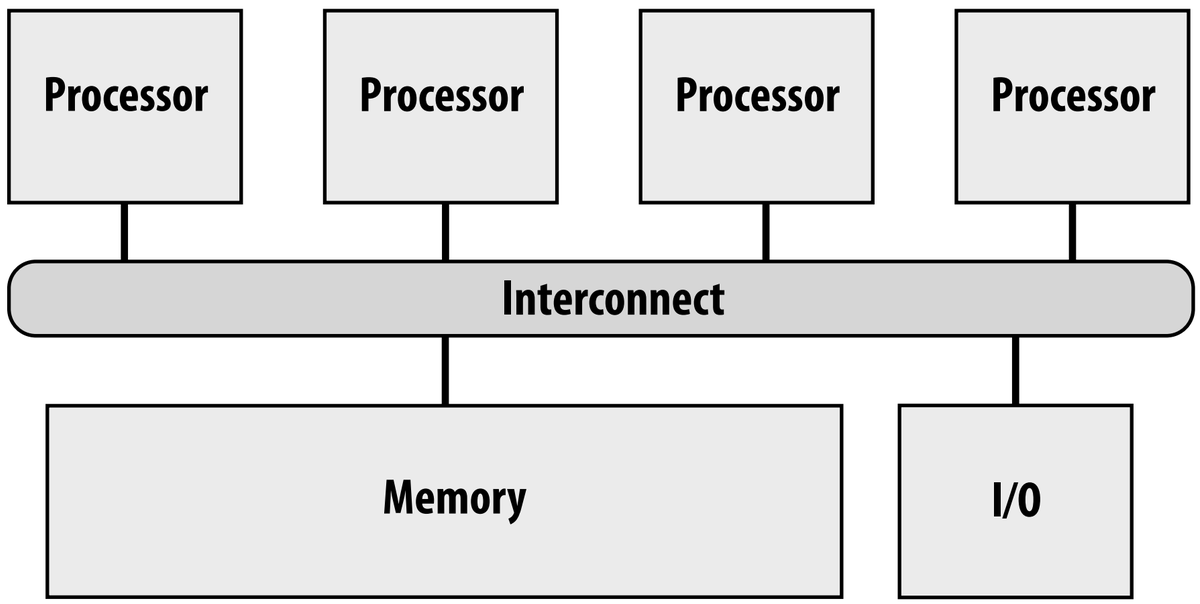
Cache Coherence

CMU 15-418: Parallel Computer Architecture and Programming (Spring 2012)

Shared memory multi-processor

- Processors read and write to shared variables
  - More precisely: processors issues load and store instructions
- Intuitively... reading value at address should return the last value written at the address *by any processor*



The cache coherence problem

Modern processors replicate contents of memory in local caches

Result of writes: processors can have different values for the same memory location

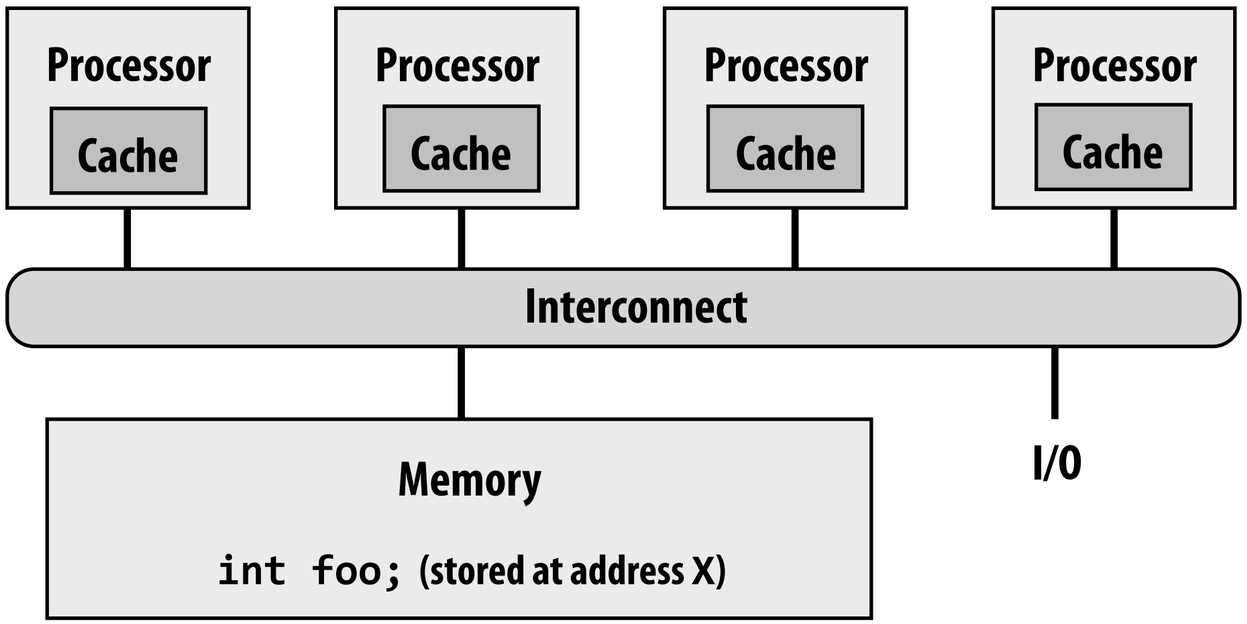


Chart shows value of `foo` (variable stored at address X) stored in main memory and in each processor's cache **

**** Assumes 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 (say this load causes eviction of foo)		0	2		1

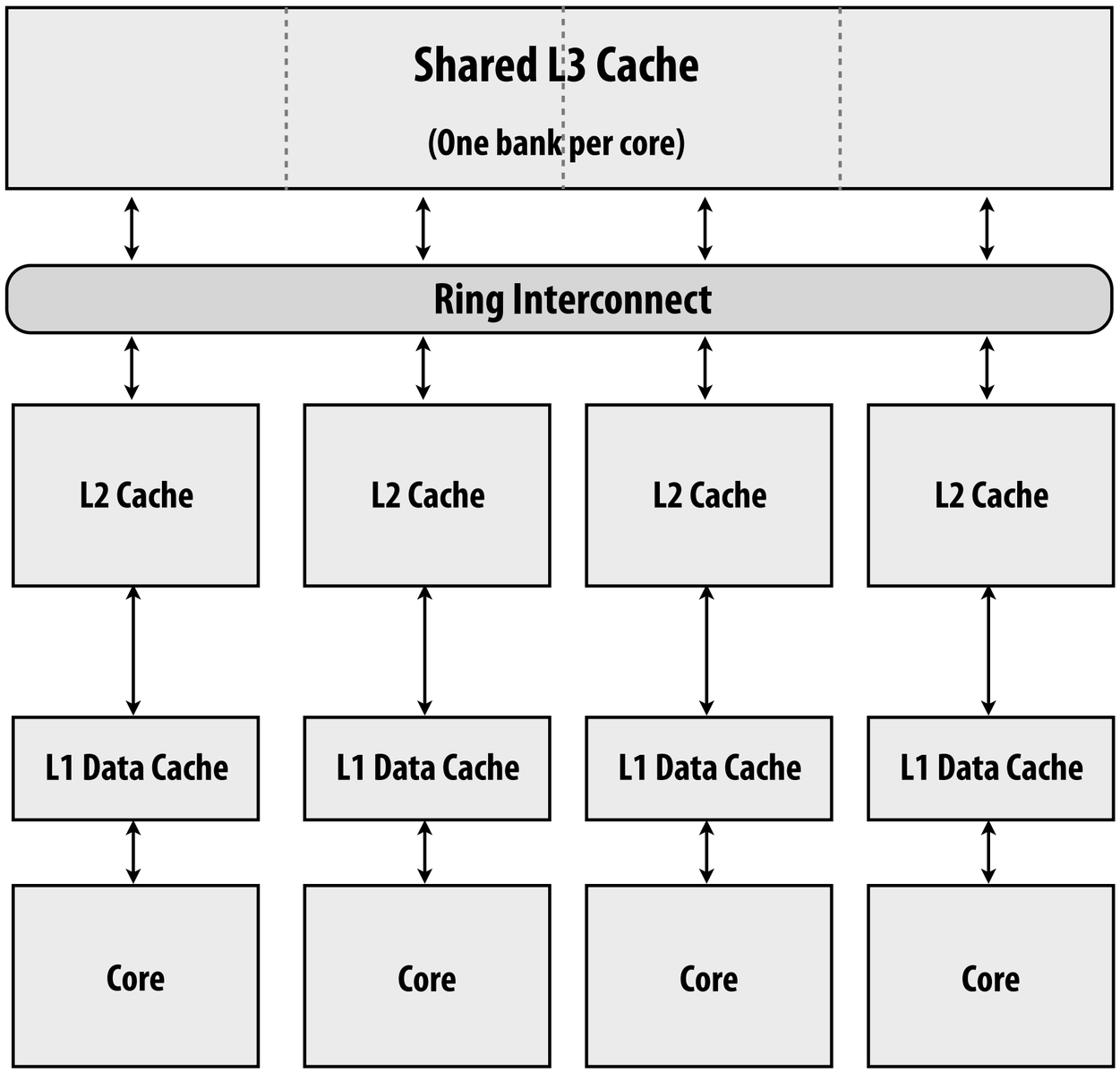
The cache coherence problem

- Reading value at address X should return the last value written at address X *by any processor.*

- Coherence problem exists because there is both global state (main memory) and local state (contents of private processor caches).

Cache hierarchy of Intel Core i7

64 byte cache line size



L3: (per chip)
8 MB, inclusive
16-way set associative
32B / clock per bank
26-31 cycle latency

L2: (private per core)
256 KB
8-way set associative, write back
32B / clock, 12 cycle latency
16 outstanding misses

L1: (private per core)
32 KB
8-way set associative, write back
2 x 16B loads + 1 x 16B store per clock
4-6 cycle latency
10 outstanding misses

Review: key terms

- cache line
- write back vs. write through policy
- inclusion

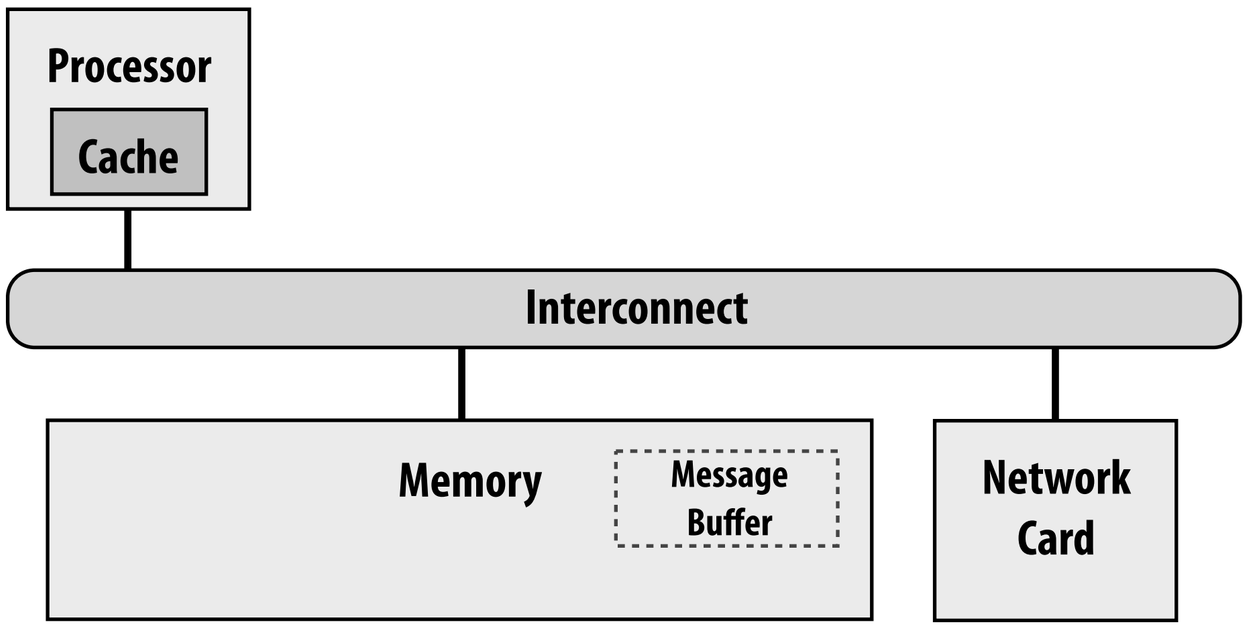
Intuitive expectation of shared memory

- Reading value at address should return the last value written at the address *by any processor.*

- Uniprocessor, providing this behavior is fairly simple, since writes typically come from one client: the processor
  - Exception: I/O via 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
Tells network card to async send buffer
Network card many transfer stale data

Case 2:
Network card receives message
DMA message contents into buffer in main memory
Notifies CPU msg received, buffer ready to read
CPU may read stale data

- **Common solutions:**
  - **CPU writes using uncached stores (e.g., driver code)**
  - **OS support:**
    - **mark pages containing shared buffers as uncached**
    - **OS flushes pages from cache when I/O completes**
- **In practice DMA transfers are infrequent compared to CPU loads/store (slower solutions are acceptable)**

Problems with the intuition

- Reading value at address should return the last value written at the address *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, it's impossible to communicate occurrence to other processors?

- In a sequential program, "last" is determined by program order (not time)
  - Holds true within a thread of a parallel program
  - But need to come up with a meaningful way to describe orders across threads

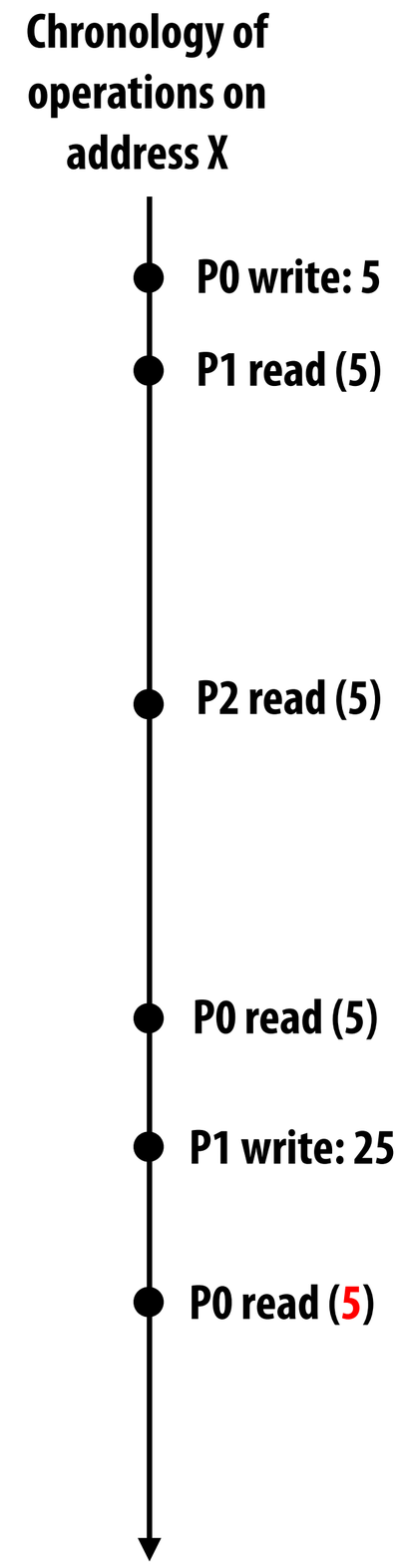
Definition: coherence

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 to the location that is consistent with the results of execution, and:

1. **Memory operations issued by any one process occur in the order issued by the process**

2. **The value returned by a read is the value written by the last write to the location in 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 a processor to address X that follows a write by another processor 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 location are serialized; two writes to the same location by any two processors are seen in the same order by all processors.**
   (Example: if values 1 and then 2 are written to address X, no processor observes 2 before 1)

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

Condition 2: write propagation: The news of the write has to eventually get to the other processors. Note that precisely when it is propagated is not defined by definition of coherence.

Condition 3: write serialization

Write serialization

Writes to the same location are serialized; two writes to the same location by any two processors are seen in the same order by all processors.

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

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

Consider situation where processors observe different order of writes:

Order observed by P1	Order observed by P2
x ← a	x ← b
⋮	⋮
x ← b	x ← a

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

Coherence vs. consistency

- Coherence defines behavior of reads and writes to the same memory location

- "Memory consistency" defines the behavior of reads and writes with respect to accesses to other locations (topic of a future lecture)
  - Consistency deals with the WHEN of write propagation

- For the purposes of this lecture:
  - If processor writes to address X and then writes to address Y. Then any processor that sees result of write to Y, also observes result of write to X.

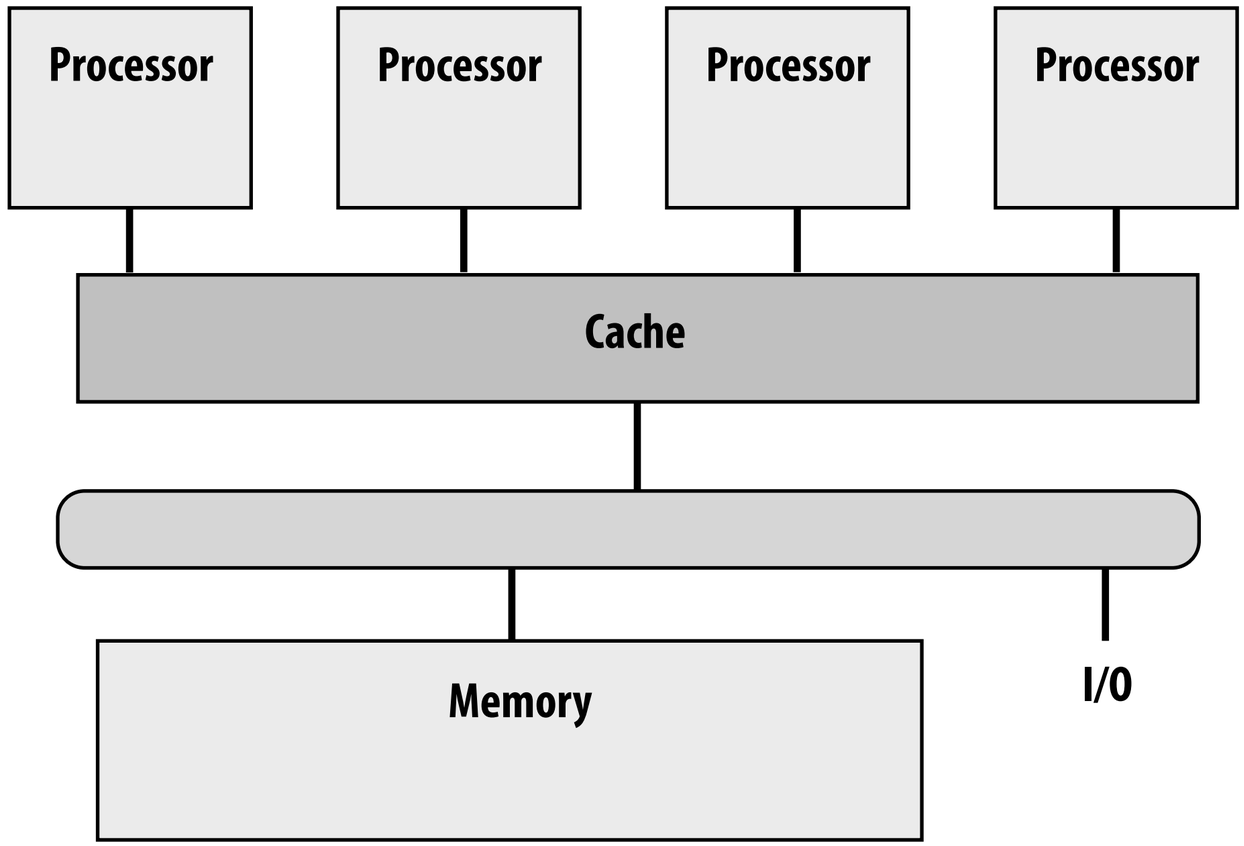
Implementing coherence

- **Software-based solutions**
  - OS uses page fault mechanism to propagate writes
  - Implementations provide memory coherence over clusters of workstations
  - We won't discuss these solutions

- **Hardware-based solutions**
  - "Snooping" based
  - Directory based

Shared caches: coherence made easy

- Obvious scalability problems
  - Interference / contention
- But can have benefits:
  - Fine-grained sharing (overlapping working sets)
  - Actions by one processor might pre-fetch for another



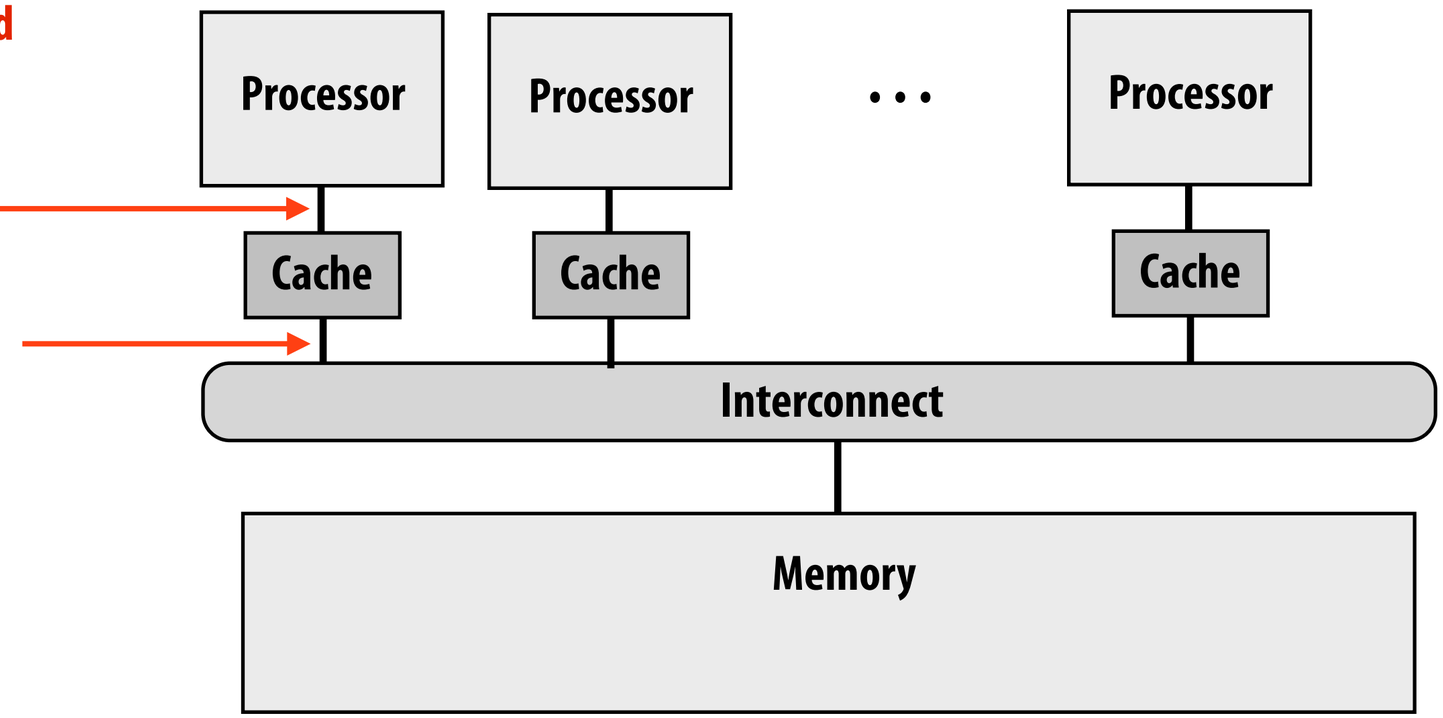
Snooping cache-coherence schemes

- All coherence-related activity is broadcast to all processors (actually, cache controllers) in the system

- Cache controllers monitor ("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 processor
2. Coherence-related activity broadcast over-interconnect



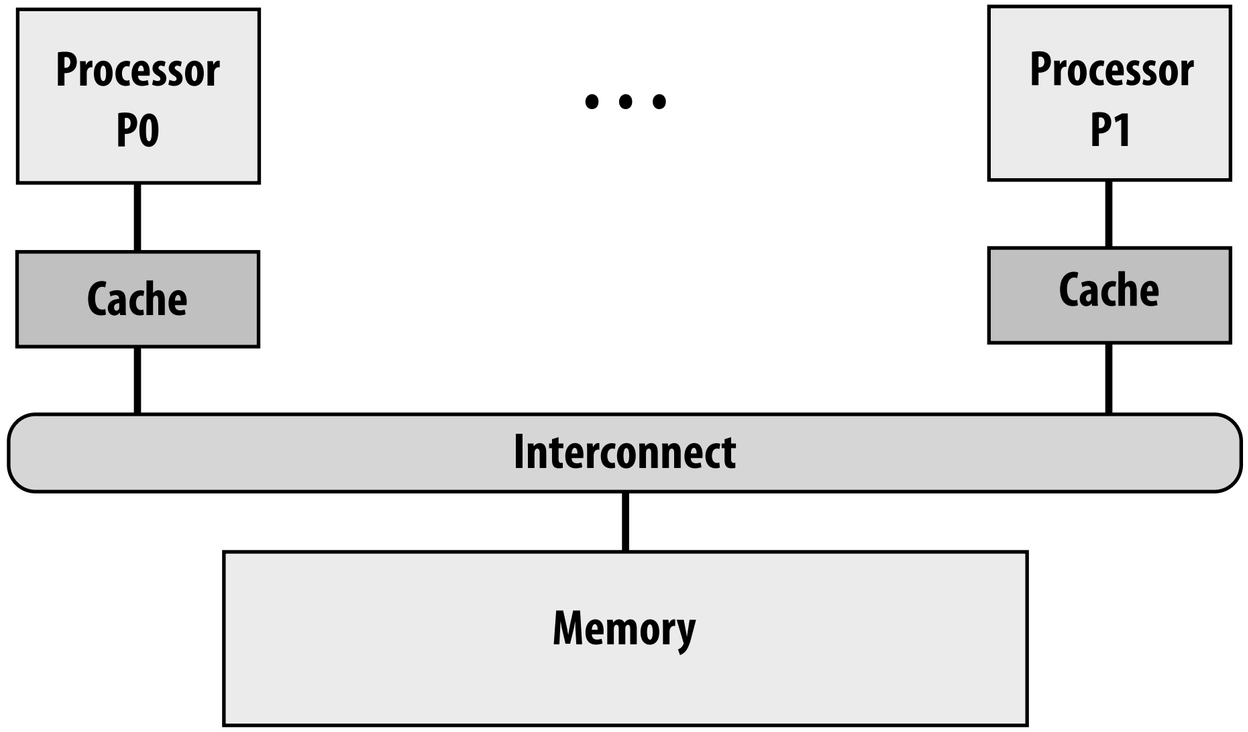
Very simple coherence implementation

Write-through caches

Granularity of coherence is cache block

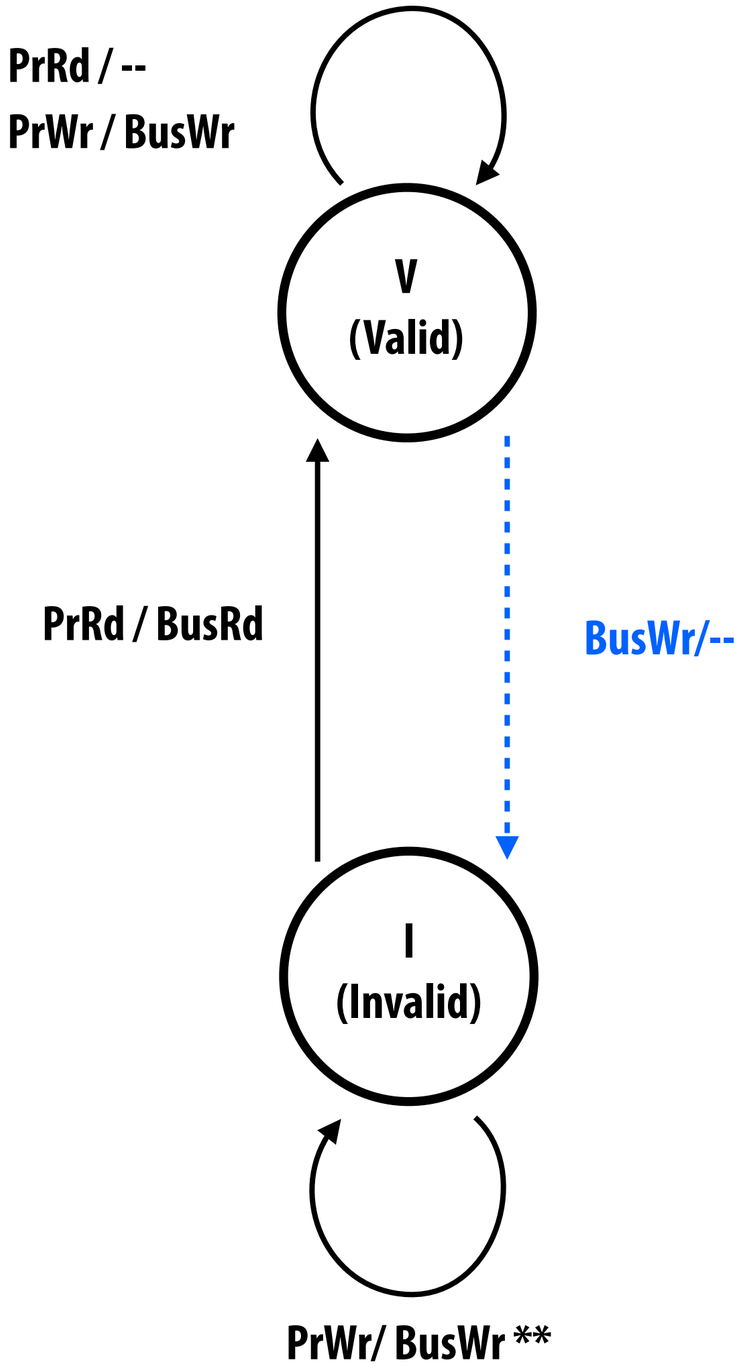
Upon write, broadcast invalidation

Next read from other processors will trigger cache miss
(retrieve updated value due to write-through policy)



Action	Bus 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	invalidation for X	100		100
P1 load X	cache miss for X	100	100	100

Write-through invalidation: state diagram



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

Broadcast (bus) initiated transaction

Processor initiated transaction

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. Process waits until previous memory operations is complete before issuing next memory operation
3. Invalidation applied immediately as part of receiving invalidation broadcast

** Write no-allocate policy (for simplicity)

Write-through policy is inefficient

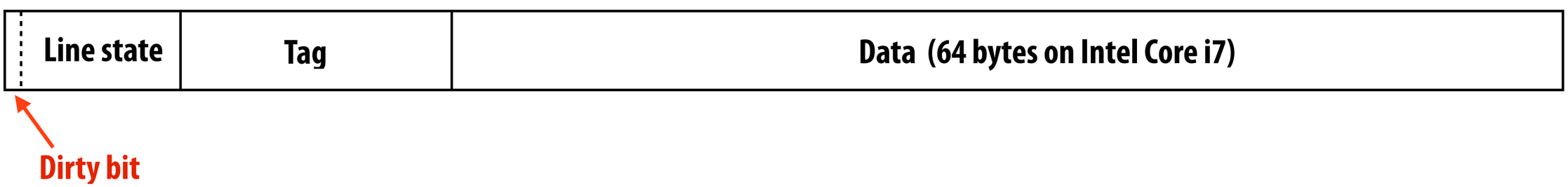
- Every write operation goes out to memory
  - Very high bandwidth requirements

- Write-back caches absorb most write traffic as cache hits
  - Significantly reduces bandwidth requirements
  - But now how do we ensure write propagation/serialization?
  - Require more sophisticated coherence protocols

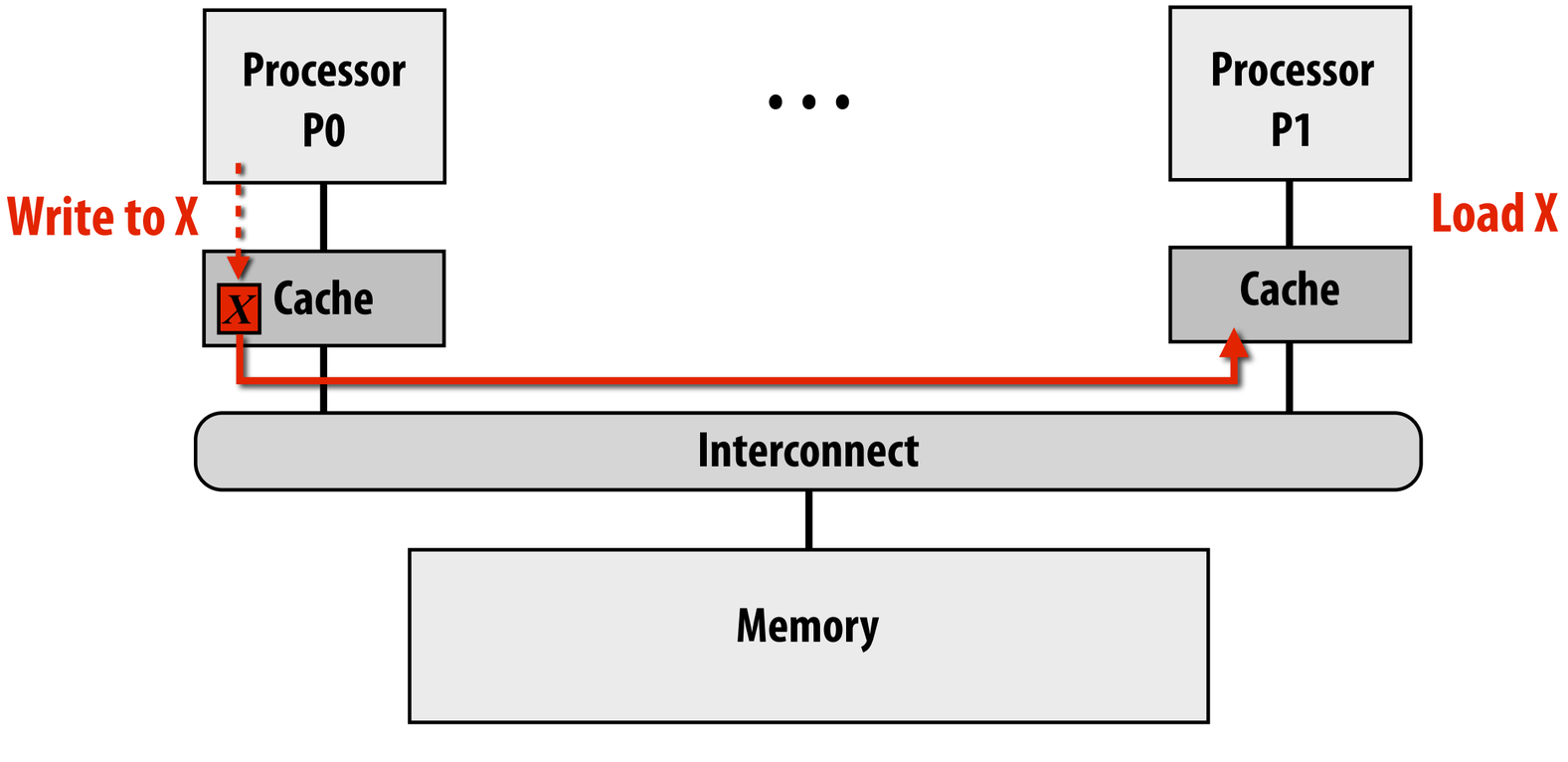
Review: write miss behavior of write-back cache (uniprocessor case)

Example: code executes `int x = 1;`

1. Processor performs write to address in line that is not resident in cache
2. Cache loads line from memory
3. One word in cache is updated
4. Cache line is marked as dirty



Cache coherence with write-back caches



- Dirty state of cache line now indicates exclusive ownership
  - Exclusive: only cache with a valid copy
  - Owner: responsible for supplying data upon request

Invalidation-based write-back protocol

- A line in the "exclusive" state can be modified without notifying other caches
  - Other caches don't have the line resident, so other processors cannot read these values [without generating a memory read transaction]

- Can only write to lines in the exclusive state
  - If processor performs a write to line that is not exclusive in cache, cache controller first broadcasts a read-exclusive transaction
  - Read-exclusive tells other caches about impending write ("you can't read anymore, because I'm going to write")
  - Read-exclusive transaction is required even if line is valid in processor's local cache
  - Dirty state implies exclusive

- When cache controller snoops a read exclusive for a line it contains
  - Must invalidate the line in its cache

Basic MSI write-back invalidation protocol

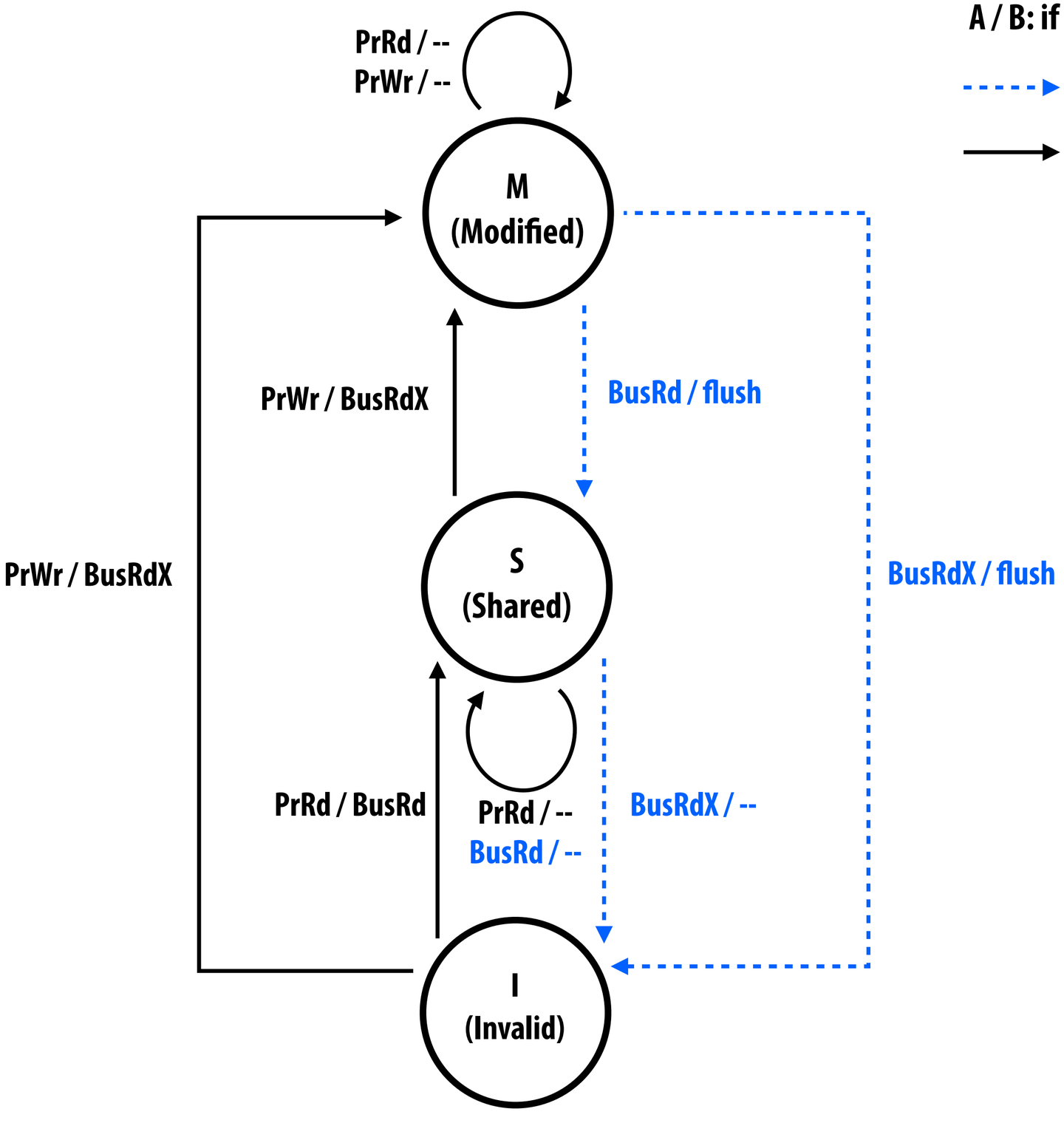
- Key tasks of protocol
  - Obtaining exclusive access for a write
  - Locating most recent copy of data on cache miss

- Cache line states
  - Invalid (I)
  - Shared (S): line valid in one or more caches
  - Modified (M): line valid in exactly one cache (a.k.a. "dirty" or "exclusive" state)

- Processor events
  - PrRd (read)
  - PrWr (write)

- Bus transactions
  - BusRd: obtain copy of line with no intent to modify
  - BusRdX: obtain copy of line with intent to modify
  - BusWB: write line out to memory

MSI state transition diagram



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

Broadcast (bus) initiated transaction

Processor initiated transaction

Alternative state names:

- E (exclusive, read/write access)
- S (potentially shared, read-only access)
- I (invalid, no access)

Does MSI satisfy coherence?

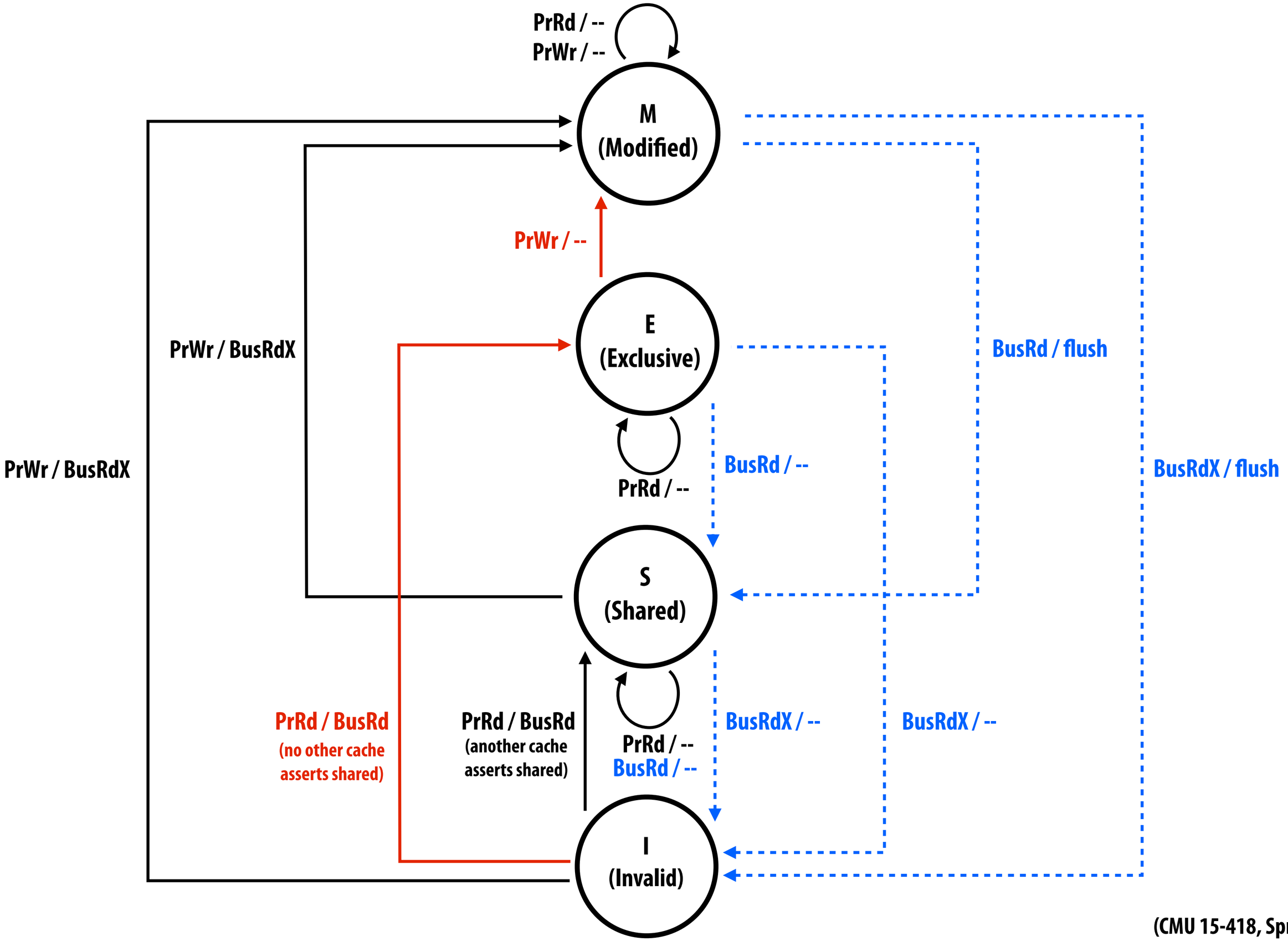
- **Write propagation**
  - Via invalidation

- **Write serialization**
  - Writes that appear on bus are ordered by the order they appear on bus (BusRdX)
  - Reads that appear on bus are ordered by order they appear on bus (BusRd)
  - Writes that don't appear on the bus (PrWr to line already in M state):
    - Sequence of writes to line comes between two bus 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 bus transaction for the line. So all the writes come before the transaction.
    - So all processors see writes in the same order.

MESI invalidation protocol

- MSI requires two bus transactions for the common case of reading data, 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 not modified, but only this cache has copy
  - 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 a bus transaction

MESI state transition diagram



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 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 today to reduce both latency of access and memory bandwidth requires

Increasing efficiency (and complexity)

- **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 <u>do not</u> 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

- **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

Implications of implementing coherence

- Each cache must listen and react to all coherent traffic broadcast on interconnect
  - Duplicate cache tags so that tag lookup in response to coherence actions does not interfere with processor loads and stores

- Additional traffic on interconnect
  - Can be significant when scaling to higher core counts

- To date, GPUs do not implement cache coherence
  - Thus far, overhead of coherence deemed not worth it for graphics applications

Implications to software developer

What could go wrong with this code?

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

Better:

```
// allocate per thread variable for local accumulation
struct PerThreadState {
  int myCounter;
  char padding[64 - sizeof(int)];
};
PerThreadState myCounter[NUM_THREADS];
```

False sharing

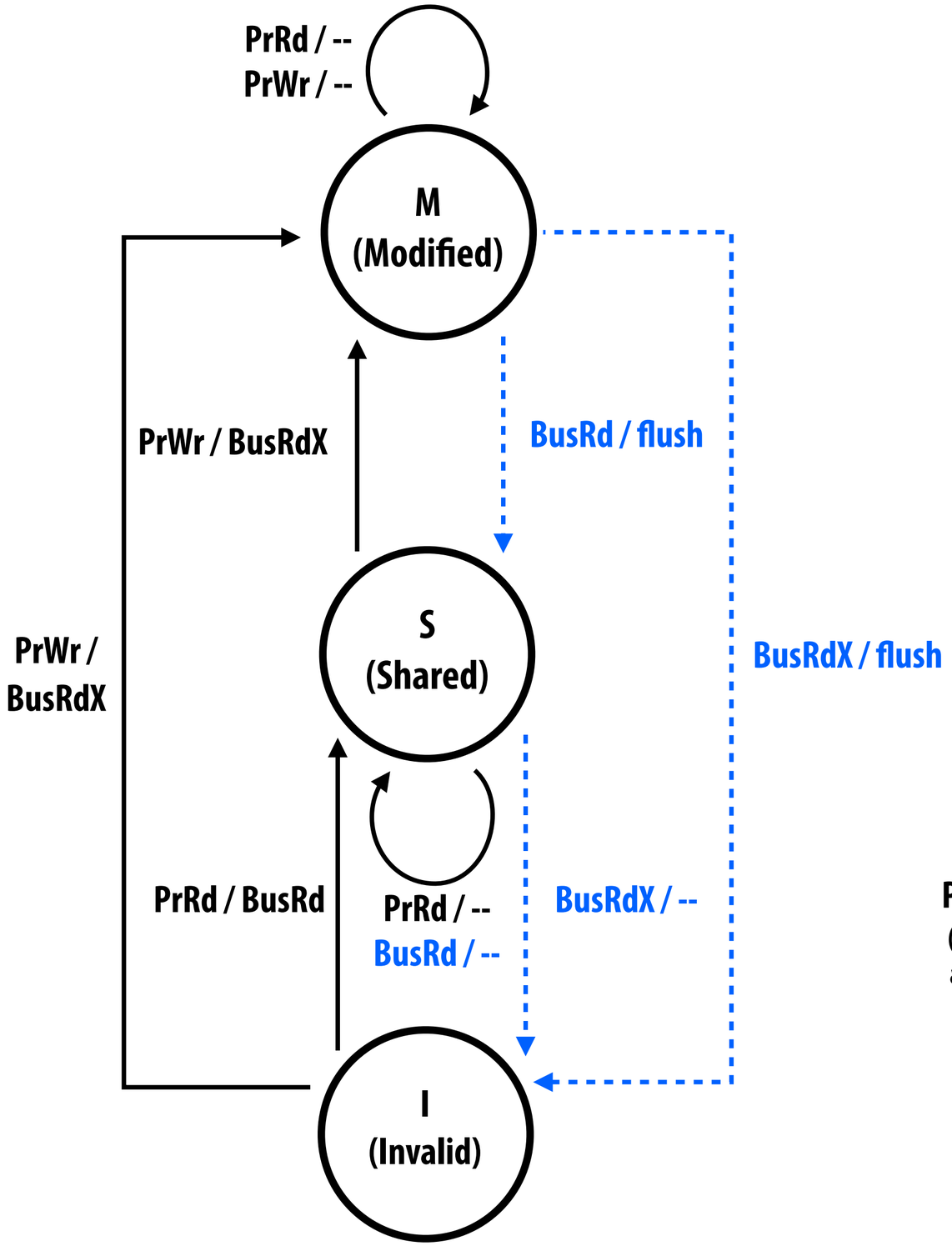
- Condition where two threads write to different variables, but variable's addresses map to same cache line

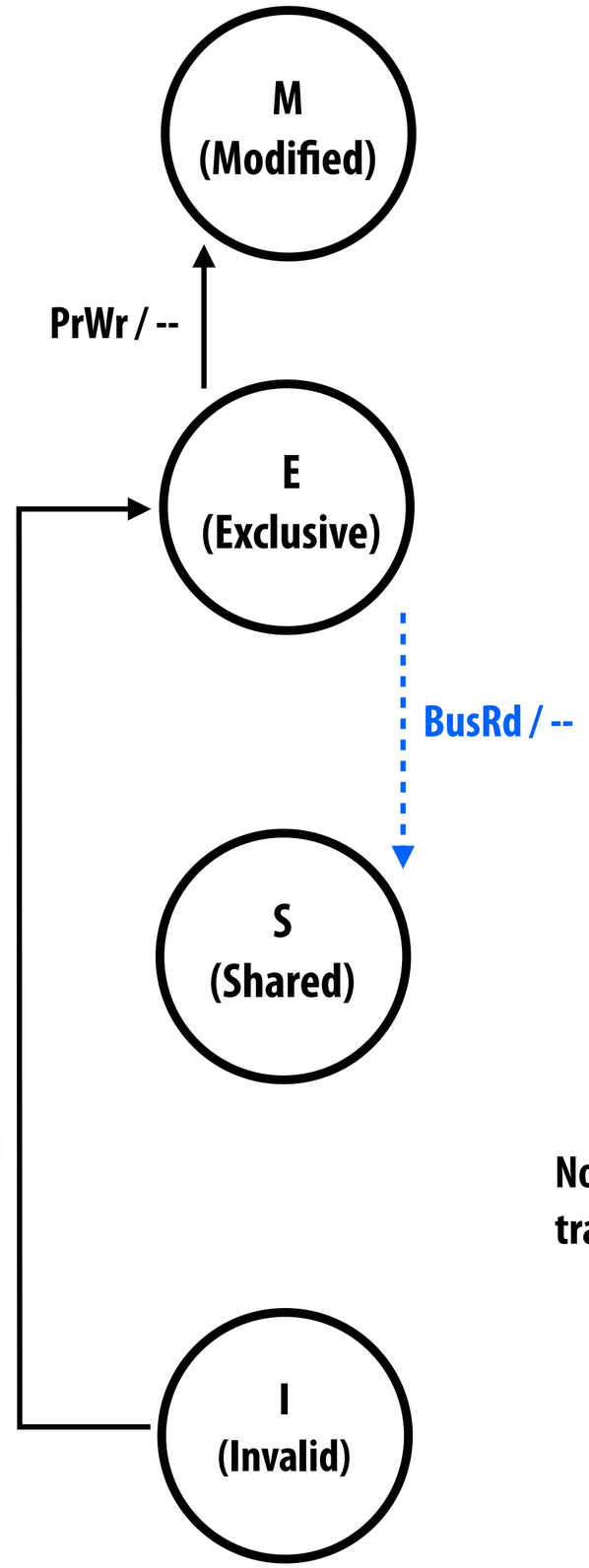
- Cache line ping-pongs between caches of writing processors, generating significant amounts of communication due to coherence protocol

- No inherent communication, all artifactual communication

- Can be a factor in when programming for cache coherent architectures (assignment 3)

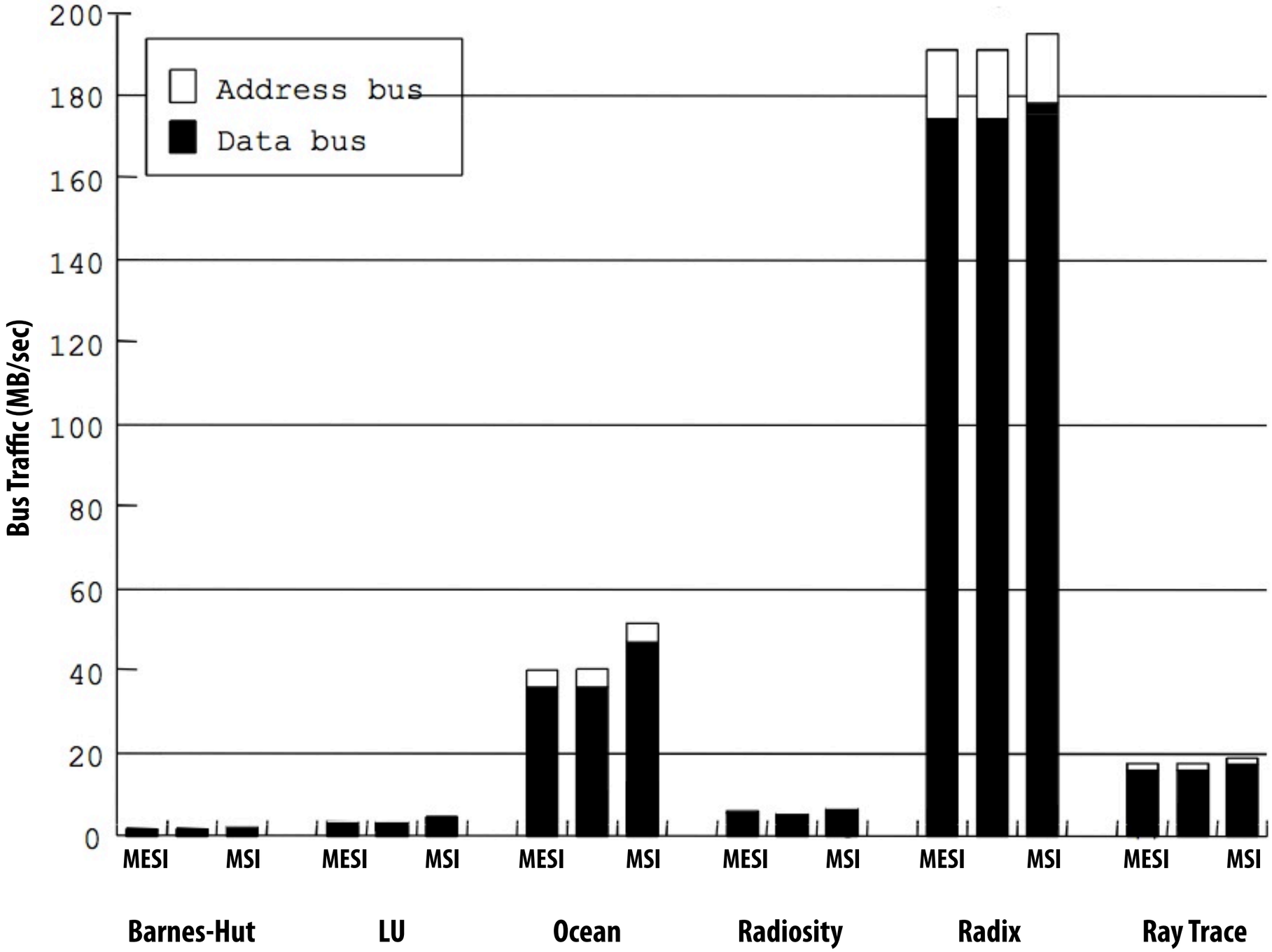
Review: MSI and MESI





Note: only showing transitions unique to MESI

MSI vs. MESI performance study



Extra complexity of MESI does not help much in these applications (best case: about 20% benefit for Ocean) since E → M transitions occur infrequently

A comment on Intel's MESIF

- 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
    - Reduces interconnect traffic: in basic MESI, all caches in S state respond
  - Upon cache read miss (with sharing present), cache line enters F state (rather than S)
    - F state migrates to last cache that loads the line
    - Rationale: this cache is the least likely to evict the line

- **Snooping coherence evaluation:**
  - How does cache block size affect coherence?

- **Upgrade-based coherence protocols**
  - Last time: invalidation-based protocols

- **Coherence with multi-level cache hierarchies**
  - How do multi-level hierarchies complicate implementation of snooping?

Impact of cache block size

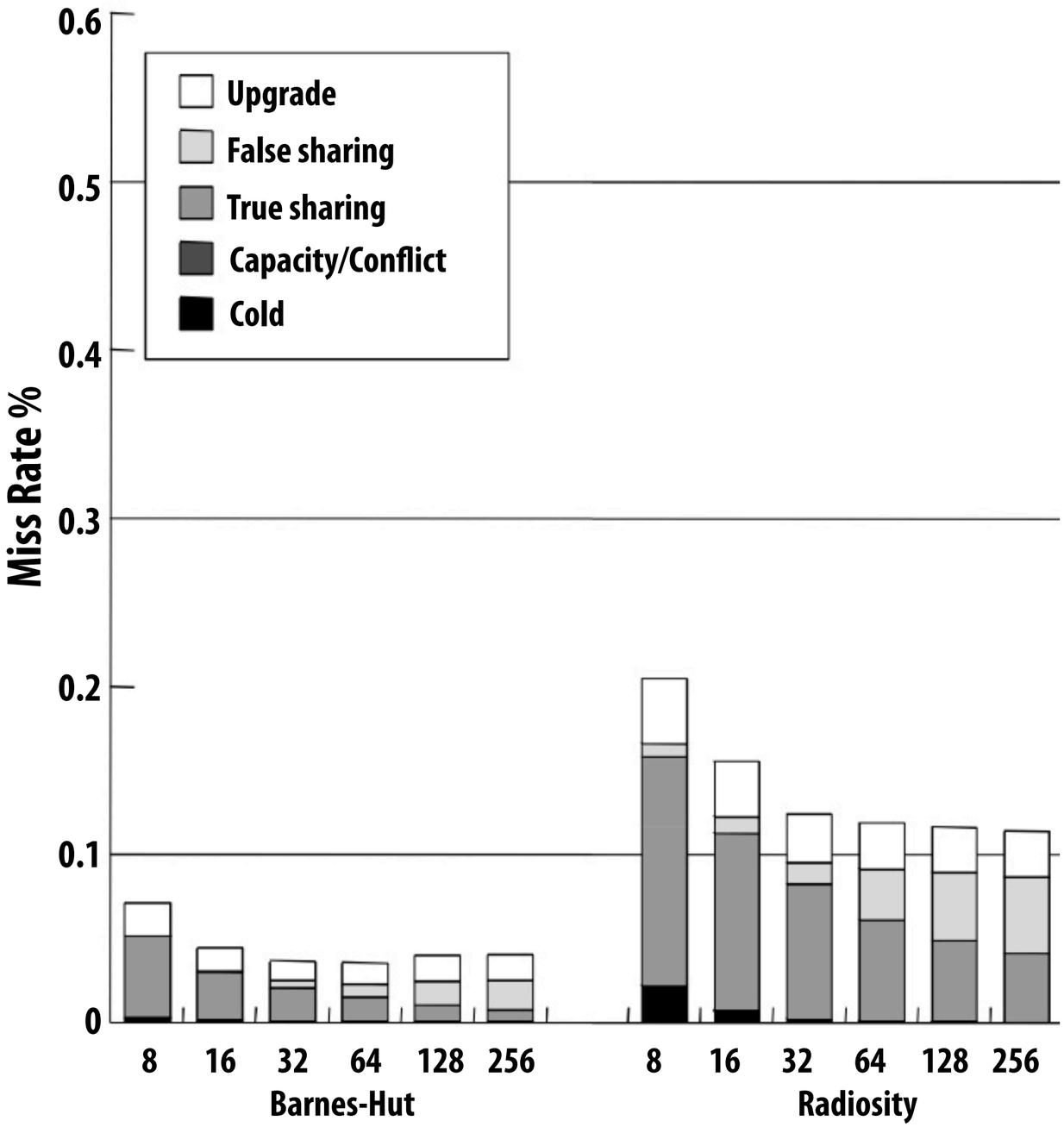
- Recall that cache coherence adds a fourth type of miss: coherence misses

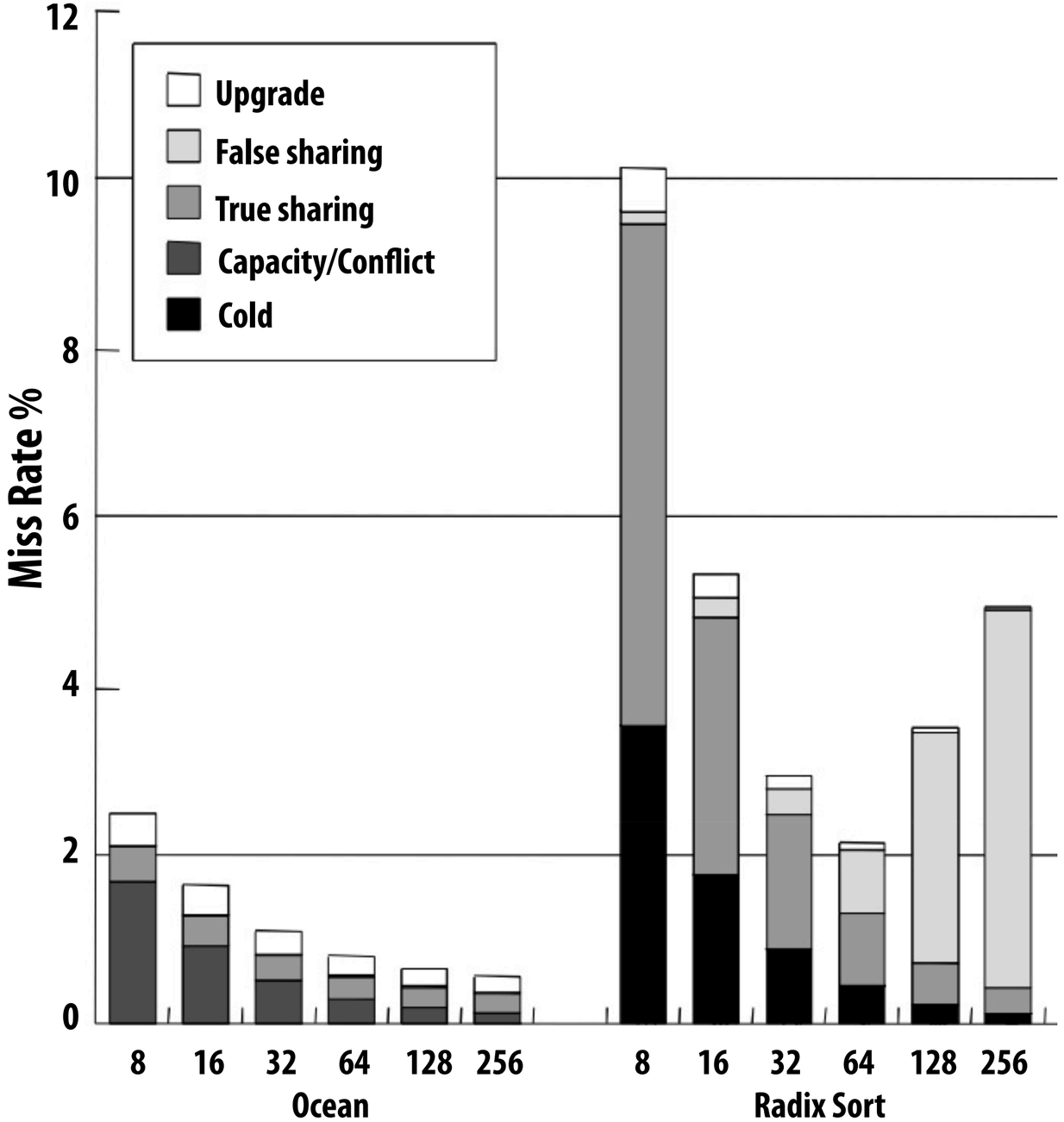
- How to reduce cache misses:
  - Capacity miss: enlarge cache, increase block size
  - Conflict miss: increase associativity
  - Cold/true sharing coherence: increase block size

- How can larger block size hurt? (assume: fixed-size cache)
  - Increase cost of a miss (larger block to load into cache)
  - Can increase misses due to conflicts
  - Can increase misses due to false sharing

Impact of cache block size: miss rate

Simulated 1 MB cache

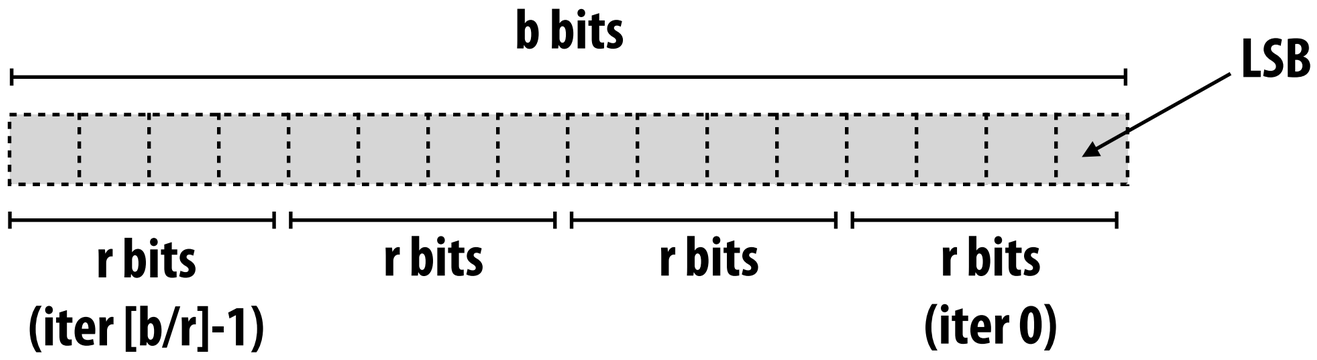




Example: parallel radix sort

Sort array of N, b-bit numbers

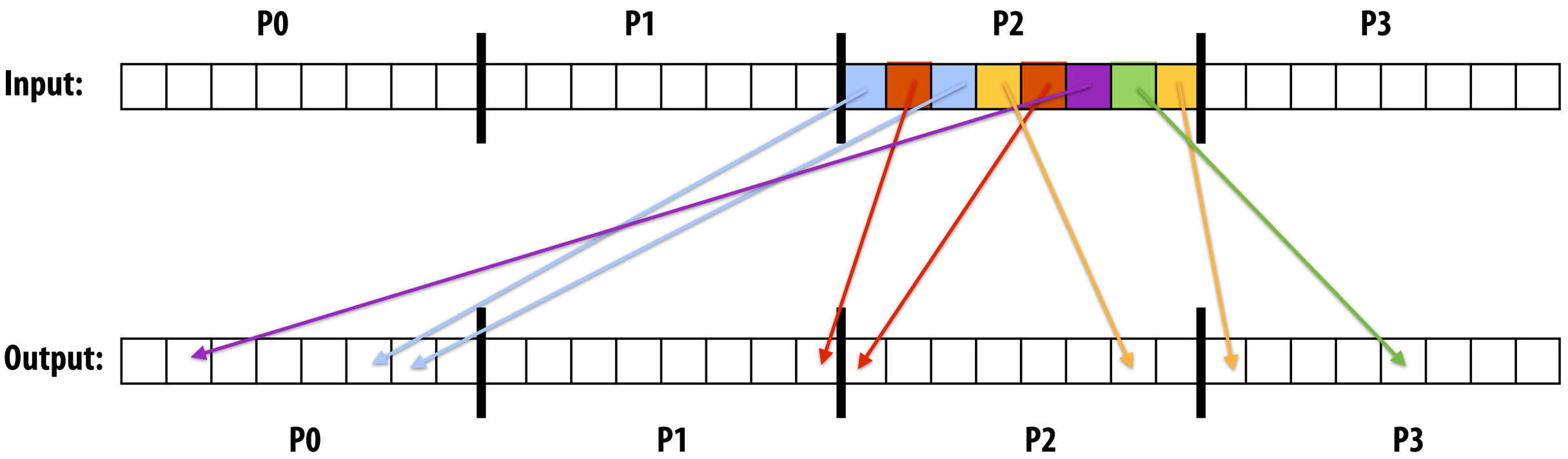
Here: radix = $2^4 = 16$



For each group of r bits (this is serial iteration)

In parallel, sort numbers by group value

(by placing numbers into bins)

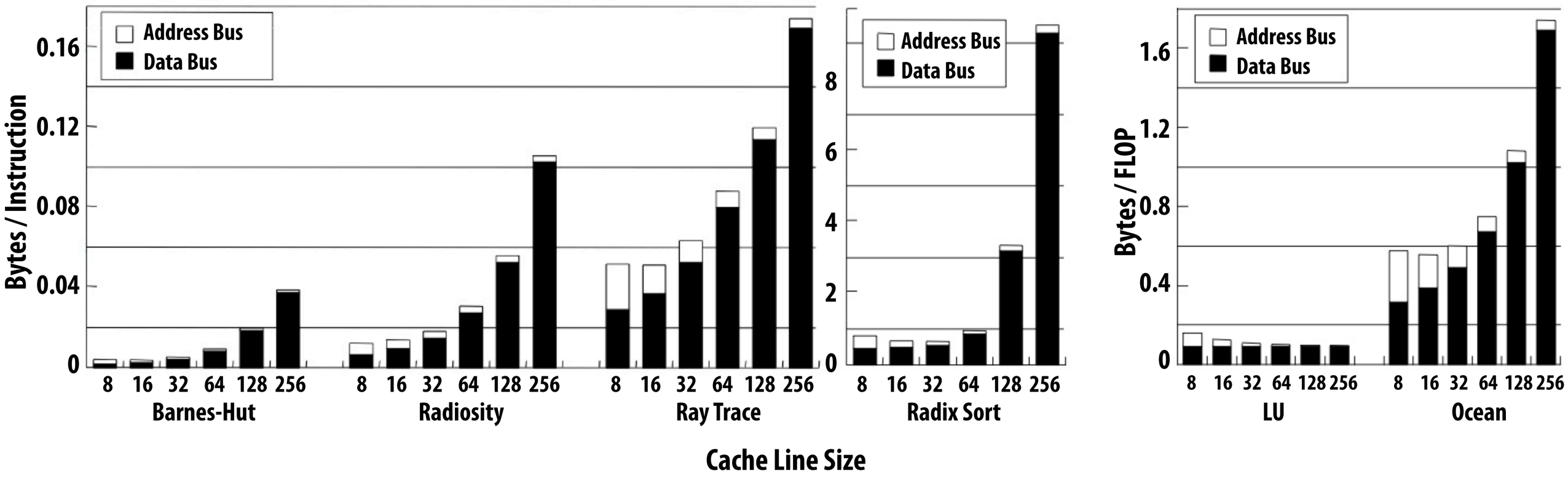


Potential for lots of false sharing

False sharing decreases with increasing array size

Impact of cache block size: traffic

Simulated 1 MB cache



Some thoughts

- In general, larger cache lines:
  - Fewer misses
  - But more traffic (unless spatial locality is perfect)

- Which should we prioritize?
  - Extra traffic okay if magnitude of traffic isn't approaching capability of interconnect
  - Latency of miss okay if processor has a way to tolerate it (e.g., multi-threading)

- These are just notions. If you were building a system, you would simulate on many important apps and make decisions based on your graphs and needs

Update-based coherence protocols

- Thus far, we've talking only about invalidation-based protocols
  - Main idea: cache obtains exclusive access to line in order to write to it
  - Possible issues:
    - Cache must reload entire line after invalidation
    - False sharing

- Invalidation-based protocols most commonly used today
  - But let's talk about one update-based protocol for fun

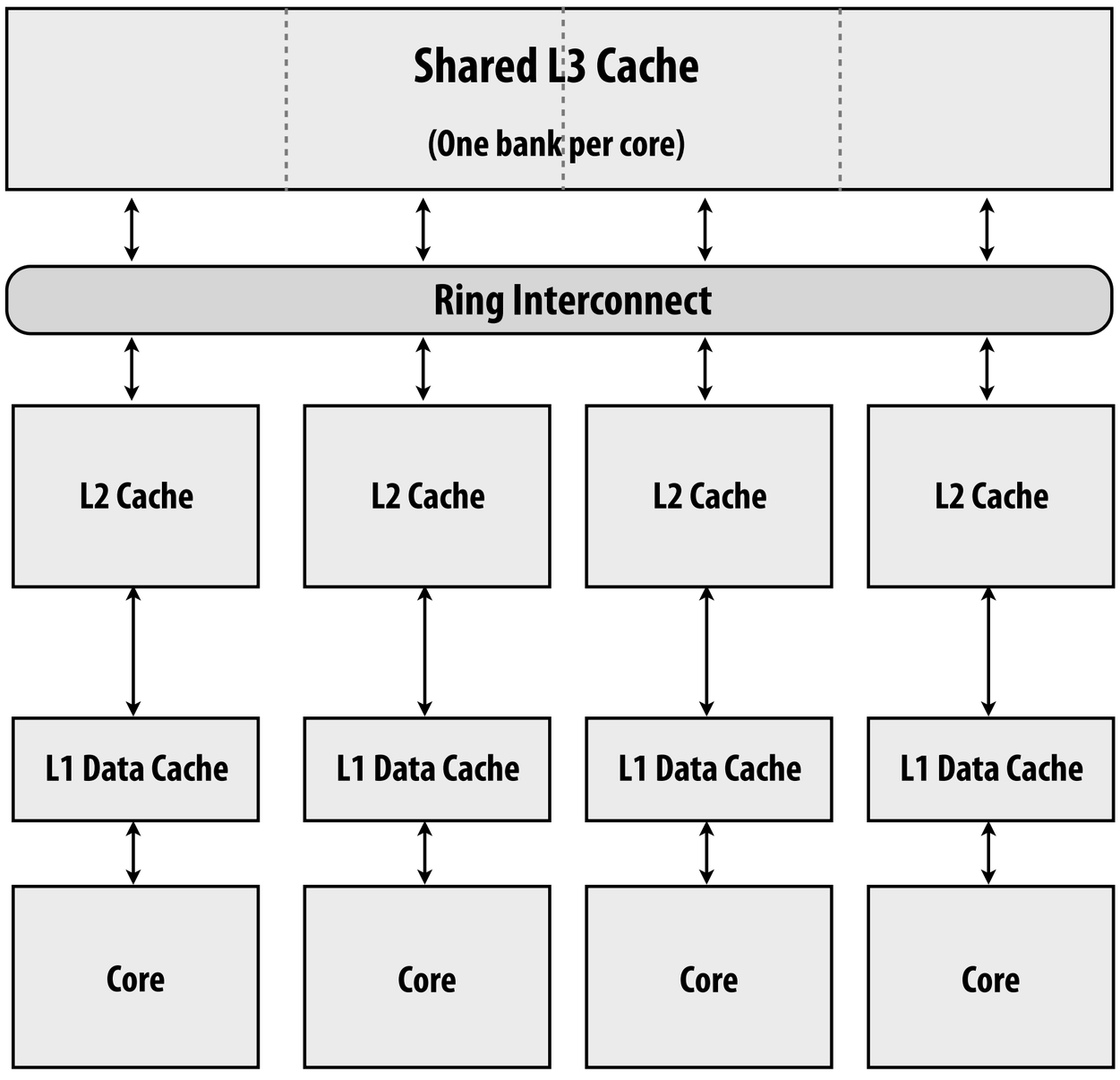
Invalidate vs. update

- Intuitively, upgrade would seem preferable if other processors sharing data will continue to access it after a write

- Upgrades are overhead if:
  - Data just sits in cache (and is never used again)
  - Lots of writes before the next read

Reality: multi-level cache hierarchies

Recall Intel Core i7 hierarchy



- Challenge: changes made 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"

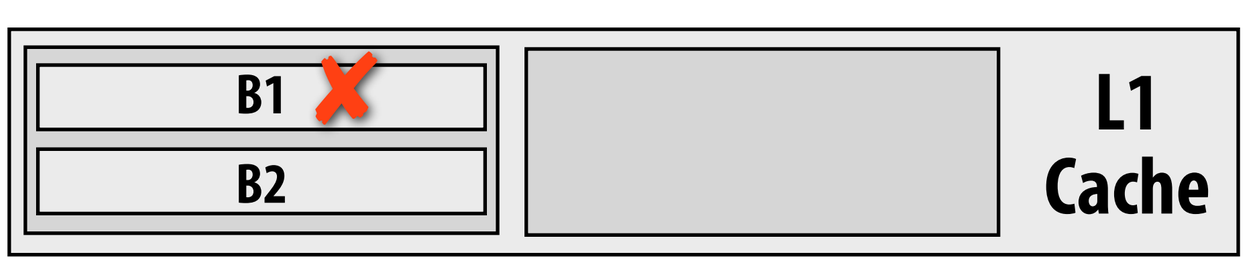
Inclusion property of caches

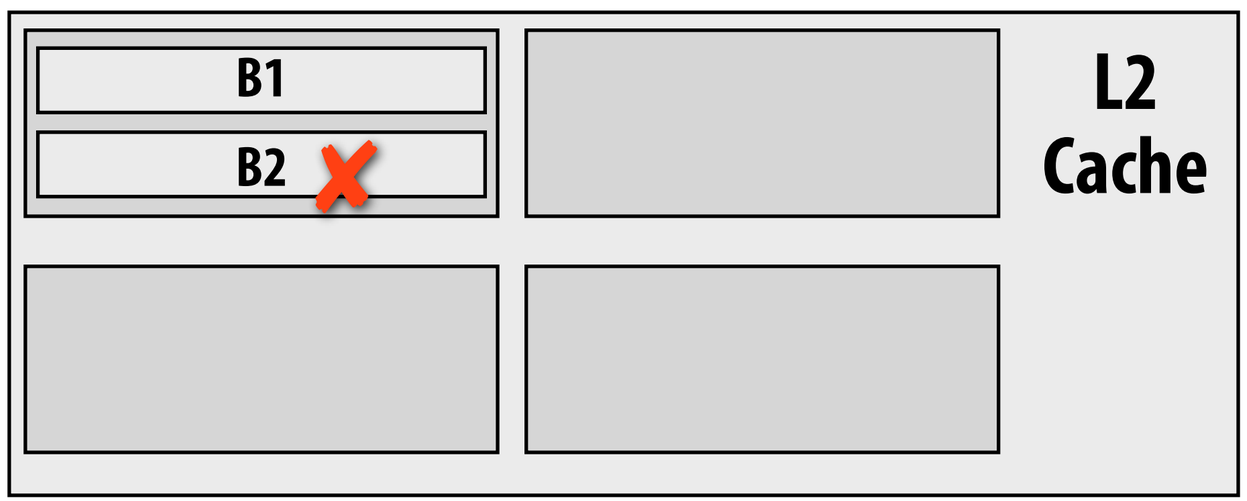
- All lines in closer [to processor] cache are in farther cache
  - e.g., contents of L1 are a subset of contents of L2
  - Thus, all transactions relevant to L1 are also relevant to L2, so it is sufficient for only the L2 to snoop the interconnect

- If line is in owned state (M in MESI, M or O in MOESI) in L1, it must also be in owned state in L2
  - Allows L2 to determine if a bus transaction is requesting a modified block in L1 without requiring information from L1

Is inclusion maintained automatically if L2 is larger than L1? No.

- Simple example:
  - Let L2 cache be twice as large as L1 cache
  - L1 an L2 have the same block size, are 2-way set associative, and use a LRU replacement policy
  - Let blocks B1, B2, B3 map to the same set of the L1 cache
  - B1 and B2 are resident in the L1 and L2 caches





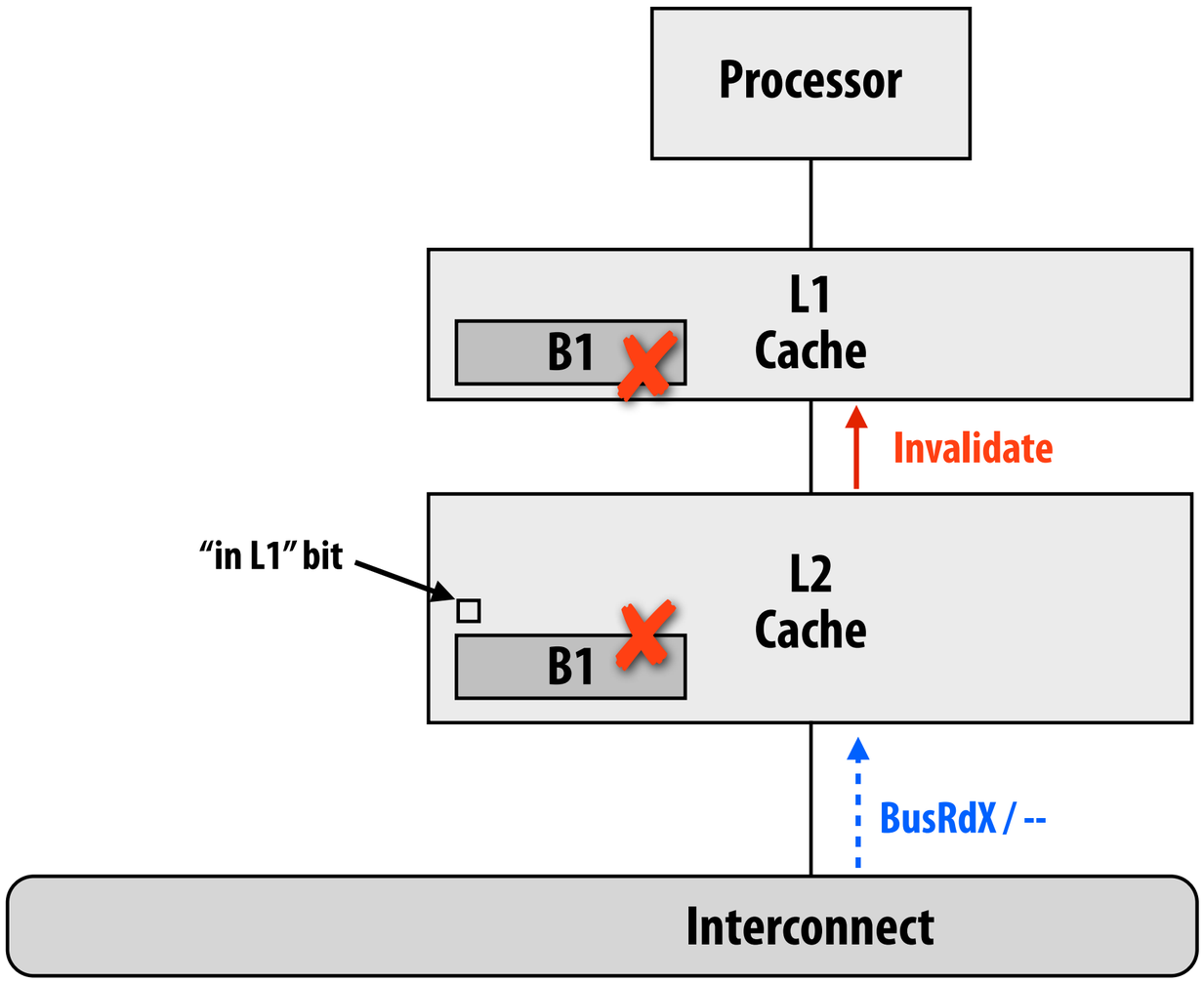
Processor references to B1 and B2 are serviced by L1 cache. The access history to B1 and B2 are different in the L1 than in the L2!

Say processor accesses B1 (L1+L2 miss). Then B2 (L1+L2 miss). Then B1 many times (L1 hits).

Now access B3. L1 and L2 might choose to evict different blocks, because access histories differ.

Inclusion no longer holds!

Maintaining inclusion: handling invalidations



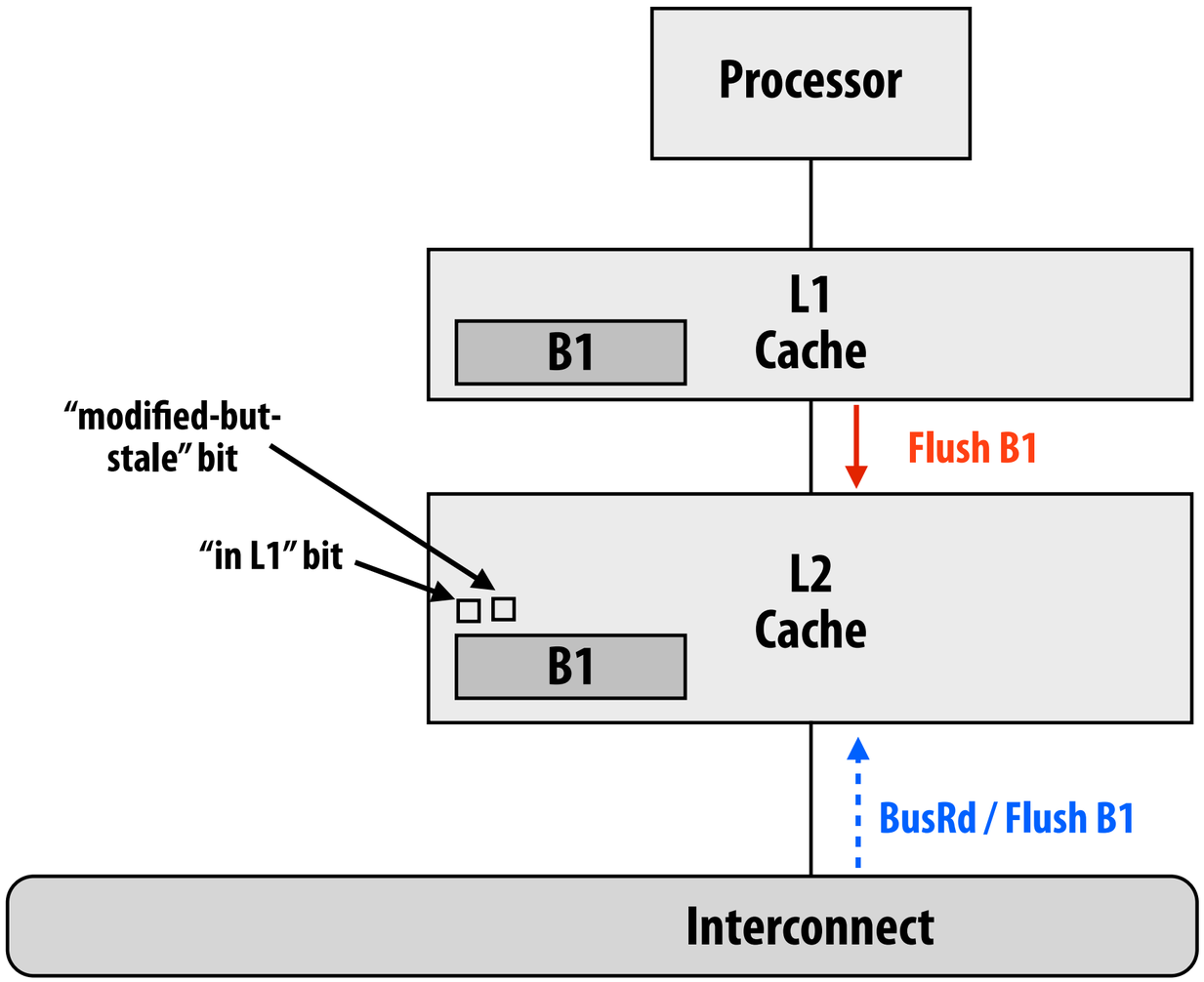
Block invalidated in L2 cache due to BusRdX from another cache.

Must also invalidate block in L1

One solution: each L2 block maintains a bit indicating if block also exists in L1

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

Maintaining inclusion: L1 write hit



Assume L1 is a write-back cache. Processor writes to block B1. (L1 write hit)

Block B1 in L2 cache is in modified state in the coherence protocol, but it has stale data!

When coherence protocol requires B1 to be flushed from L2 (e.g., another processor loads B1), L2 cache must request the data from L1.

Add another bit for "modified-but-stale"

Snooping based cache coherence summary

- Main idea: cache operations that effect coherence are broadcast to all other caches
- Caches listen ("snoop") for these messages, react accordingly
- Multi-level cache hierarchies add complexity to implementations
- Workload driven evaluation: Larger cache block sizes...
  - Decrease cold, capacity, true sharing misses
  - Can increase false sharing misses
  - Increase interconnect traffic

- Scalability of snooping implementations limited by ability to broadcast coherence messages to all caches
  - Snooping used in smaller-scale multiprocessors

    (such as the multi-core chips in all our machines today)
  - Next time: scaling cache coherence via directory-based approaches