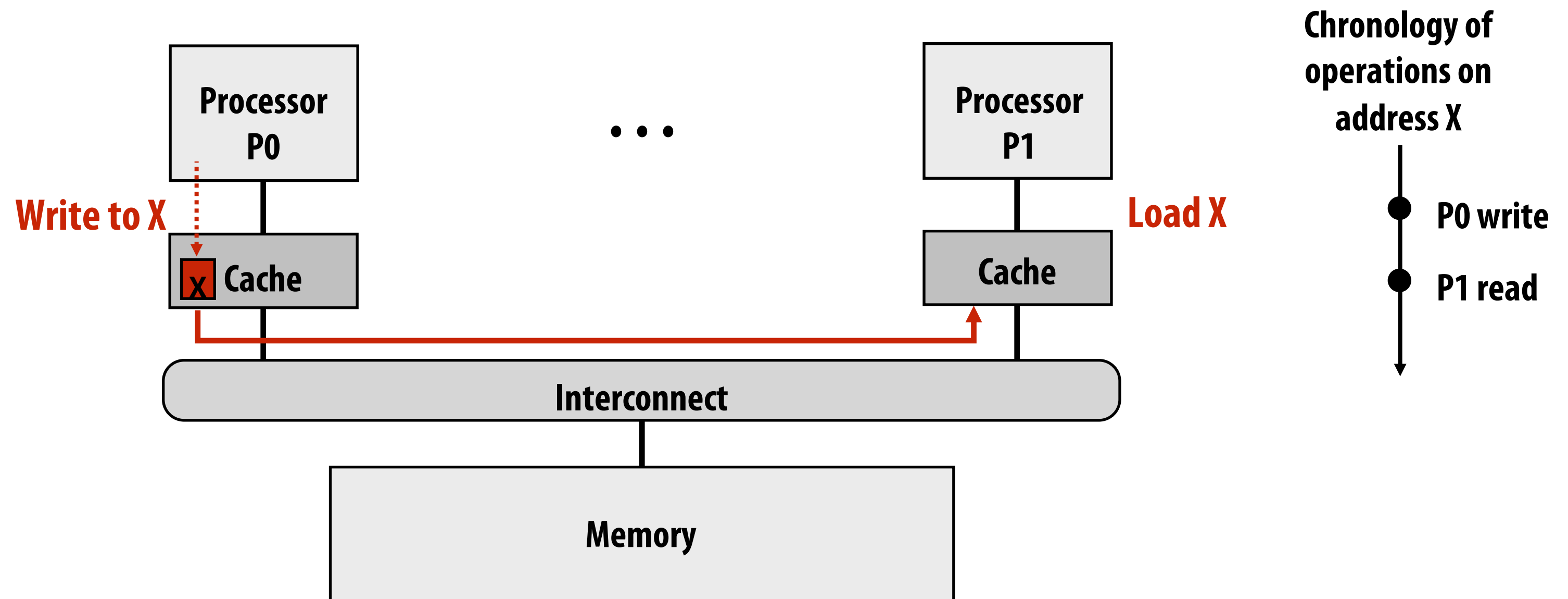
- **Every write operation goes out to memory**
  - **Very high bandwidth requirements**
  
- **Write-back caches absorb most write traffic as cache hits**
  - **Significantly reduces bandwidth requirements**
  - **But now how do we ensure write propagation/serialization?**
  - **This requires more sophisticated coherence protocols**

# Recall cache line state bits





# Cache coherence with write-back caches



- **Dirty state of cache line now indicates exclusive ownership**
  - **Exclusive:** cache is only cache with a valid copy of line (it can safely be written to)
  - **Owner:** cache is responsible for supplying the line to other processors when they attempt to load it from memory (otherwise a load from another processor will get stale data from memory)

# Invalidation-based write-back protocol

## Key ideas:

- **A line in the “exclusive” state can be modified without notifying the other caches**
- **Processor can only write to lines in the exclusive state**
  - So they need a way to tell other caches that they want exclusive access to the line
  - They will do this by sending all the other caches messages
- **When cache controller snoops a request for exclusive access to line it contains**
  - It must invalidate the line in its own cache

# MSI write-back invalidation protocol

## ■ Key tasks of protocol

- Ensuring processor obtains exclusive access for a write
- Locating most recent copy of cache line's data on cache miss

## ■ Three cache line states

- Invalid (I): same as meaning of invalid in uniprocessor cache
- Shared (S): line valid in one or more caches
- Modified (M): line valid in exactly one cache (a.k.a. "dirty" or "exclusive" state)

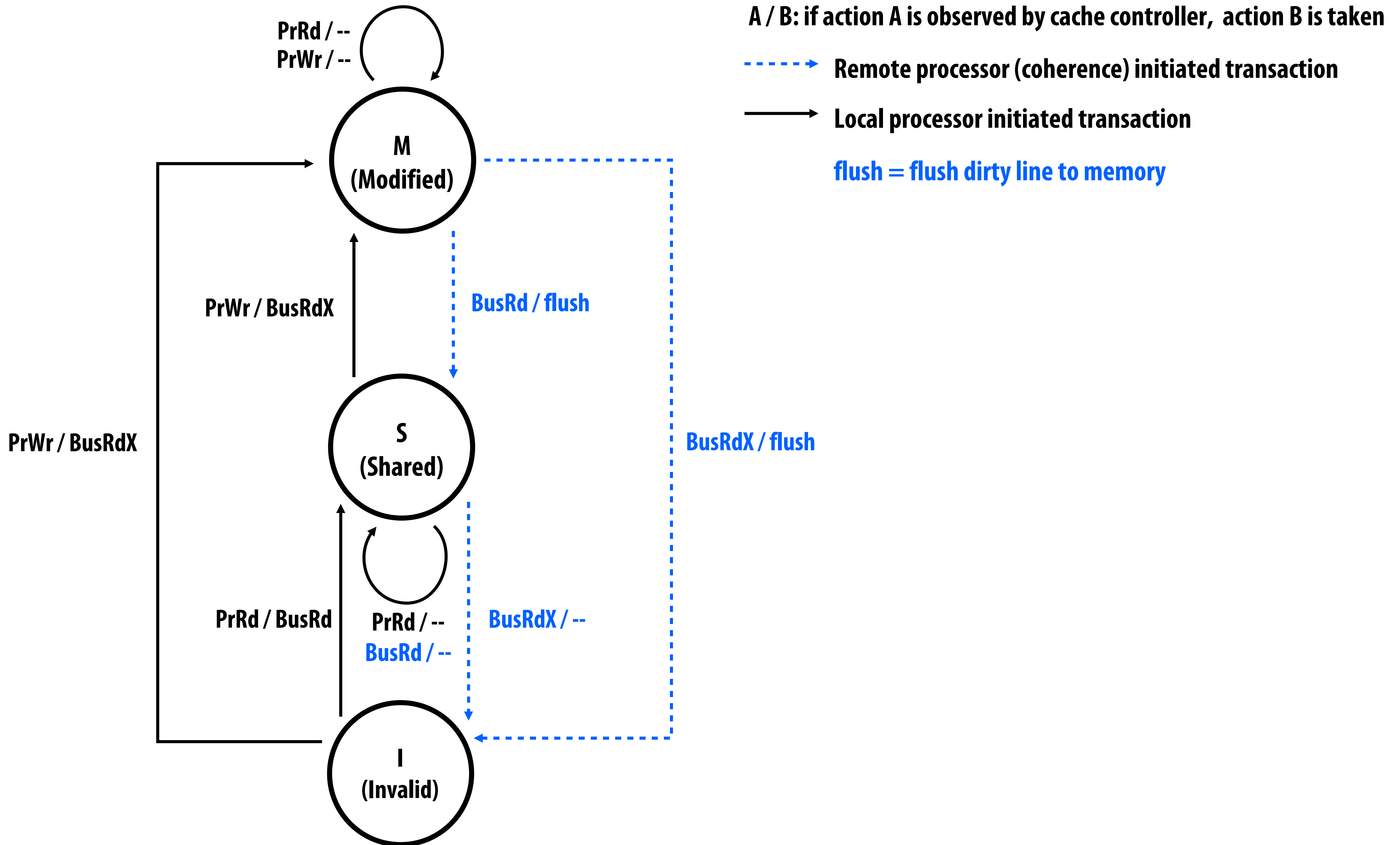
## ■ Two processor operations (triggered by local CPU)

- PrRd (read)
- PrWr (write)

## ■ Three coherence-related bus transactions (from remote caches)

- BusRd: obtain copy of line with no intent to modify
- BusRdX: obtain copy of line with intent to modify
- flush: write dirty line out to memory

# MSI state transition diagram \*



\* Remember, all caches are carrying out this logic independently to maintain coherence

# Example Execution

Action	P0 X	P0 Y	P1 X	P1 Y
Initial	I	I	I	I
P0: LD X	S/O			
P1: LD X				
P0: ST X ← 1				
P0: ST X ← 2				
P1: ST X ← 3				
P1: LD X				
P0: LD X				
P0: ST X ← 4				
P1: LD X				
P0: LD Y				
P0: ST Y ← 1				
P1: ST Y ← 2				

**X and Y have value 0 at start of execution.**

# Example Execution

Action	P0 X	P0 Y	P1 X	P1 Y
Initial	I	I	I	I
P0: LD X	S/0			
P1: LD X	S/0		S/0	
P0: ST X ← 1	M/1		I	
P0: ST X ← 2	M/2			
P1: ST X ← 3	I		M/3	
P1: LD X			M/3	
P0: LD X	S/3		S/3	
P0: ST X ← 4	M/4		I	
P1: LD X	S/4		S/4	
P0: LD Y		S/0		
P0: ST Y ← 1	S/4	M/1	S/4	
P1: ST Y ← 2		I		M/2

X and Y have value 0 at start of execution.

# Summary: MSI

- **A line in the M state can be modified without notifying other caches**
  - No other caches have the line resident, so other processors cannot read these values (without generating a memory read transaction)
- **Processor can only write to lines in the M state**
  - If processor performs a write to a line that is not exclusive in cache, cache controller must first broadcast a read-exclusive transaction to move the line into that state
  - Read-exclusive tells other caches about impending write (“you can’t read any more, because I’m going to write”)
  - Read-exclusive transaction is required even if line is valid (but not exclusive... it’s in the S state) in processor’s local cache (why?)
  - Dirty state implies exclusive
- **When cache controller snoops a “read exclusive” for a line it contains**
  - Must invalidate the line in its cache
  - Because if it didn’t, then multiple caches will have the line (and so it wouldn’t be exclusive in the other cache!)

# Does MSI satisfy coherence?

## ■ Write propagation

- Achieved via combination of invalidation on BusRdX, and flush from M-state on subsequent BusRd/BusRdX from another processors

## ■ Write serialization

- Writes that appear on interconnect are ordered by the order they appear on interconnect (BusRdX)
- Reads that appear on interconnect are ordered by order they appear on interconnect (BusRd)
- Writes that don't appear on the interconnect (PrWr to line already in M state):
  - Sequence of writes to line comes between two interconnect transactions for the line
  - All writes in sequence performed by same processor, P (that processor certainly observes them in correct sequential order)
  - All other processors observe notification of these writes only after a interconnect transaction for the line. So all the writes come before the transaction.
  - So all processors see writes in the same order.



# MESI invalidation protocol

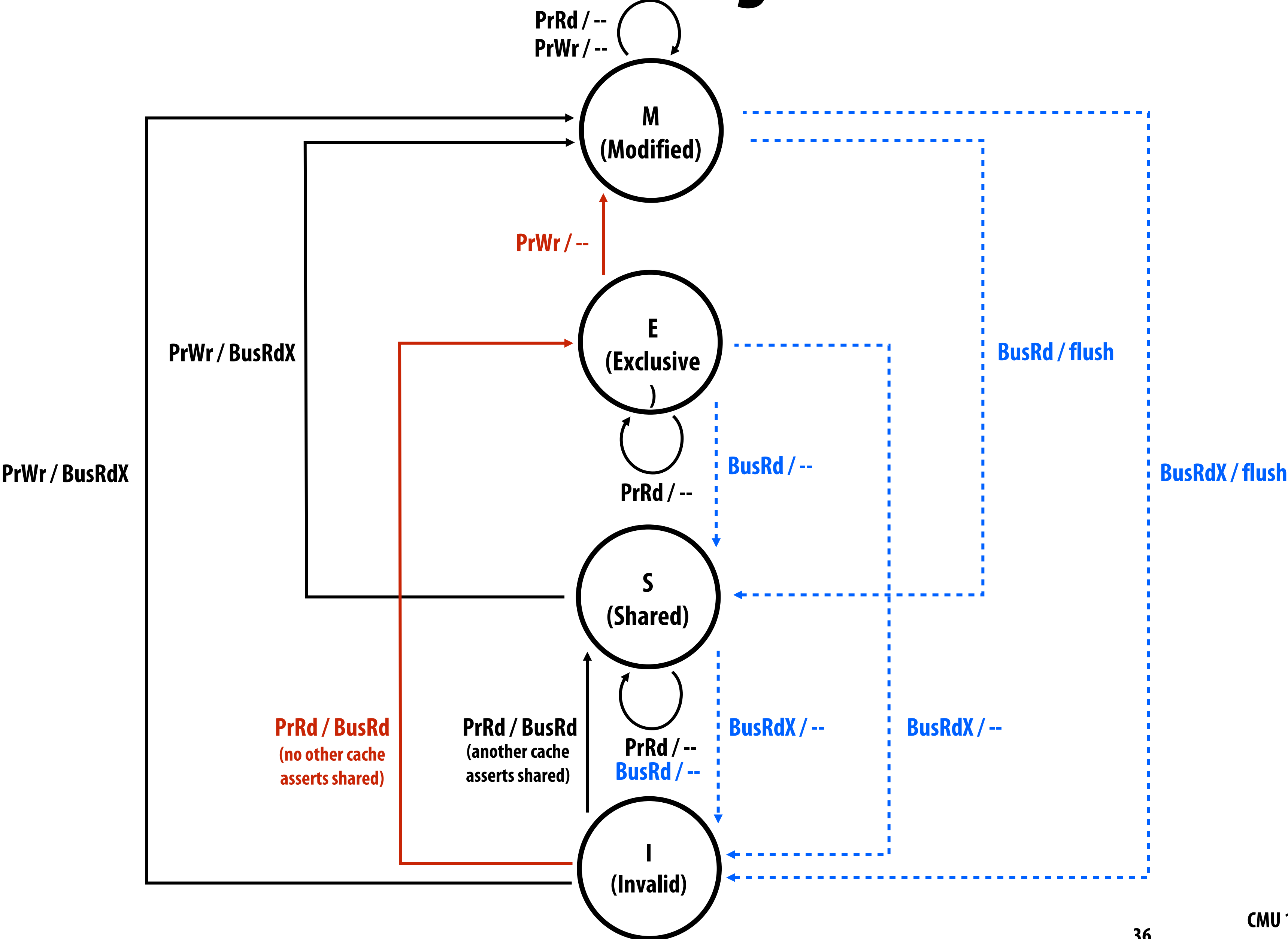
Also known as the Illinois Protocol

- **MSI requires two interconnect transactions for the common case of reading an address, then writing to it**
  - Transaction 1: BusRd to move from I to S state
  - Transaction 2: BusRdX to move from S to M state
- **This inefficiency exists even if application has no sharing at all**
- **Solution: add additional state E (“exclusive clean”)**
  - Line has not been modified, but only this cache has a copy of the line
  - Decouples exclusivity from line ownership (line not dirty, so copy in memory is valid copy of data)
  - Upgrade from E to M does not require an interconnect transaction



MESI, not Messi!

# MESI state transition diagram



# Example Execution

**X and Y have value 0 at start of execution.**

Action	P0 X	P0 Y	P1 X	P1 Y
Initial	0	0	0	0
P0: LD X				
P1: LD X				
P0: ST X ← 1				
P0: ST X ← 2				
P1: ST X ← 3				
P0: LD Y				
P0: LD X				
P0: ST Y ← 4				
P1: LD Y				

# Example Execution

X and Y have value 0 at start of execution.

Action	P0 X	P0 Y	P1 X	P1 Y
Initial	I	I	I	I
P0: LD X	E/0			
P1: LD X	S/0		S/0	
P0: ST X ← 1	M/1		I	
P0: ST X ← 2	M/2			
P1: ST X ← 3	I		M/3	
P0: LD Y		E/0		
P0: LD X	S/3		S/3	
P0: ST Y ← 4		M/4		
P1: LD Y		S/4		S/4

# Lower-level choices

- **Who should supply data on a cache miss when line is in the E or S state of another cache?**
  - **Can get cache line data from memory or can get data from another cache**
  - **If source is another cache, which one should provide it?**
- **Cache-to-cache transfers add complexity, but commonly used to reduce both latency of data access and reduce memory bandwidth required by application**

# Increasing efficiency (and complexity)

## ■ MESIF (5-stage invalidation-based protocol)

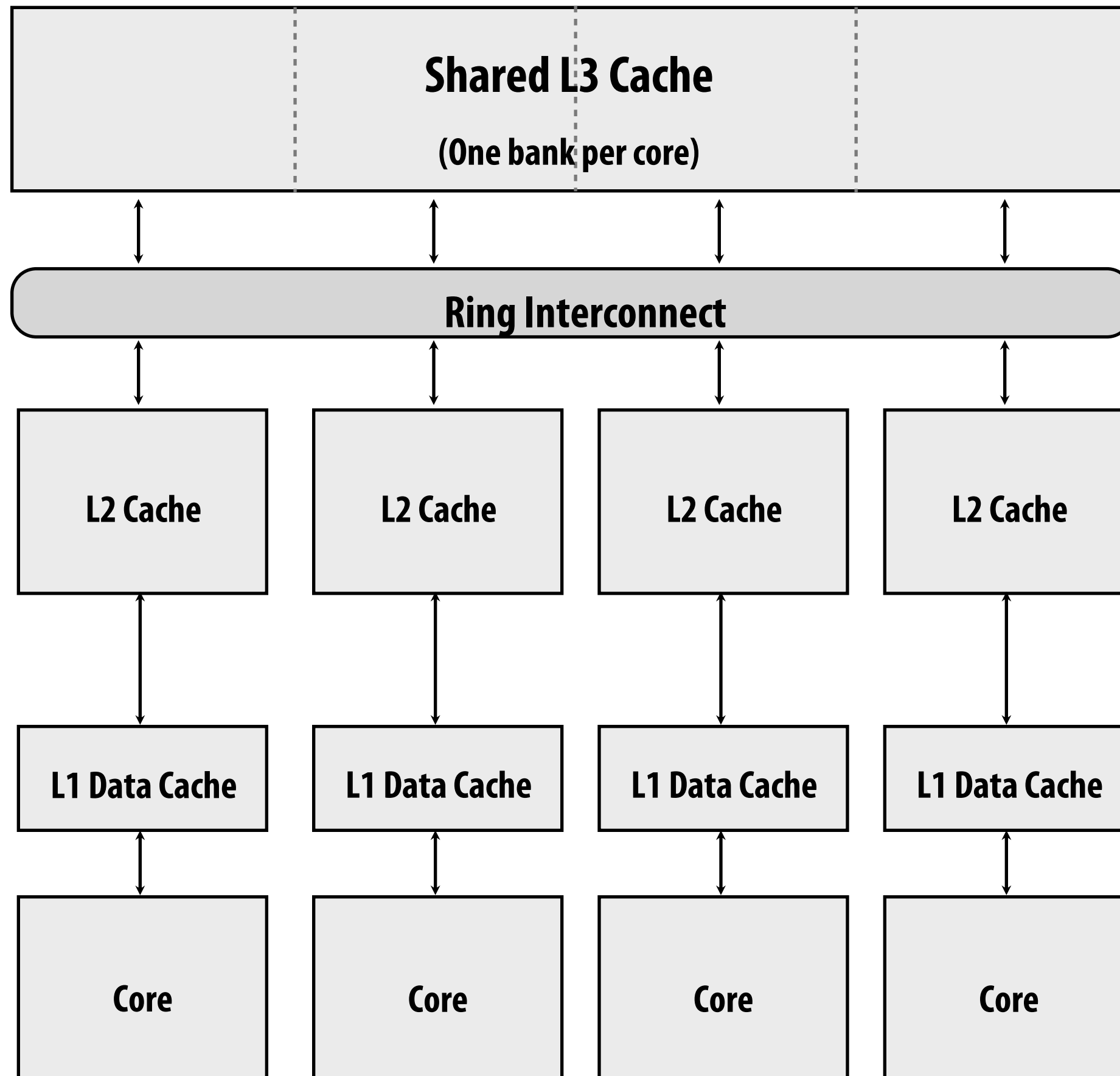
- Like MESI, but one cache holds shared line in F state rather than S (F="forward")
- Cache with line in F state services miss
- Simplifies decision of which cache should service miss (basic MESI: all caches respond)
- Used by Intel processors

## ■ MOESI (5-stage invalidation-based protocol)

- In MESI protocol, transition from M to S requires flush to memory
- Instead transition from M to O (O="owned, but not exclusive") and do not flush to memory
- Other processors maintain shared line in S state, one processor maintains line in O state
- Data in memory is stale, so cache with line in O state must service cache misses
- Used in AMD Opteron

# Reality: multi-level cache hierarchies

Recall Intel Core i7 hierarchy



- **Challenge: changes made to data at first level cache may not be visible to second level cache controller than snoops the interconnect.**
- **How might snooping work for a cache hierarchy?**
  1. **All caches snoop interconnect independently? (inefficient)**
  2. **Maintain “inclusion”**

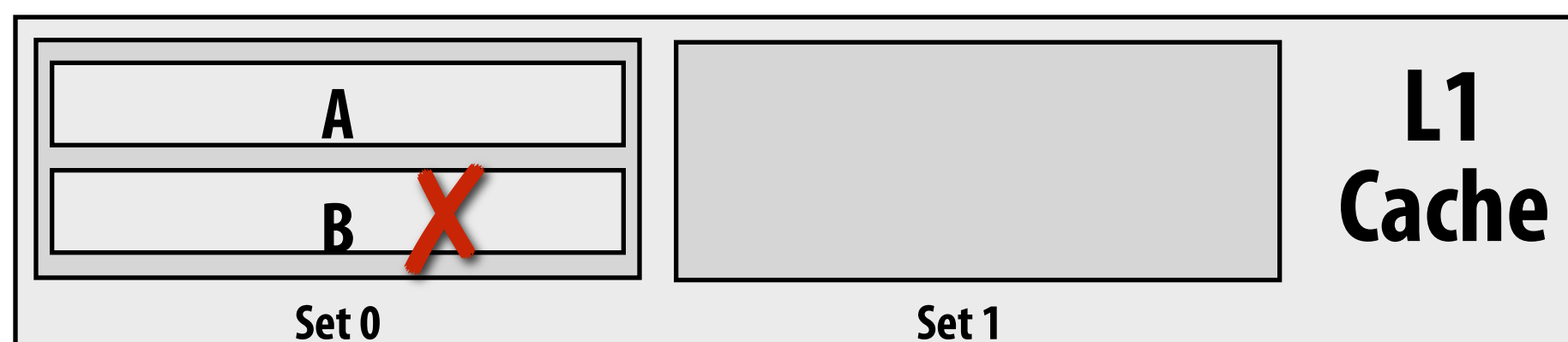
# Inclusion property of caches

- **All lines in closer [to processor] cache are also in farther [from processor] cache**
  - **e.g., contents of L1 are a subset of contents of L2**
  - **Thus, all transactions relevant to L1 are also relevant to L2, so it is sufficient for only the L2 to snoop the interconnect**
  
- **If line is in owned state (M in MSI/MESI) in L1, it must also be in owned state in L2**
  - **Allows L2 to determine if a bus transaction is requesting a modified cache line in L1 without requiring information from L1**



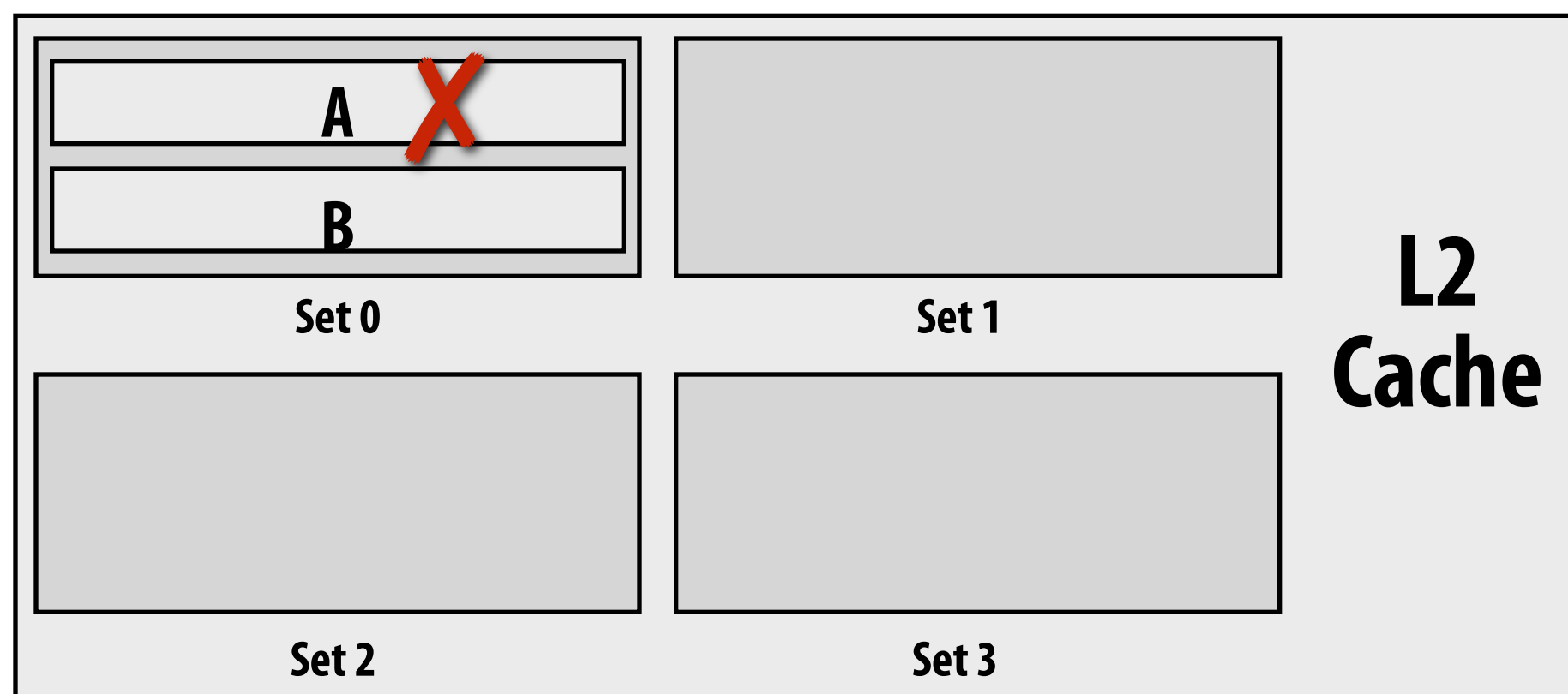
# Is inclusion maintained automatically if L2 is larger than L1? **No!**

- Consider this example:
  - Let L2 cache be twice as large as L1 cache
  - Let L1 and L2 have the same line size, are 2-way set associative, and use LRU replacement policy
  - Let A, B, C map to the same set of the L1 cache



Processor accesses A (L1+L2 miss)

Processor accesses B (L1+L2 miss).

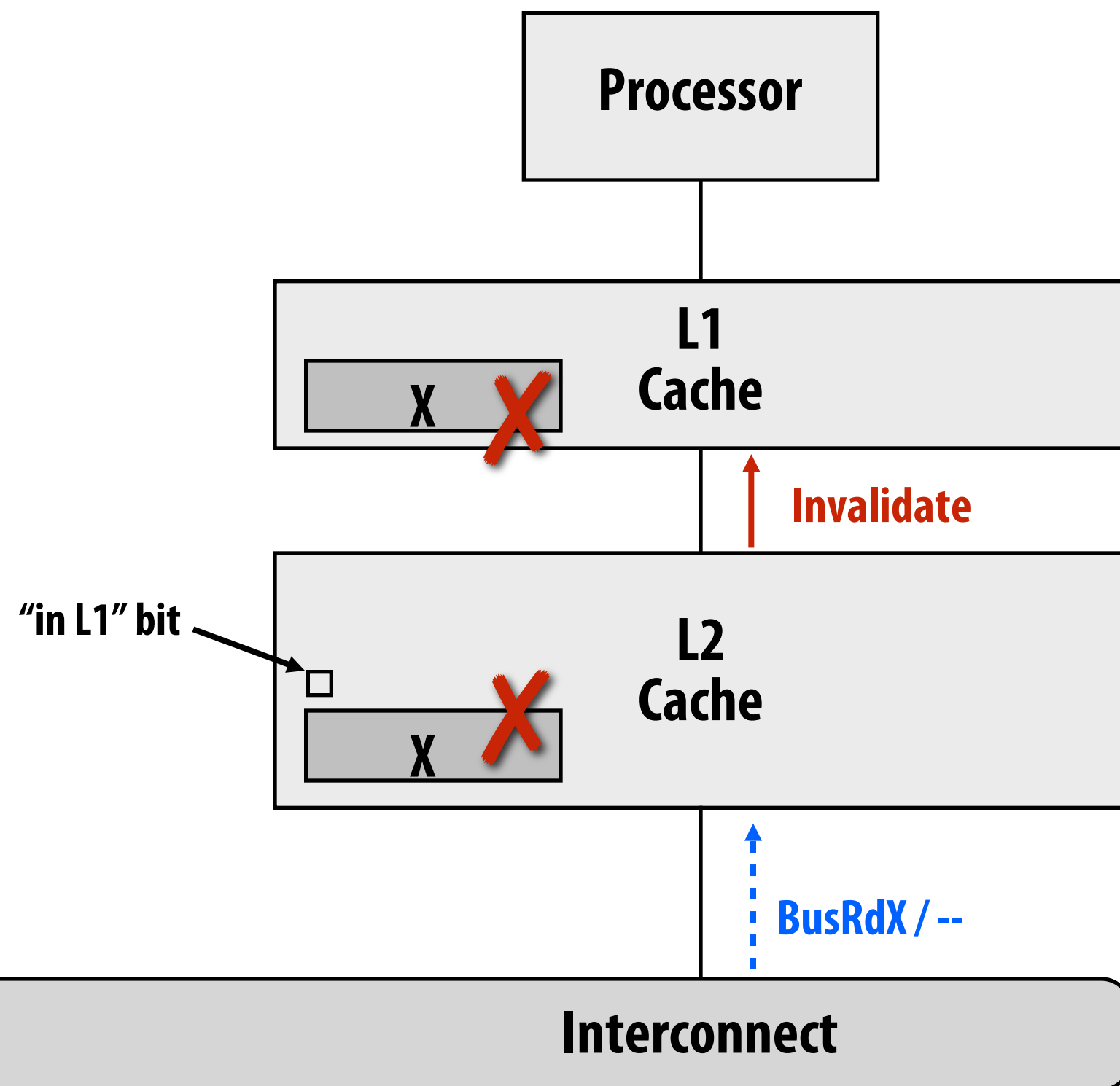


Processor accesses A many times (all L1 hits).

Processor now accesses C, triggering an L1 and L2 miss. L1 and L2 might choose to evict different lines, because the access histories differ.

As a result, inclusion no longer holds!

# Maintaining inclusion: handling invalidations



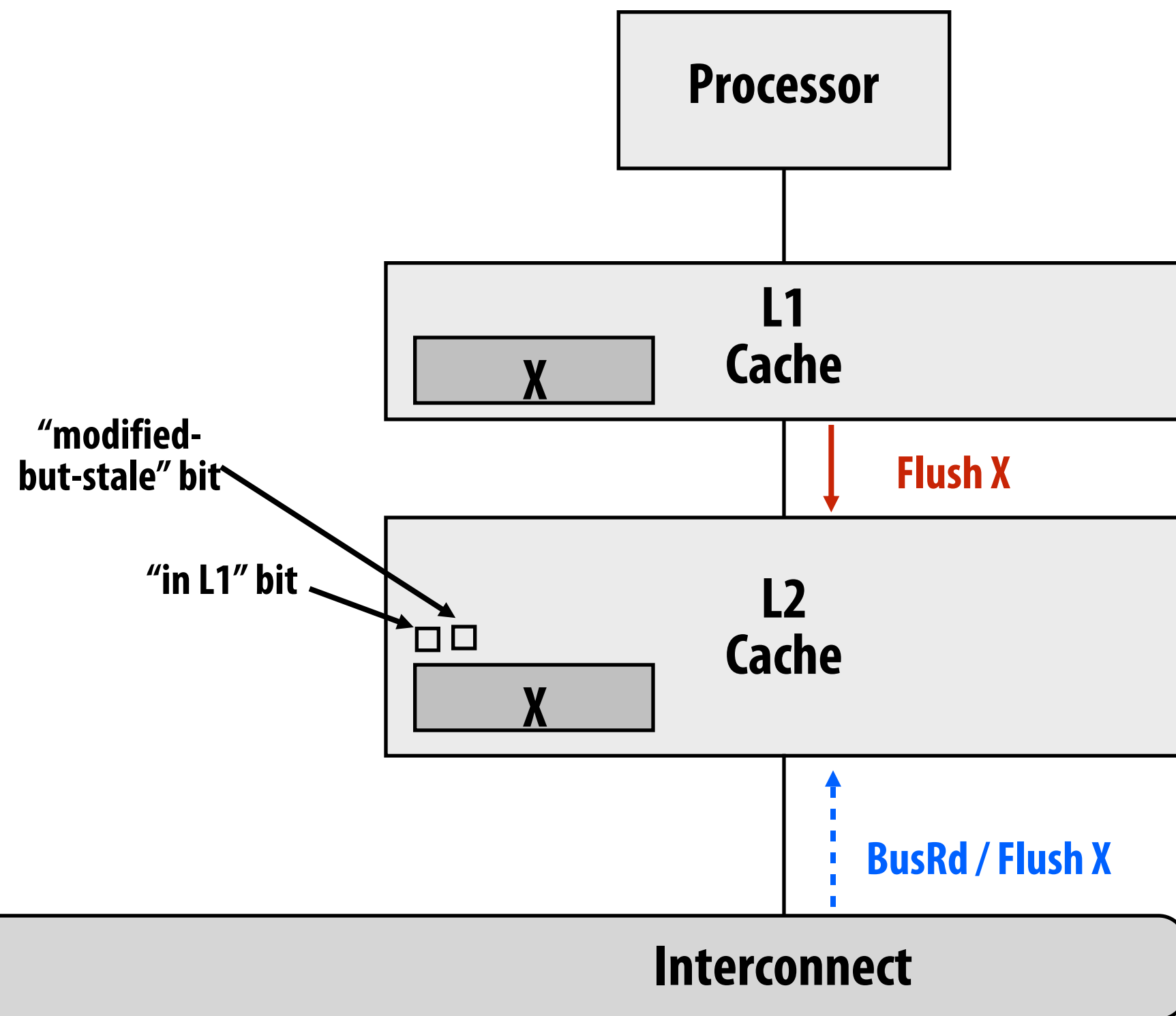
When line X is invalidated in L2 cache due to BusRdX from another cache.

Must also invalidate line X in L1

One solution: each L2 line contains an additional state bit indicating if line also exists in L1

This bit tells the L2 invalidations of the cache line due to coherence traffic need to be propagated to L1.

# Maintaining inclusion: L1 write hit



Assume L1 is a write-back cache. Processor writes to line X. (L1 write hit)

Line X in L2 cache is in modified state in the coherence protocol, but it has stale data!

When coherence protocol requires X to be flushed from L2 (e.g., another processor loads X), L2 cache must request the data from L1.

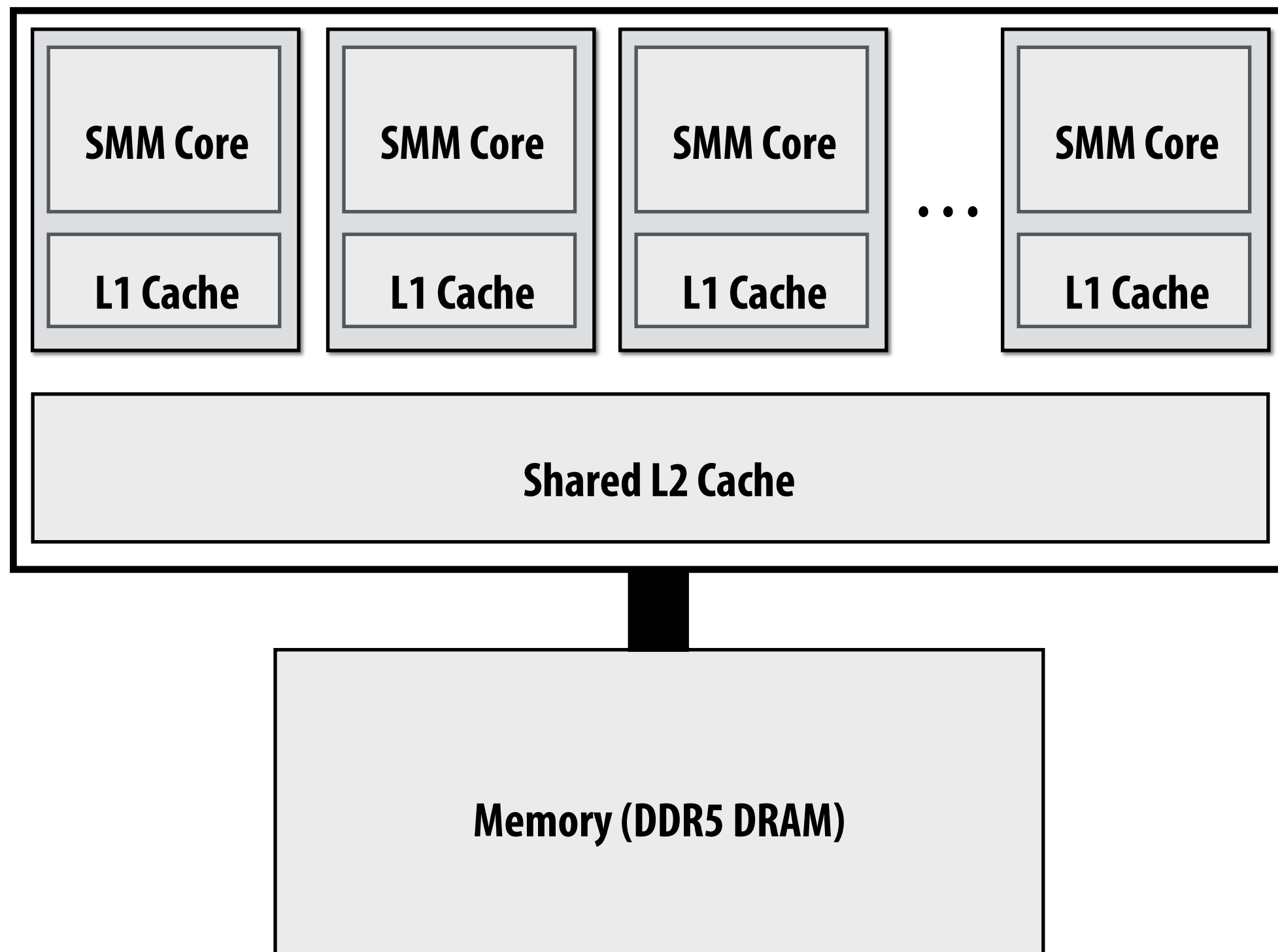
Add another bit for “modified-but-stale” (flushing a “modified-but-stale” L2 line requires getting the real data from L1 first.)

# HW implications of implementing coherence

- **Each cache must listen for and react to all coherence traffic broadcast on interconnect**
- **Additional traffic on interconnect**
  - **Can be significant when scaling to higher core counts**
- **Most modern multi-core CPUs implement cache coherence**
- **To date, discrete GPUs do not implement cache coherence**
  - **Thus far, overhead of coherence deemed not worth it for graphics and scientific computing applications (NVIDIA GPUs provide single shared L2 + atomic memory operations)**
  - **But the latest Intel Integrated GPUs do implement cache coherence**

# NVIDIA GPUs do not implement cache coherence

- Incoherent L1 caches (L1 per SMM)
- Single, unified L2 cache



CUDA global memory atomic operations “bypass” L1 cache, so an atomic operation will always observe up-to-date data

```
// this is a read-modify-write performed atomically on the  
// contents of a line in the L2 cache  
atomicAdd(&x, 1);
```

L1 caches are write-through to L2 by default

CUDA volatile qualifier will cause compiler to generate a LD instruction that will bypass the L1 cache. (see ld.cg instruction)

NVIDIA graphics driver will clear L1 caches between any two kernel launches (ensures stores from previous kernel are visible to next kernel. Imagine a case where driver did not clear the L1 between kernel launches...)

Kernel launch 1:

SMM core 0 reads x (so it resides in L1)

SMM core 1 writes x (updated data available in L2)

Kernel launch 2:

SMM core 0 reads x (**cache hit! processor observes stale data**)

If interested in more details, see “Cache Operators” section of NVIDIA PTX Manual (Section 8.7.6.1 of Parallel Thread Execution ISA Version 4.1)

# **Implications of cache coherence to the programmer**

# Artifactual communication via false sharing

## What is the potential performance problem with this code?

```
// allocate per-thread variable for local per-thread accumulation  
int myPerThreadCounter[NUM_THREADS];
```

## Why is this better?

```
// allocate per thread variable for local accumulation  
struct PerThreadState {  
    int myPerThreadCounter;  
    char padding[CACHE_LINE_SIZE - sizeof(int)];  
};  
PerThreadState myPerThreadCounter[NUM_THREADS];
```

# False sharing

```
void* worker(void* arg) {  
    volatile int* counter = (int*)arg;  
    for (int i=0; i<MANY_ITERATIONS; i++)  
        (*counter)++;  
    return NULL;  
}
```

threads update a per-thread counter many times

```
void test1(int num_threads) {  
    pthread_t threads[MAX_THREADS];  
    int counter[MAX_THREADS];  
    for (int i=0; i<num_threads; i++)  
        pthread_create(&threads[i], NULL,  
                      &worker, &counter[i]);  
    for (int i=0; i<num_threads; i++)  
        pthread_join(threads[i], NULL);  
}
```

Execution time with num\_threads=12  
on 12 core system: 5.1 sec

```
struct padded_t {  
    int counter;  
    char padding[CACHE_LINE_SIZE - sizeof(int)];  
};
```

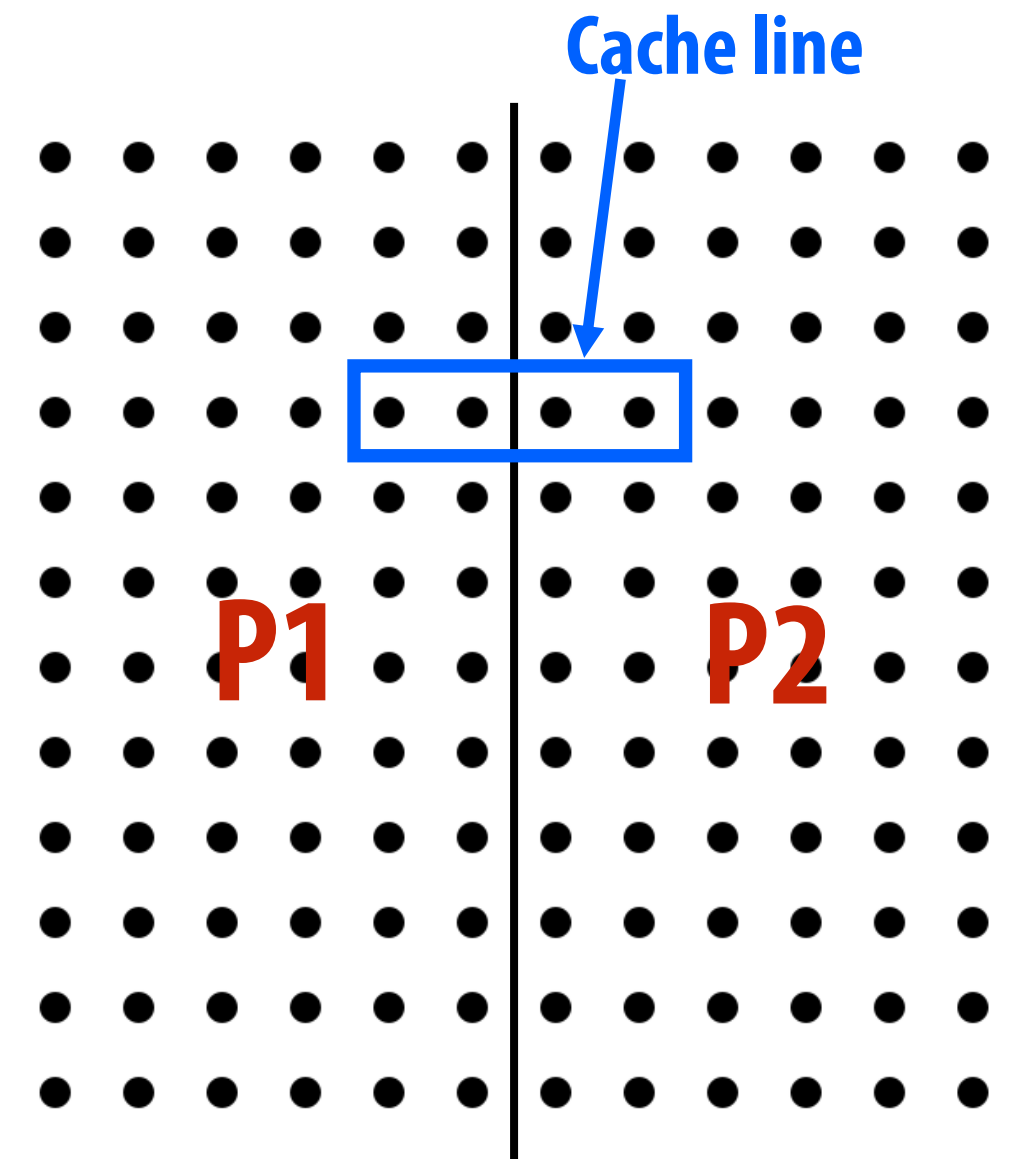
```
void test2(int num_threads) {  
    pthread_t threads[MAX_THREADS];  
    padded_t counter[MAX_THREADS];  
    for (int i=0; i<num_threads; i++)  
        pthread_create(&threads[i], NULL,  
                      &worker, &(counter[i].counter));  
    for (int i=0; i<num_threads; i++)  
        pthread_join(threads[i], NULL);  
}
```

Execution time with num\_threads=12  
on 12 core system: 2.1 sec



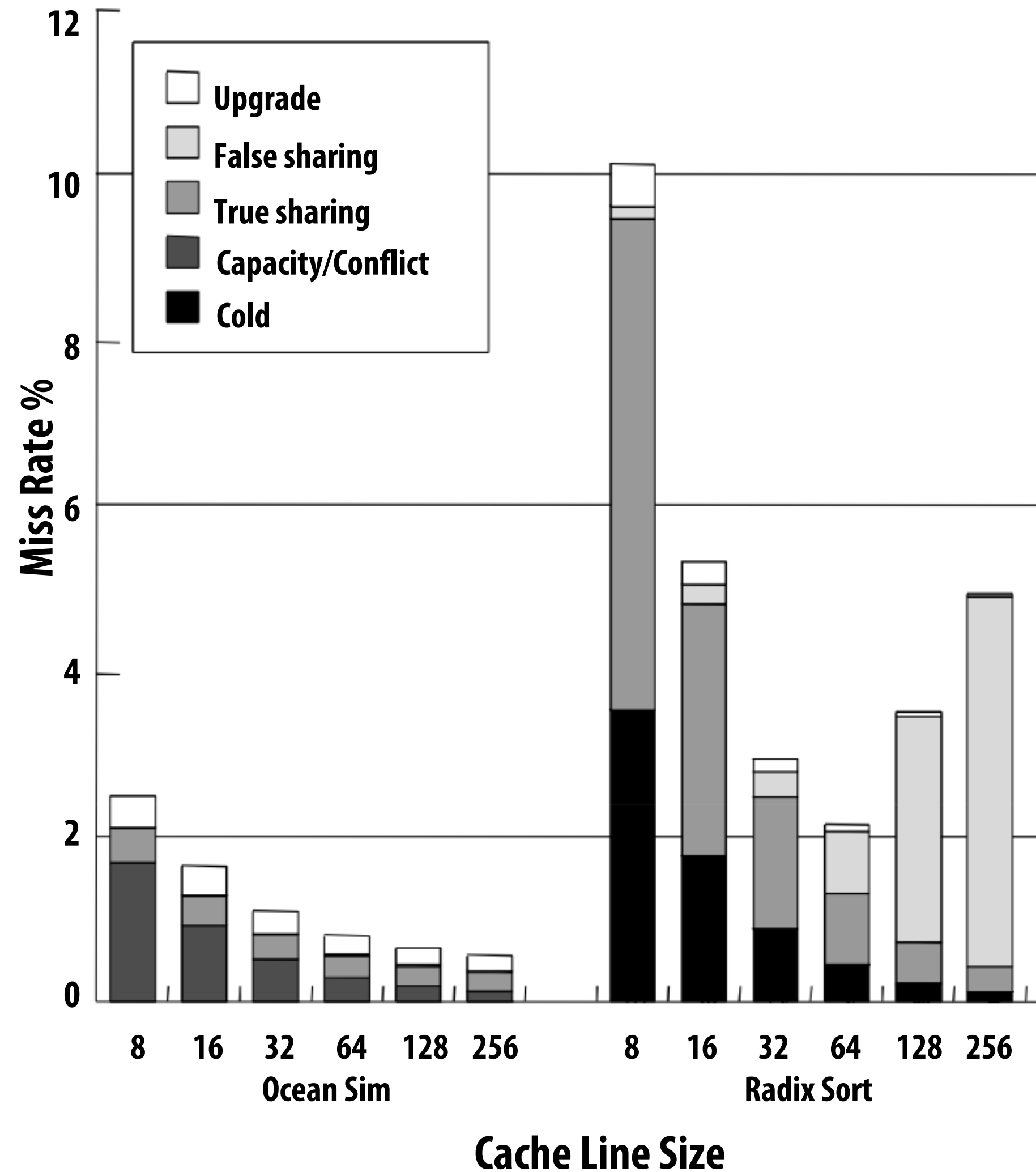
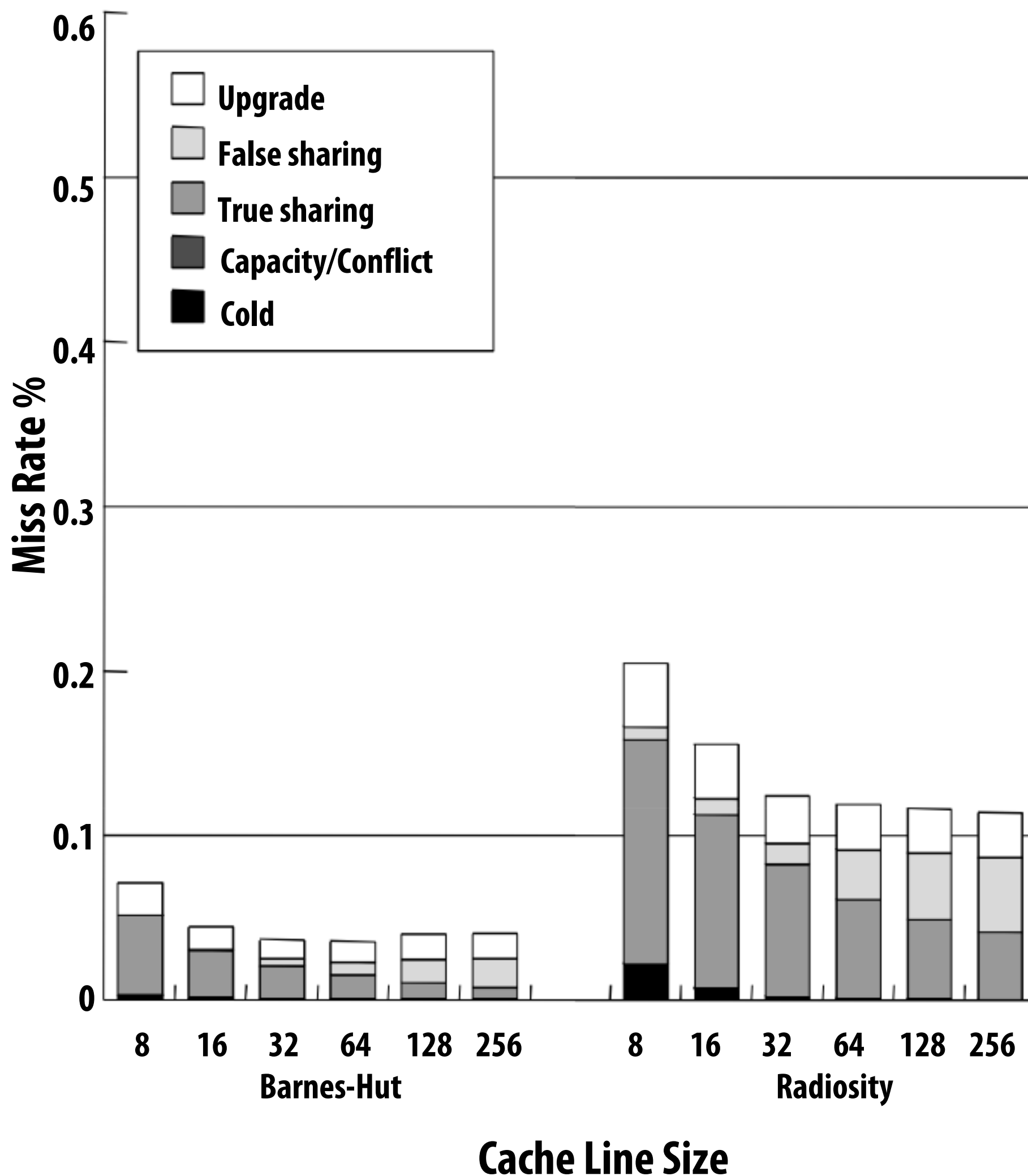
# False sharing

- **Condition where two processors write to different addresses, but addresses map to the same cache line**
- **Cache line “ping-pongs” between caches of writing processors, generating significant amounts of communication due to the coherence protocol**
- **No inherent communication, this is entirely artifactual communication**
- **False sharing can be a factor in when programming for cache-coherent architectures**



# Impact of cache line size on miss rate

Results from simulation of a 1 MB cache (four example applications)



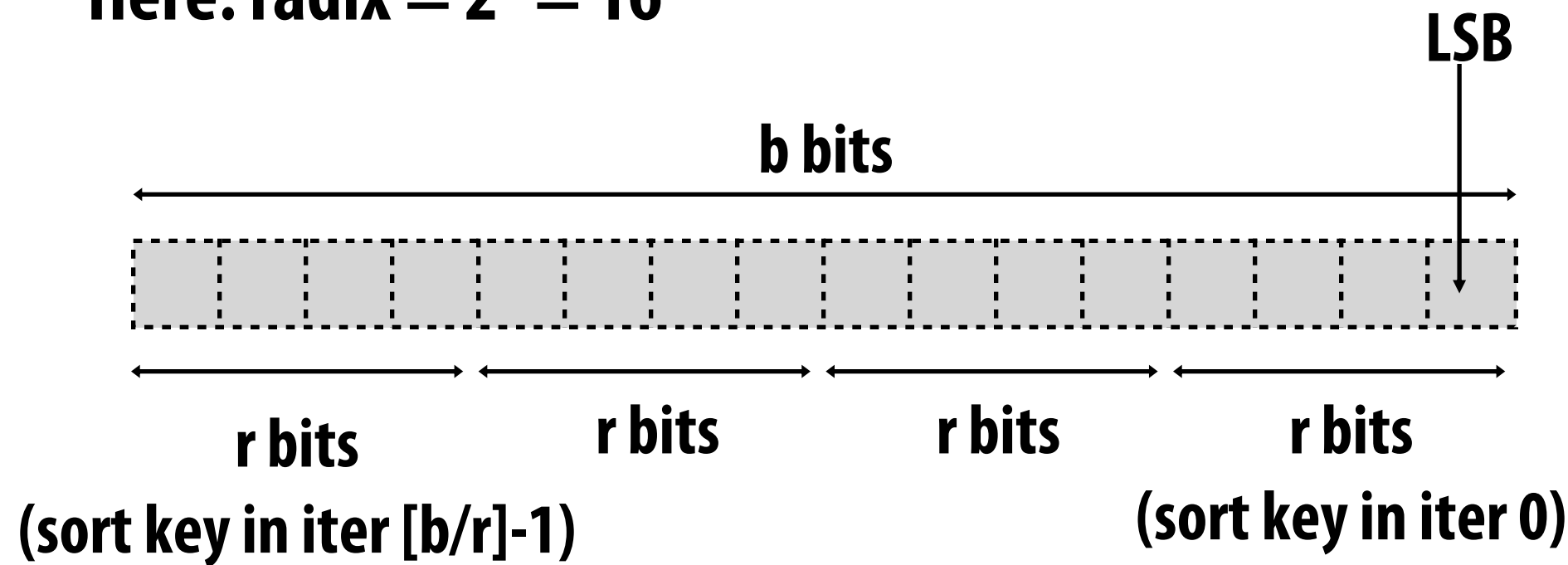
\* Note: I separated the results into two graphs because of different Y-axis scales

Figure credit: Culler, Singh, and Gupta

# Parallel radix sort of b-bit numbers

Sort array of  $N$ ,  $b$ -bit numbers

Here: radix =  $2^4 = 16$



For each group of  $r$  bits: (serial loop)

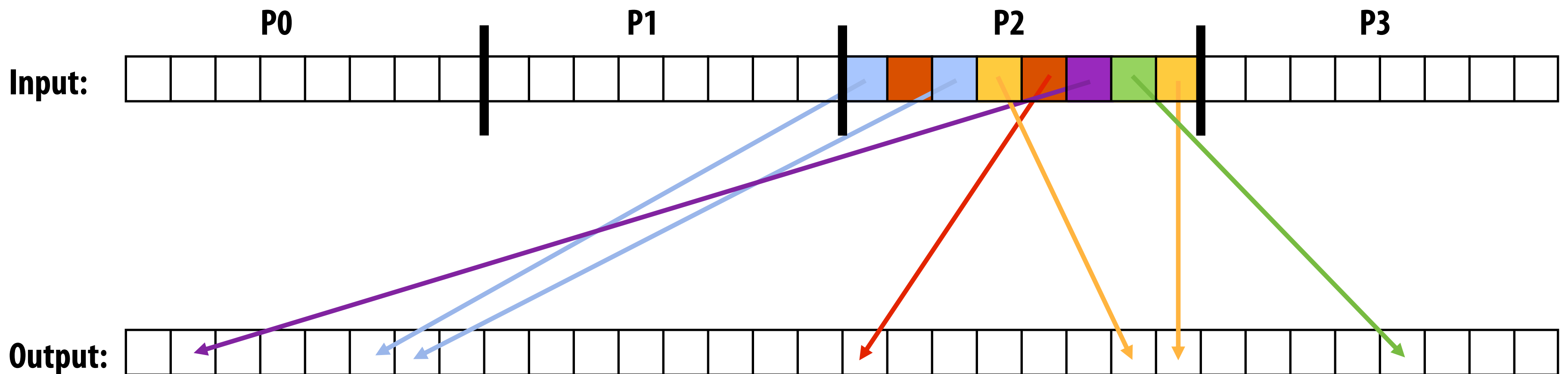
In parallel, on each processor:

Sort elements by  $r$ -bit value

Compute number of elements in each bin ( $2^r$  bins)

Aggregate per-processor counts to compute  
compute bin starts

Write elements to appropriate position



Potential for lots of false sharing

False sharing decreases with increasing array size

# Summary: snooping-based coherence

- The cache coherence problem exists because the abstraction of a single shared address space is not implemented by a single storage unit
  - Storage is distributed among main memory and local processor caches
  - Data is replicated in local caches for performance
- Main idea of snooping-based cache coherence: whenever a cache operation occurs that could affect coherence, the cache controller **broadcasts a notification to all other cache controllers**
  - Challenge for HW architects: minimizing overhead of coherence implementation
  - Challenge for SW developers: be wary of artifactual communication due to coherence protocol (e.g., false sharing)
- Scalability of snooping implementations is limited by ability to broadcast coherence messages to all caches!

# Quiz Time

# Bonus Material

- **For projects, not exams**

# Update-based coherence protocols

- **So far, we have talked about invalidation-based protocols**
  - **Main idea: to write to a line, cache must obtain exclusive access to it**  
**(the other caches invalidate their copy of the line)**
  - **Possible issues:**
    - **Cache must reload entire line after invalidation (consumes bandwidth)**
    - **False sharing**
- **Invalidation-based protocols are most commonly used designs today**
  - **You should know the difference between invalidation-based and update-based cache coherence... (but we don't require you to know the details of the Dragon update-based scheme shown on the next slide)**
  - **But let's talk about one update-based protocol for completeness**

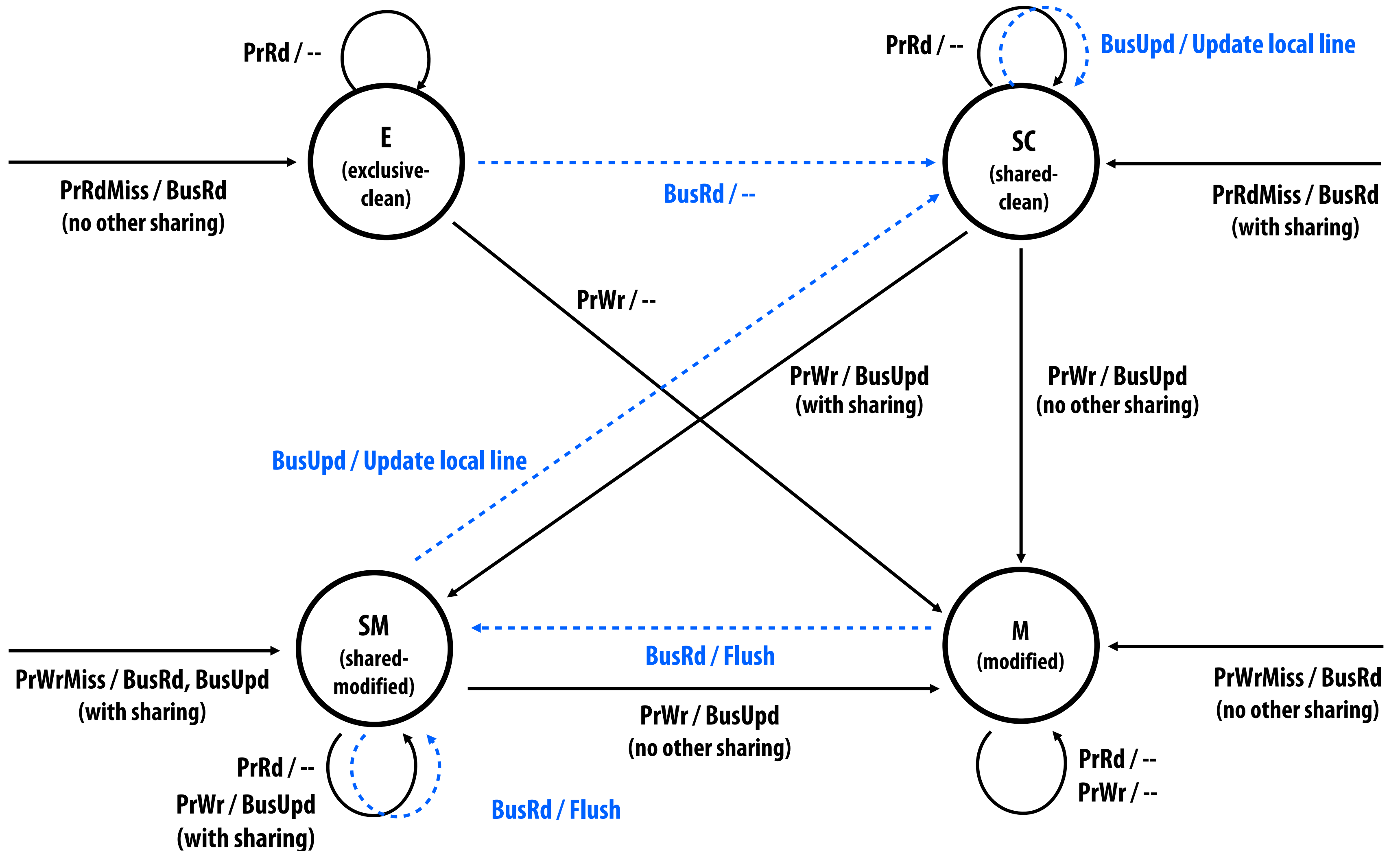
# Dragon write-back update protocol

- **States:** (no invalid state, but can think of lines as invalid before loaded for the first time)
  - Exclusive-clean (E): only one cache has line, memory up-to-date
  - Shared-clean (SC): multiple caches may have line, and memory may or may not \*\* be up to date
  - Shared-modified (SM): multiple caches may have line, memory not up to date
    - Only one cache can be in this state for a given line (but others can be in SC)
    - Cache with line in SM state is “owner” of data. Must update memory upon eviction
  - Modified (M): only one cache has line, it is dirty, memory is not up to date
    - Cache is owner of data. Must update memory upon replacement
- **Processor actions:**
  - PrRd, PrWr, PrRdMiss, PrWrMiss
- **Bus transactions:**
  - Bus read (BusRd), flush (provide line), bus update (BusUpd)

\*\* Why “may or may not”? Because memory IS up to date if all processors with line have it in SC state. But memory is not up to date if some other processor has line in SM state.



# Dragon write-back update protocol



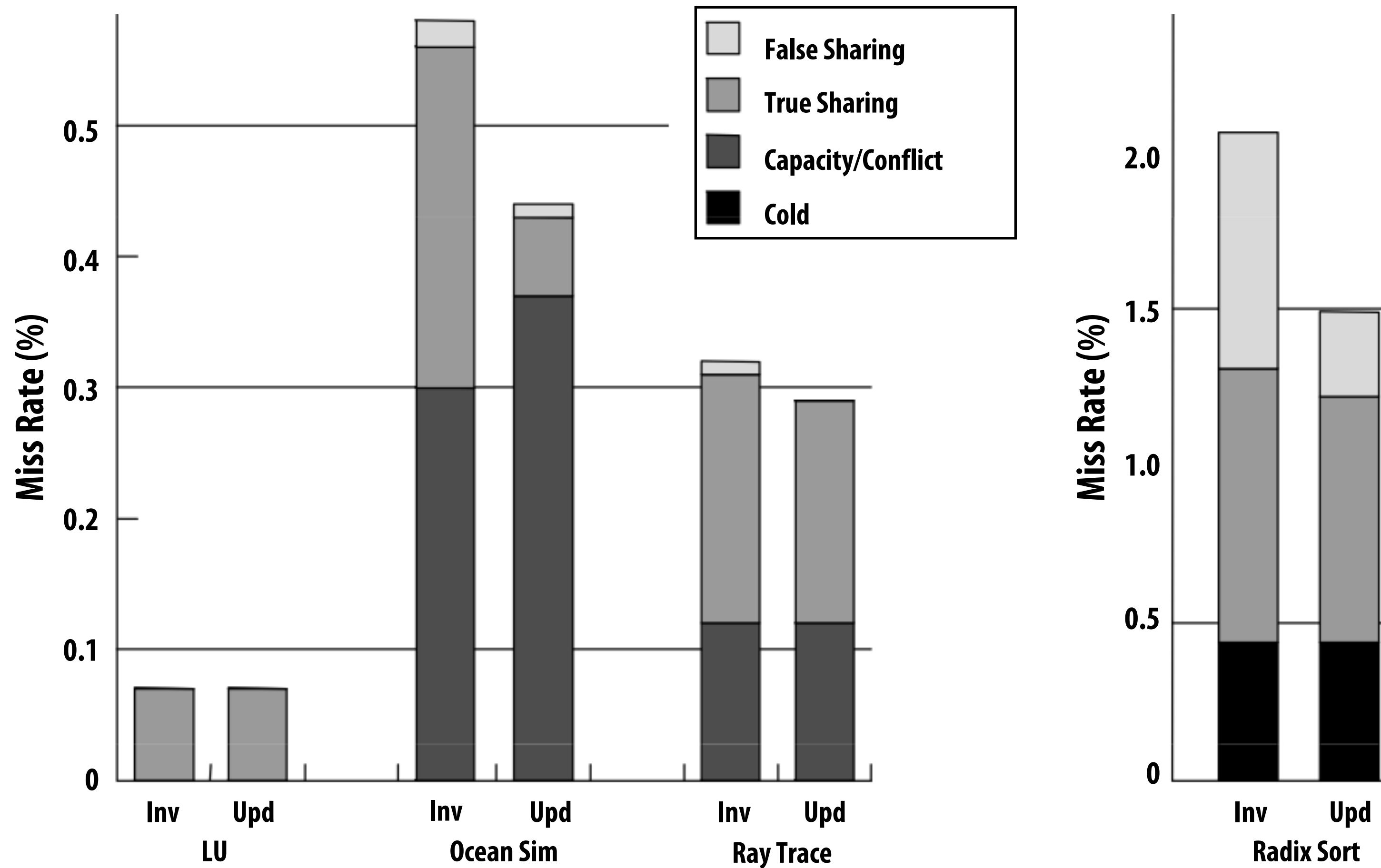
Not shown: upon line replacement, cache must flush line to memory if line is in SM or M state  
 (Note: there's no invalid state here: why?)

# Invalidate vs. update-based protocols

- **Which is better?**
- **Intuitively, update would seem preferable if other processors sharing data continue to access it after a write occurs**
- **But updates are overhead if:**
  - **Data just sits in caches (and is never read by another processor again)**
  - **Application performs many writes before the next read**

# Invalidate vs. update evaluation: miss rate

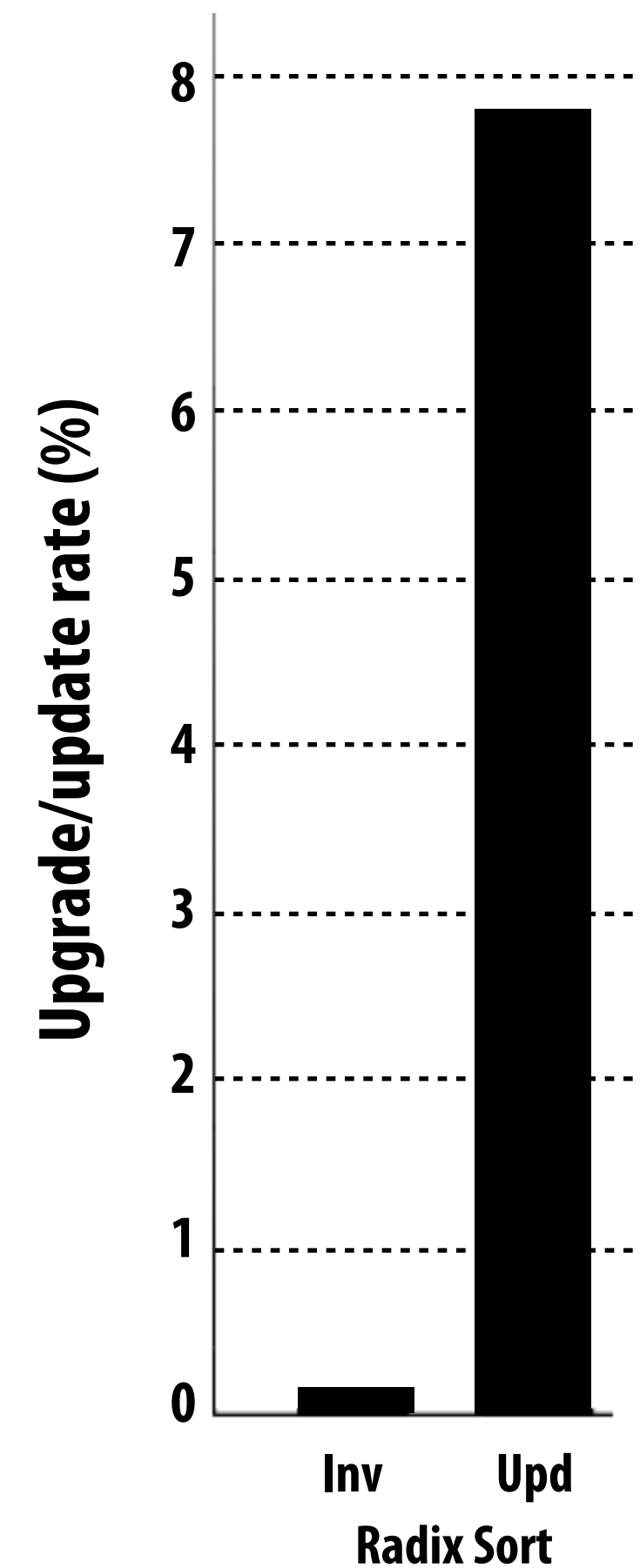
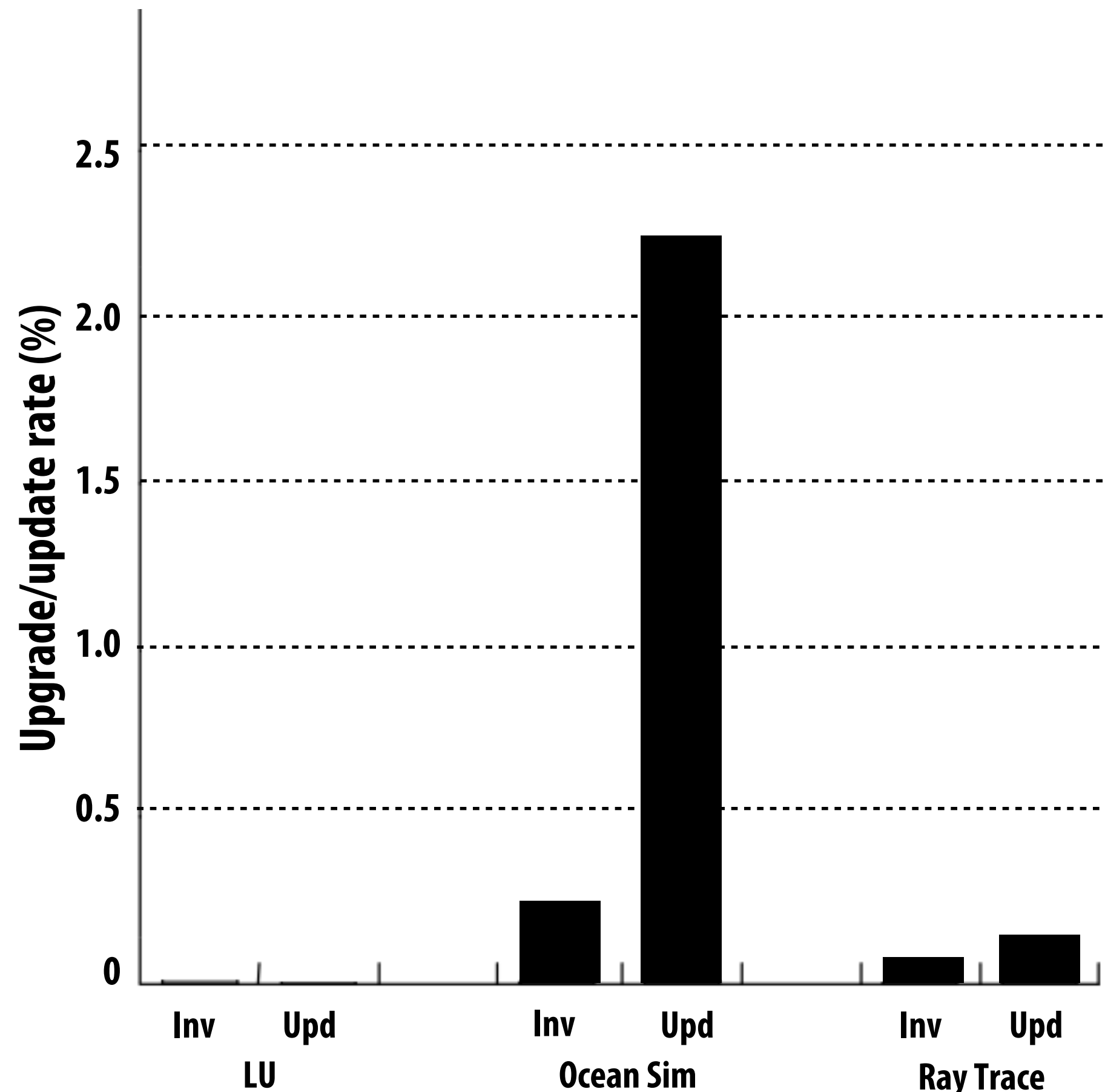
Simulated 1 MB cache, 64-byte lines



So... is update better?

# Invalidate vs. update evaluation: traffic

Simulated 1 MB cache, 64-byte lines



- Update can suffer from high traffic due to multiple writes before the next read by another processor
- Current AMD and Intel implementations of cache coherence are invalidation based

\*\* Charts compare frequency of upgrades in invalidation-based protocol to frequency of updates in update-based protocol

Figure credit: Culler, Singh, and Gupta