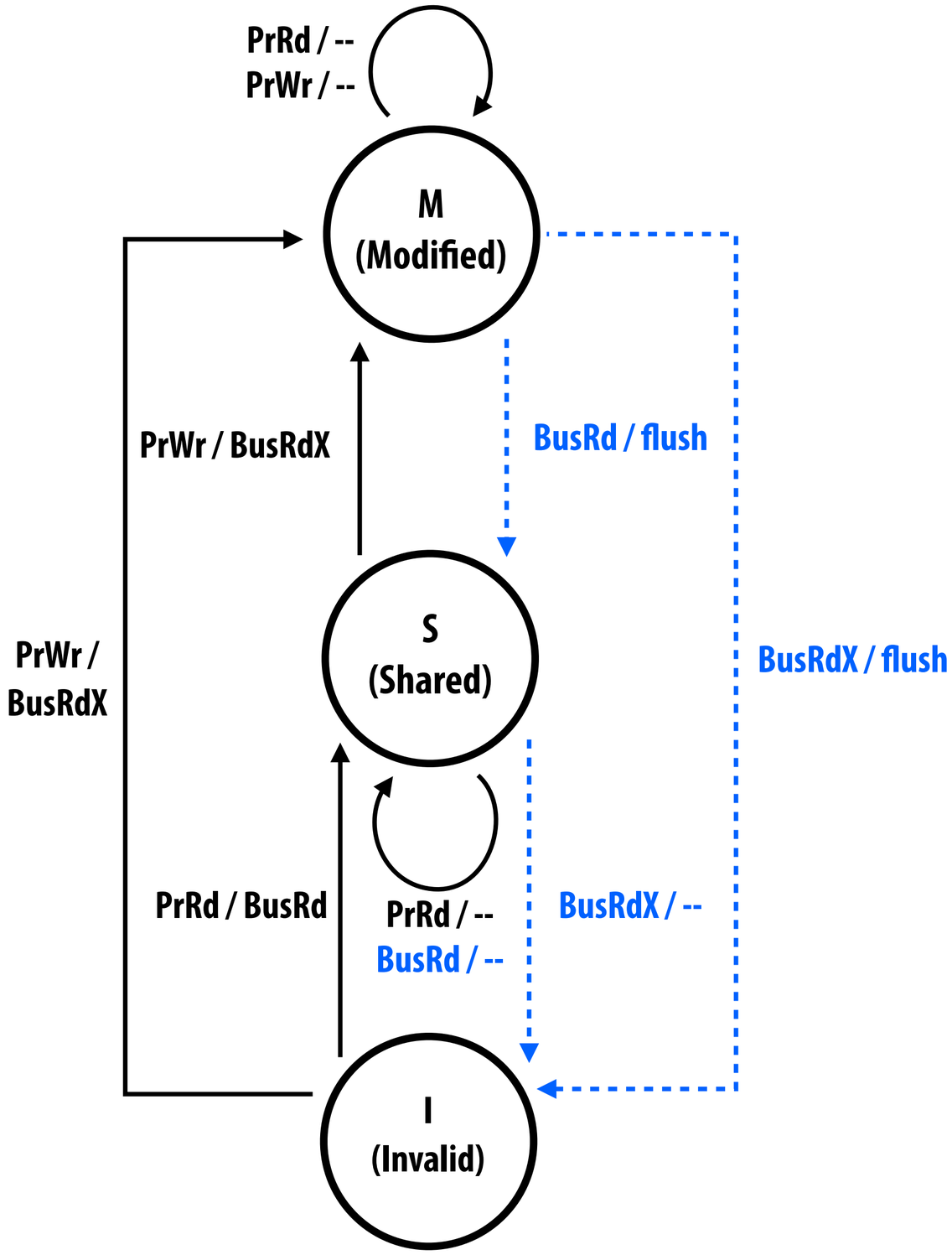
Lecture 11:

Cache Coherence: Part II

Parallel Computer Architecture and Programming
CMU 15-418, Spring 2013

Demo: MSI coherence protocol

(Assume X and Y are initialized to 0 and lie on different cache lines)



Proc 0: Load X

Proc 1: Load X

Proc 0: Store 1 →X

Proc 1: Store 10 →X

Proc 1: Store 25 →X

Proc 0: Load X

Proc 1: Load X

Proc 1: Store 100 →Y

Proc 1: Load X

Review: false sharing

What could go wrong if you declared per-thread variables this way in a cache-coherent system?

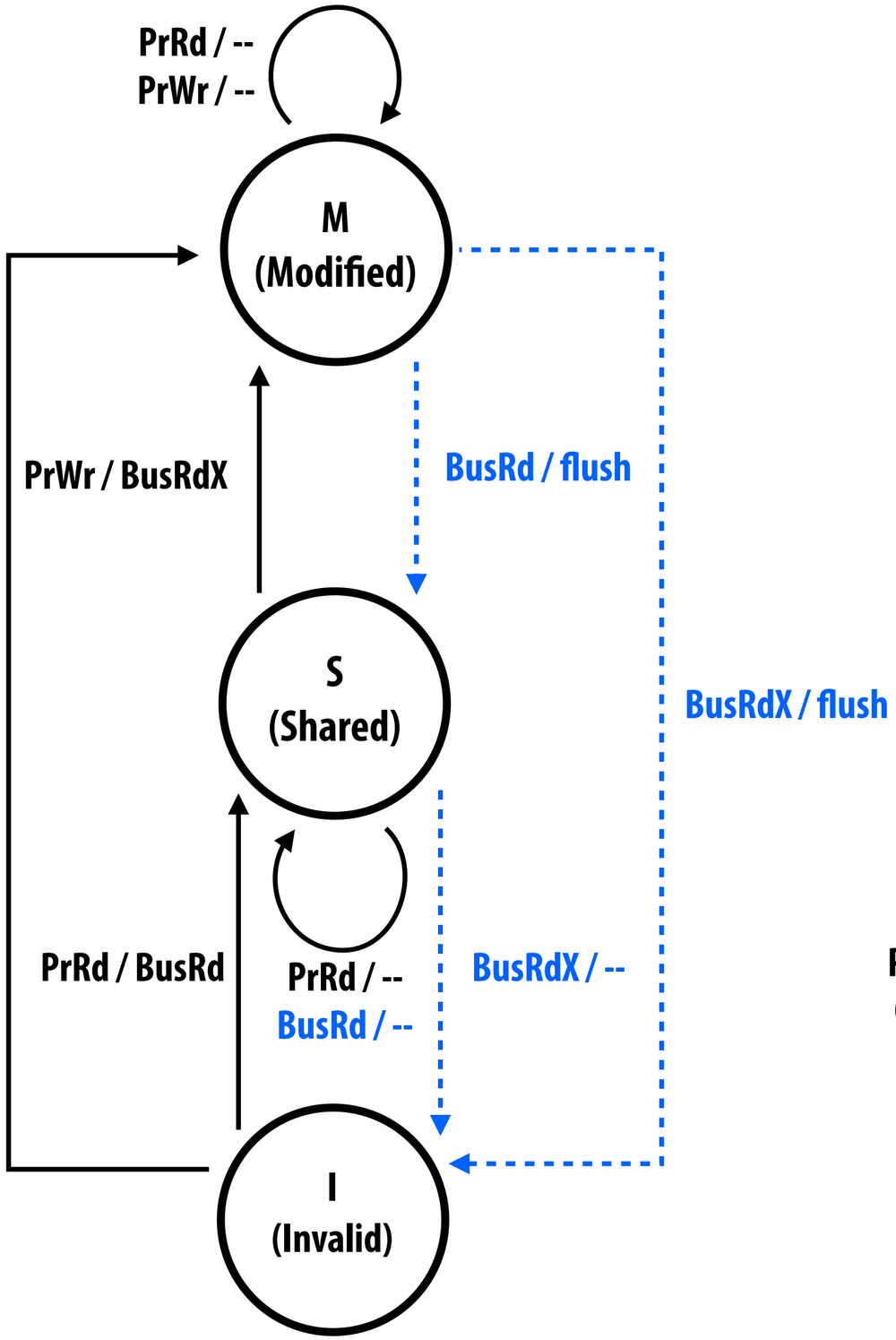
```
// allocate variables intended for per-thread accumulation
int myCounter[NUM_THREADS];
```

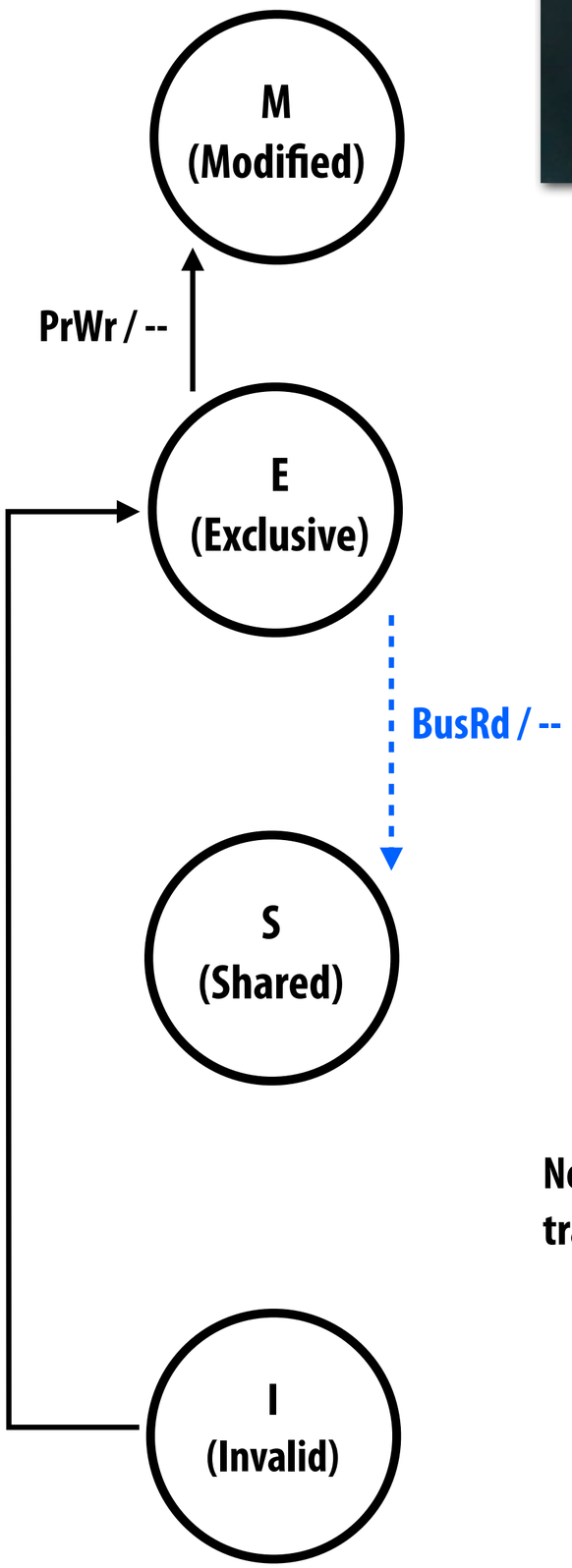
Why might this code perform better?

```
// assume 64-byte cache line
struct PerThreadState {
  int myCounter;
  char padding[64 - sizeof(int)];
};

// allocate variables intended for per-thread accumulation
PerThreadState myCounter[NUM_THREADS];
```

Review: MSI and MESI

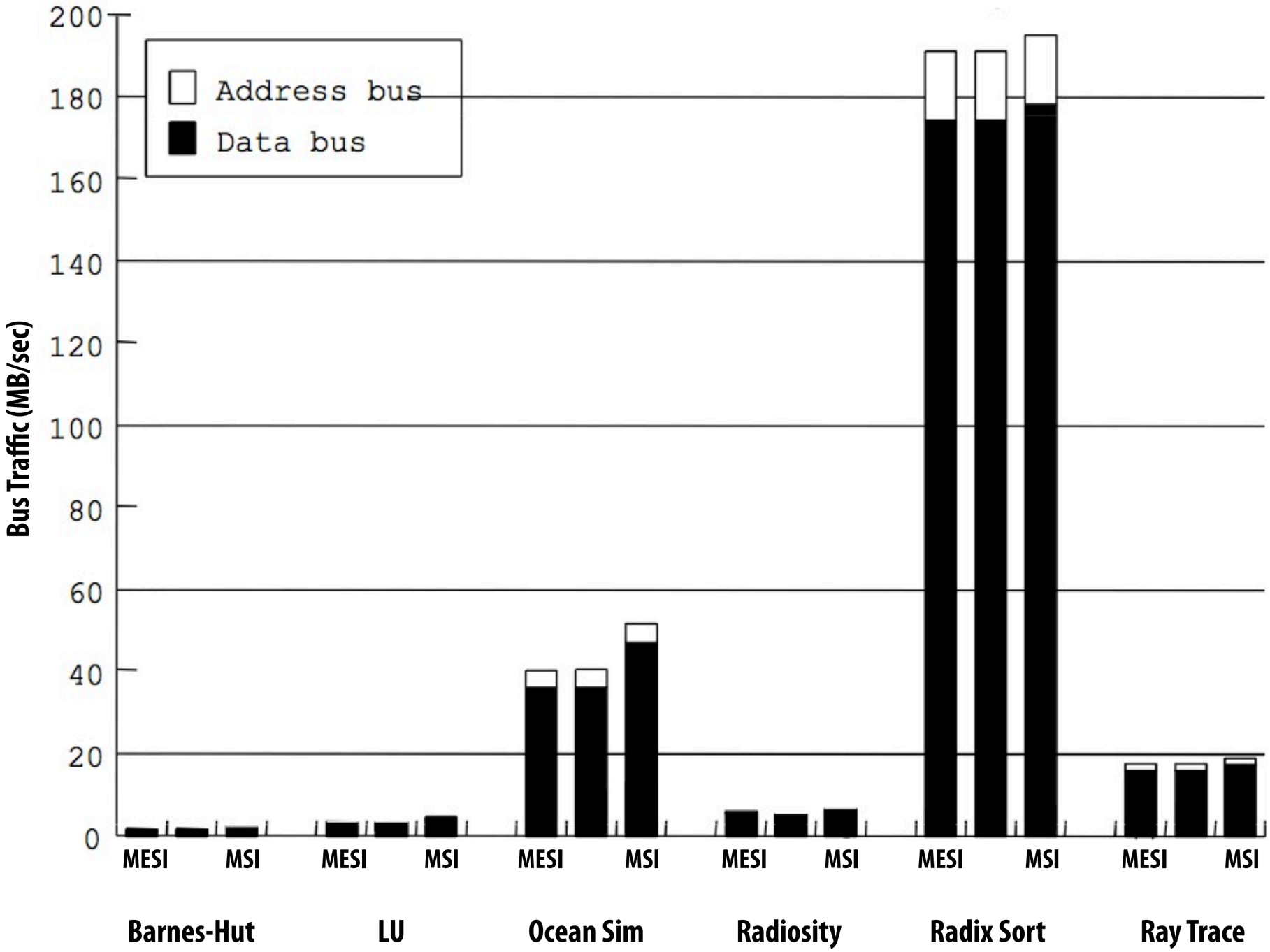




MESI, not Messi !

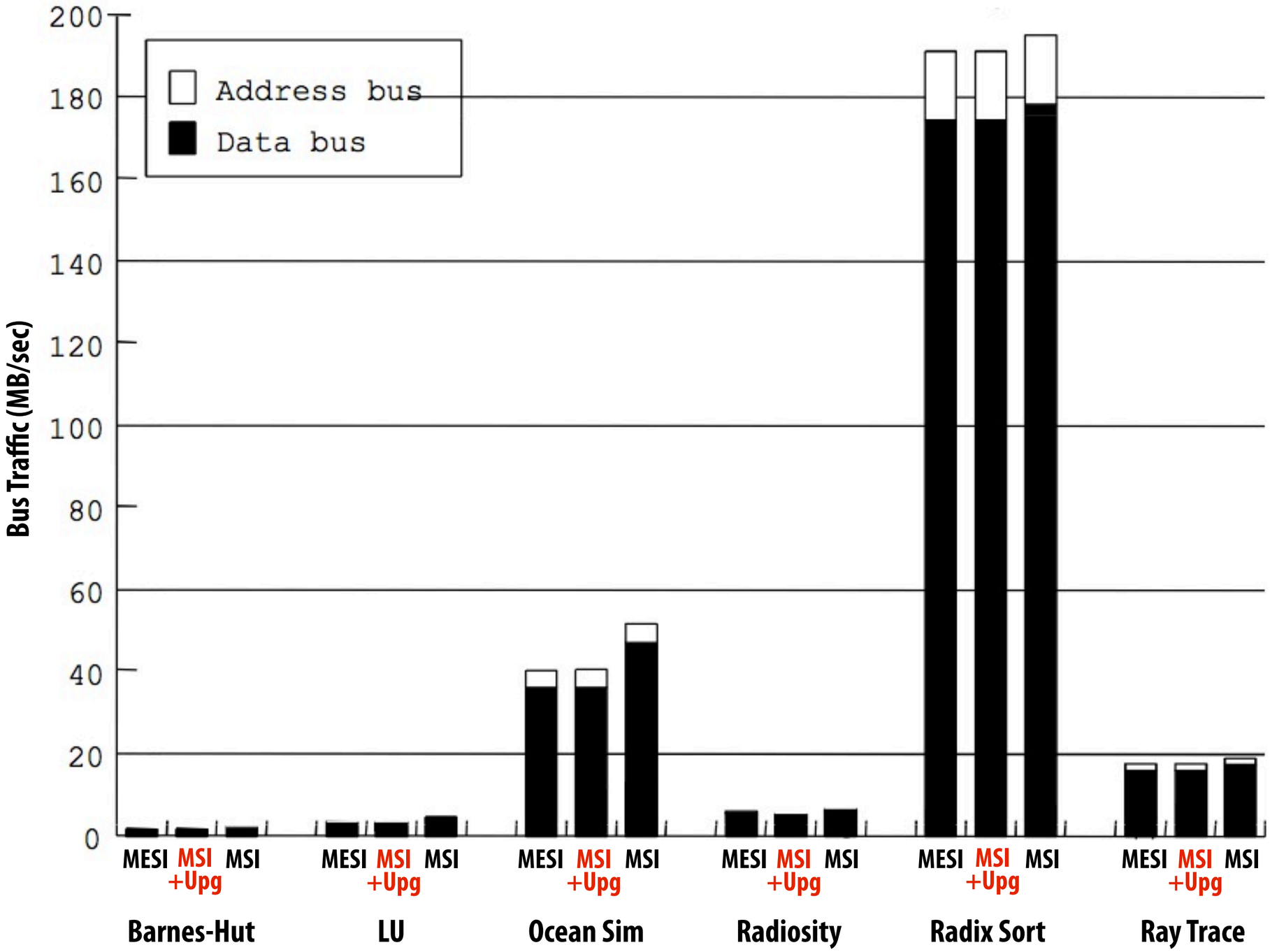
Note: only showing transitions unique to MESI

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

MSI with BusUpgr transaction



Data transferred on bus for E → M transition is small if "upgrade" transaction used, rather than BusRdX (no need to transfer a cache line from memory, just broadcast BusUpgr)

Here: larger benefit achieved from adding support for upgrade to MSI than implementing full MESI protocol

Today's topics

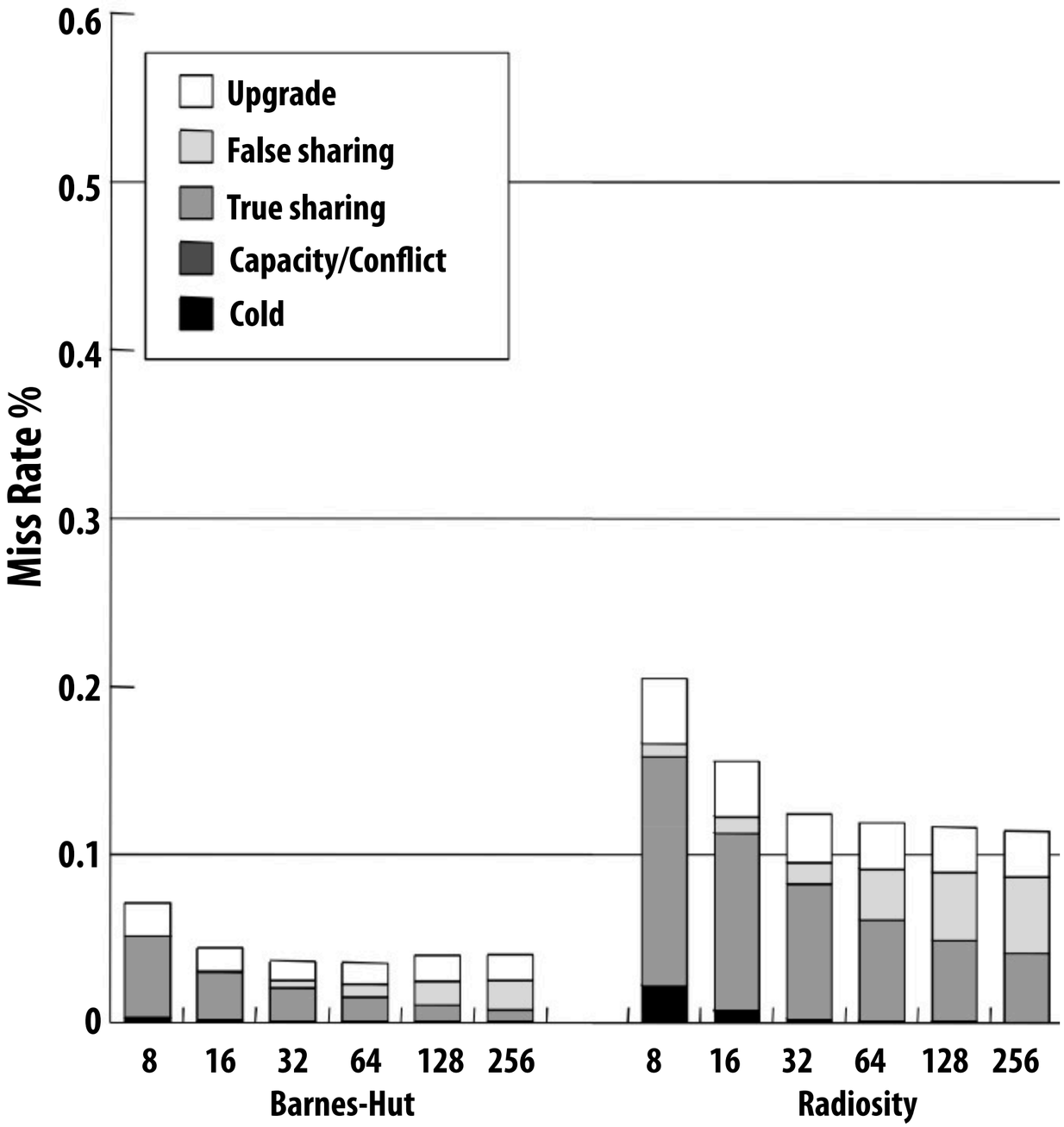
- Snooping coherence evaluation:
  - How does cache line size affect coherence?
- Upgrade-based coherence protocols
  - Last time: invalidation-based protocols
- Coherence with multi-level cache hierarchies
  - How do multi-level hierarchies complicate implementation of snooping?

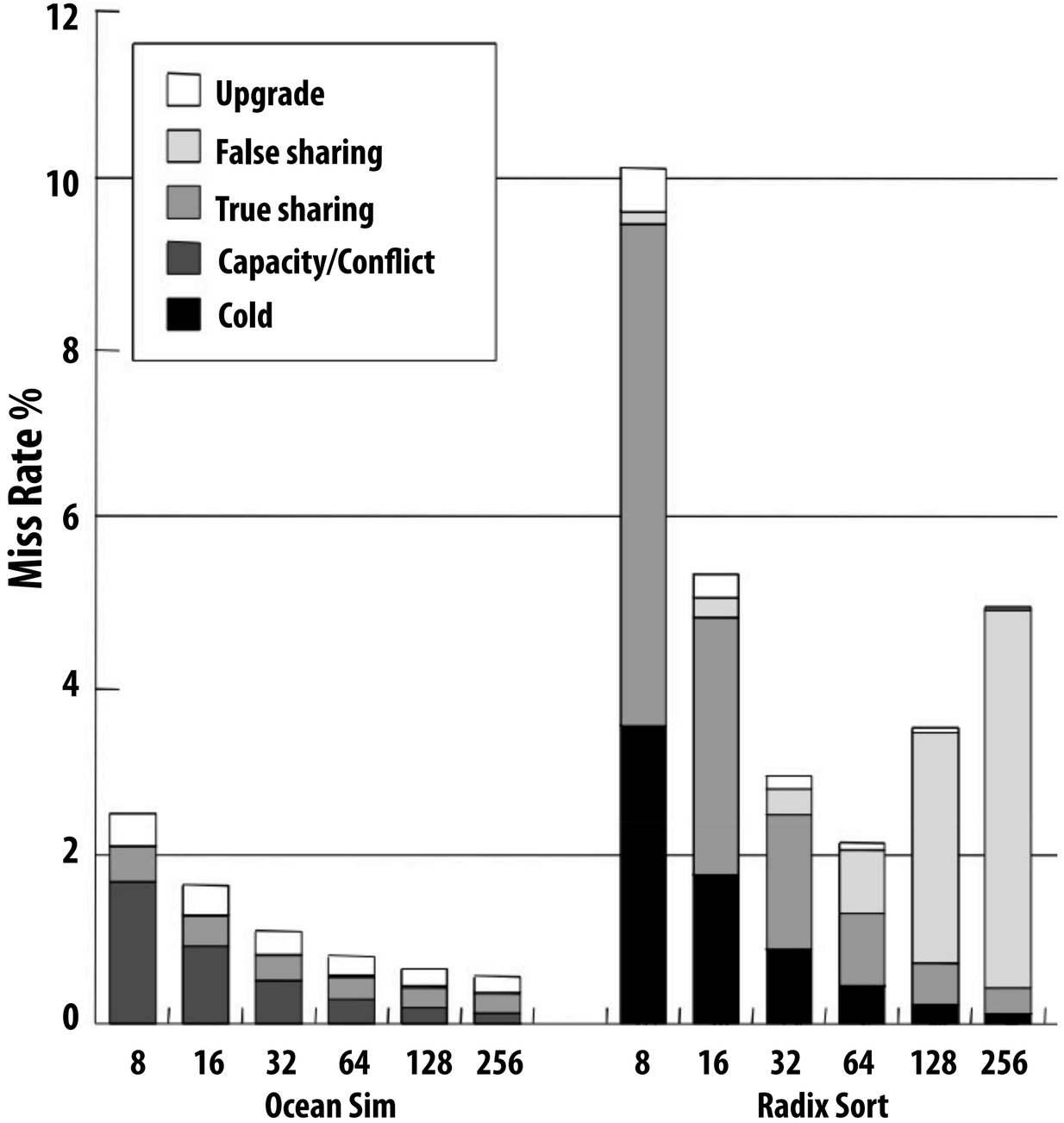
Impact of cache line size

- Cache coherence adds a fourth type of miss: coherence misses
- How to reduce cache misses:
  - Capacity miss: enlarge cache. (Also increase line size: if spatial locality present)
  - Conflict miss: increase cache associativity or change application data layout
  - Cold/true sharing coherence miss: increase line size (if spatial locality present)
- How can larger line size hurt? (assume: fixed-size cache)
  - Increase cost of a miss (larger line to load into cache increases miss latency)
  - Can increase misses due to conflicts
  - Can increase misses due to false sharing

Impact of cache line size on miss rate

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



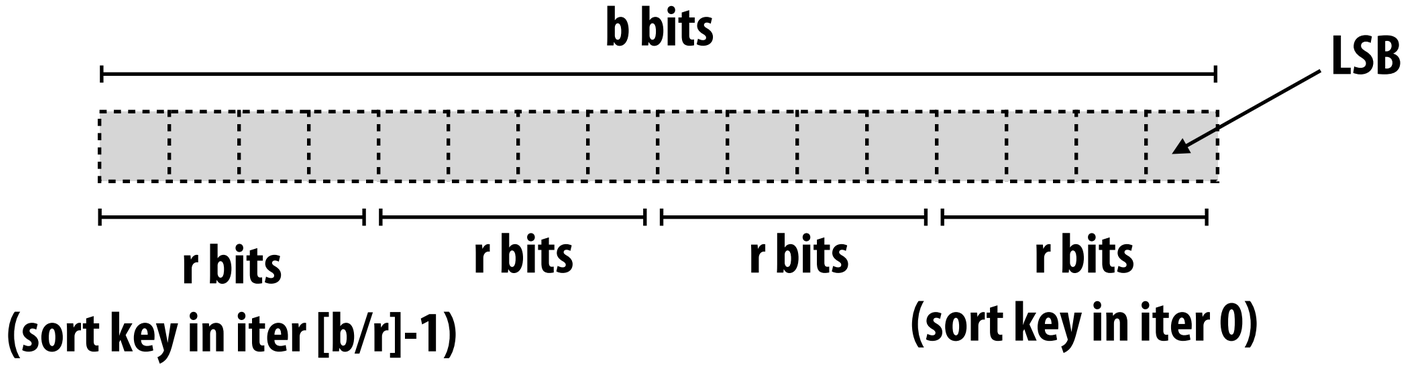


*** Note: I separated the results into two graphs because of different Y-axis scales**

Example: parallel radix sort

Sort array of N, b-bit numbers

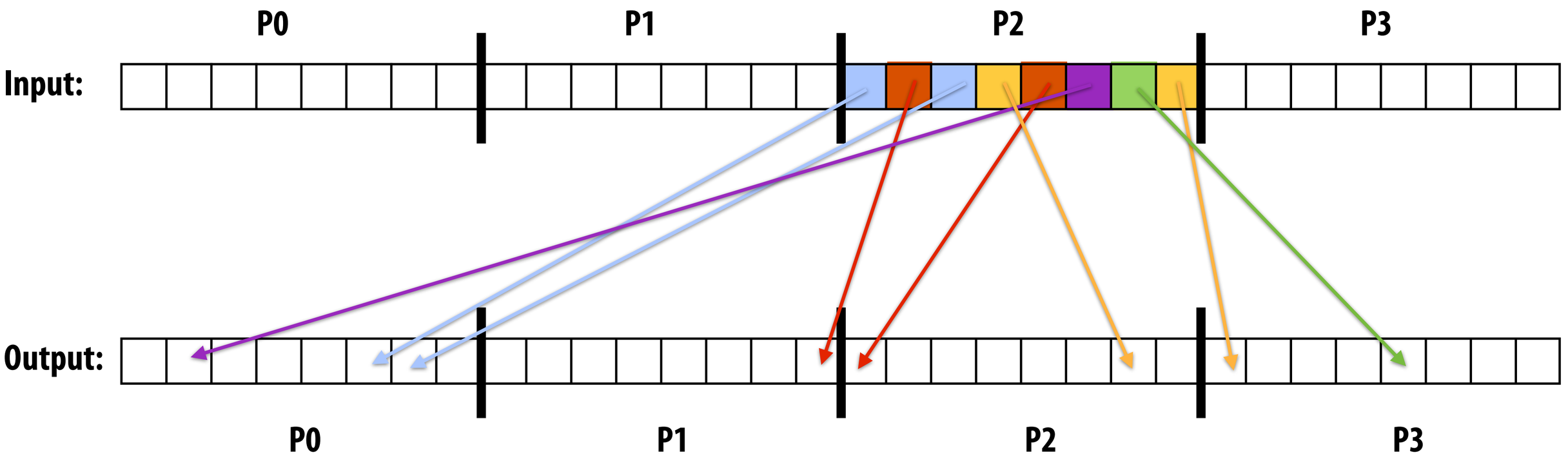
Here: radix = $2^4 = 16$



For each group of r bits: (serial iteration)

In parallel, sort numbers by group value

(by placing numbers into bins)

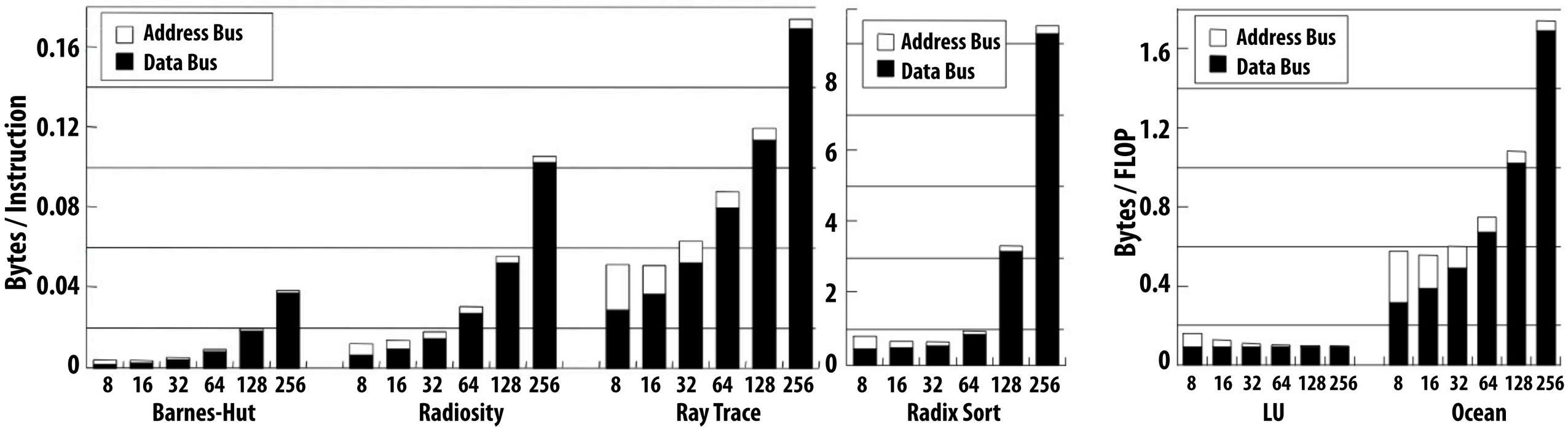


Potential for lots of false sharing

False sharing decreases with increasing array size

Impact of cache line size on interconnect traffic

Results from simulation of a 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 using many important apps and make decisions based on your graphs and needs. WORKLOAD-DRIVEN OPTIMIZATION!

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 BW)
    - False sharing

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

Dragon write-back protocol

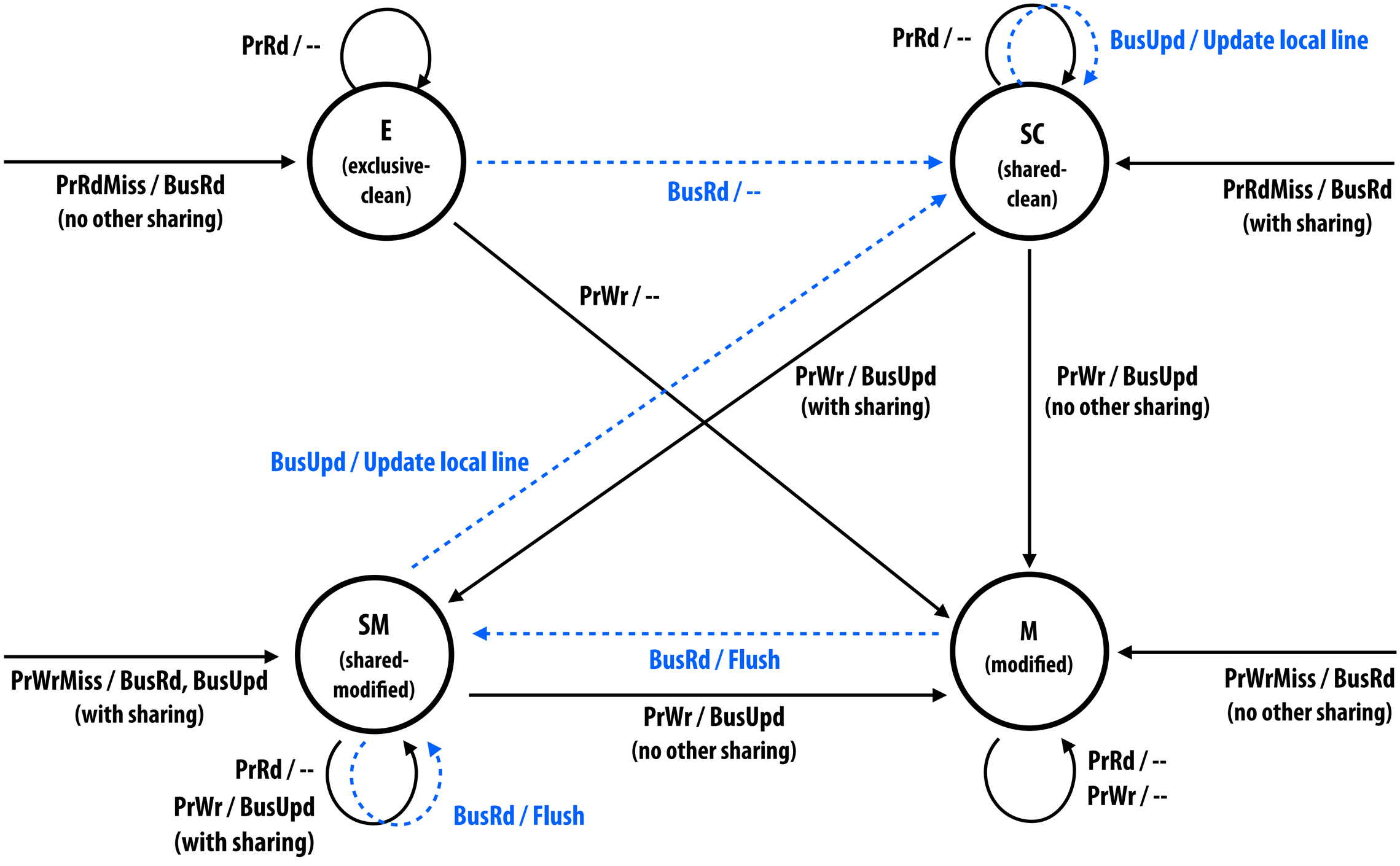
- **States:** (no invalid state, but can think of lines as invalid before loaded for the first time)
  - Exclusive-clean (E): only one cache has line, memory up-to-date
  - Shared-clean (SC): multiple caches may have line, and memory may or may not ** be up to date
  - Shared-modified (SM): multiple 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 replacement
  - 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 (write back), 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



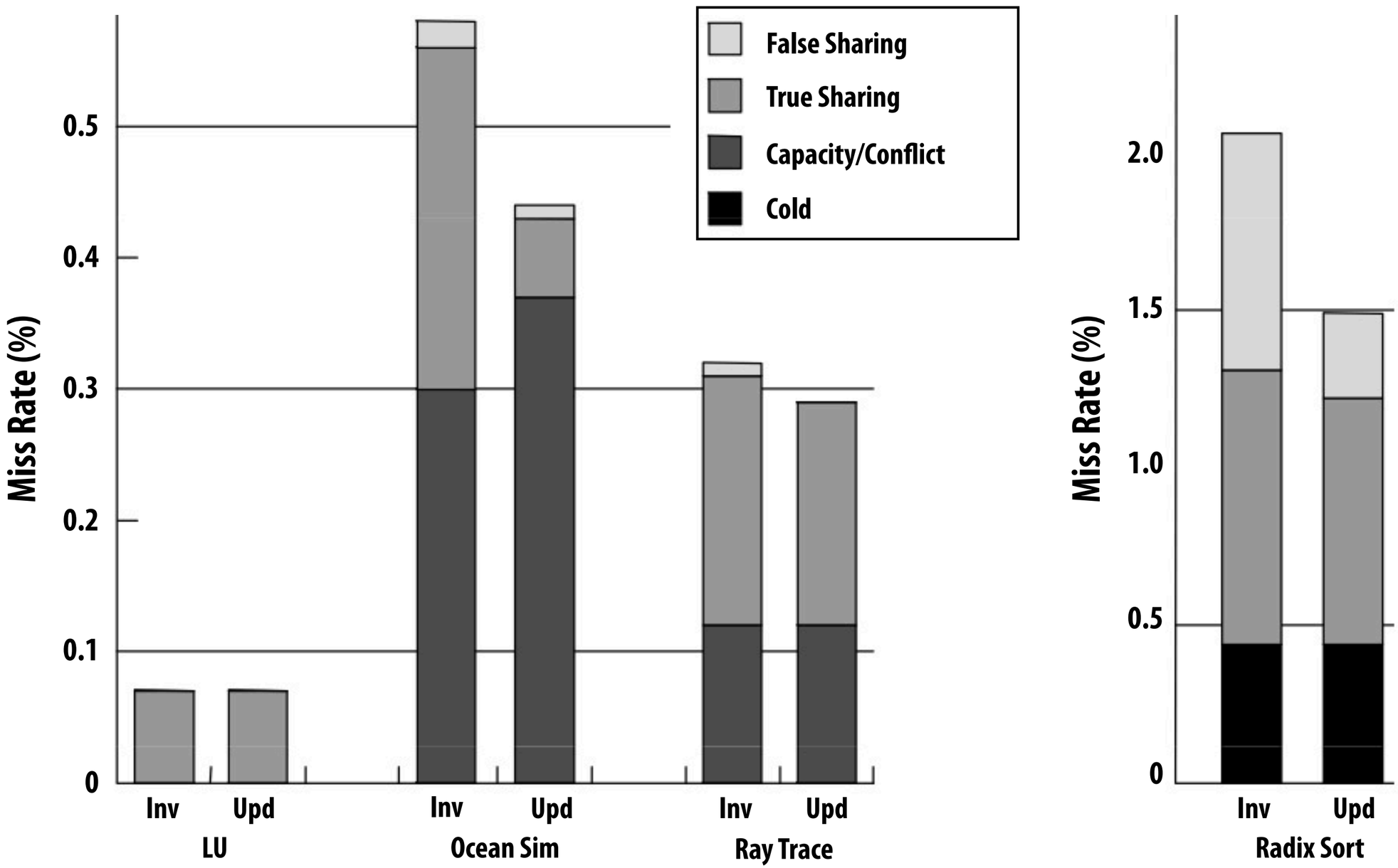
Not shown: upon line replacement, cache must write line to memory if line is in SM or M state

Invalidate vs. update

- Which is better?
- 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 read by another processor again)
  - Application performs many writes before the next read

Invalidate vs. update evaluation: miss rate

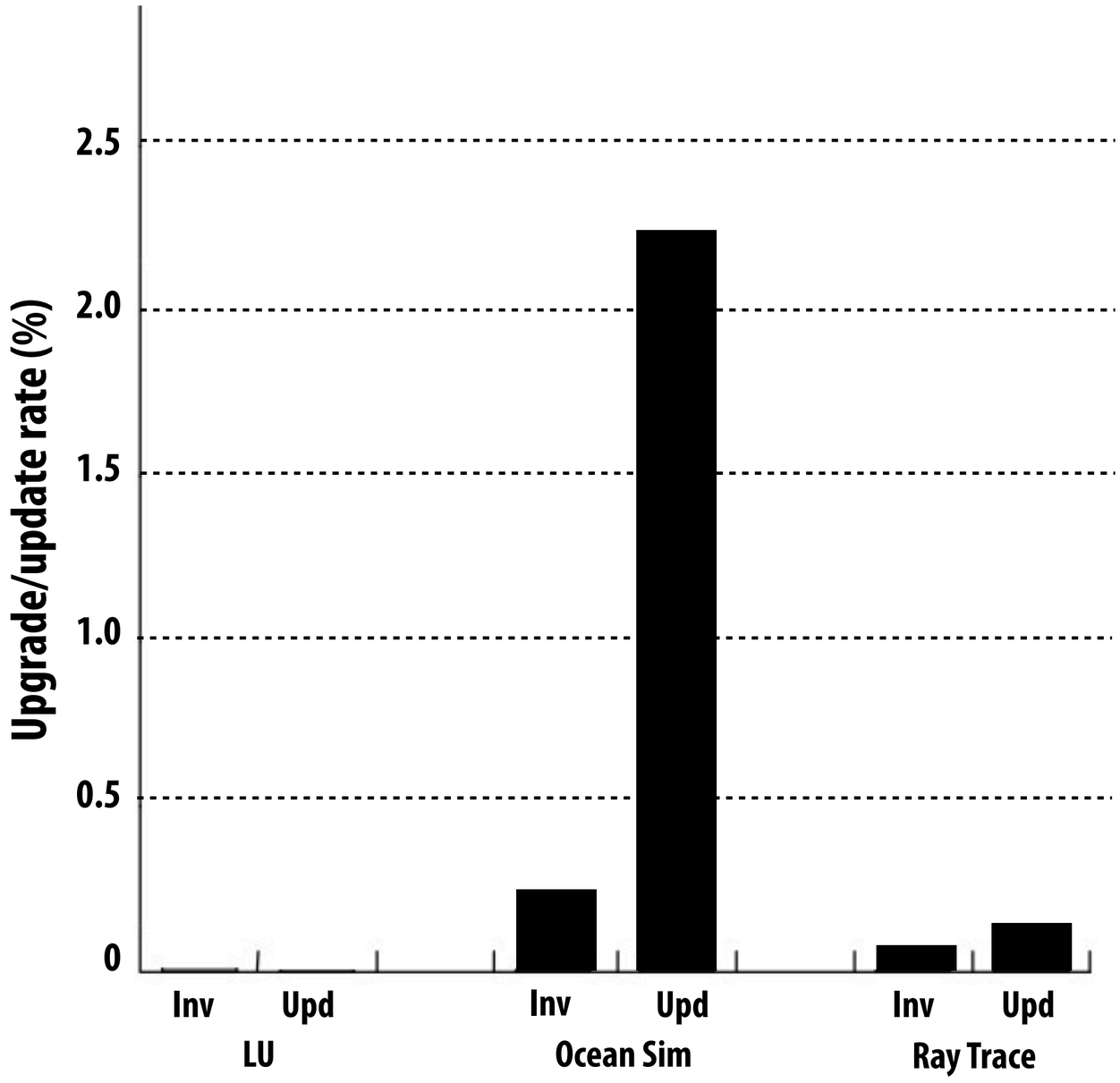
Simulated 1 MB cache, 64 byte lines

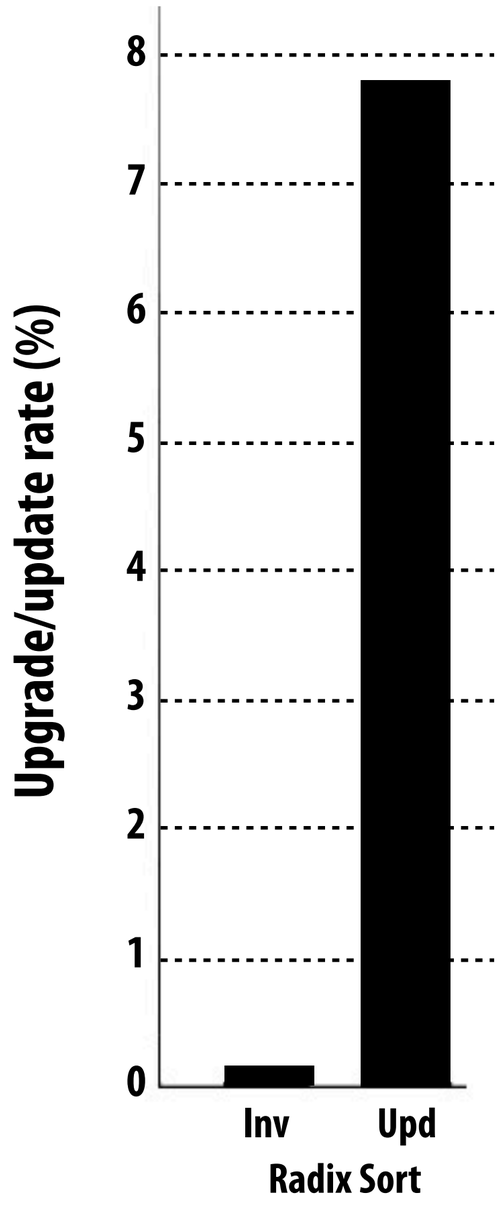


So... is update better?

Invalidate vs. update evaluation: traffic

Simulated 1 MB cache, 64-byte lines



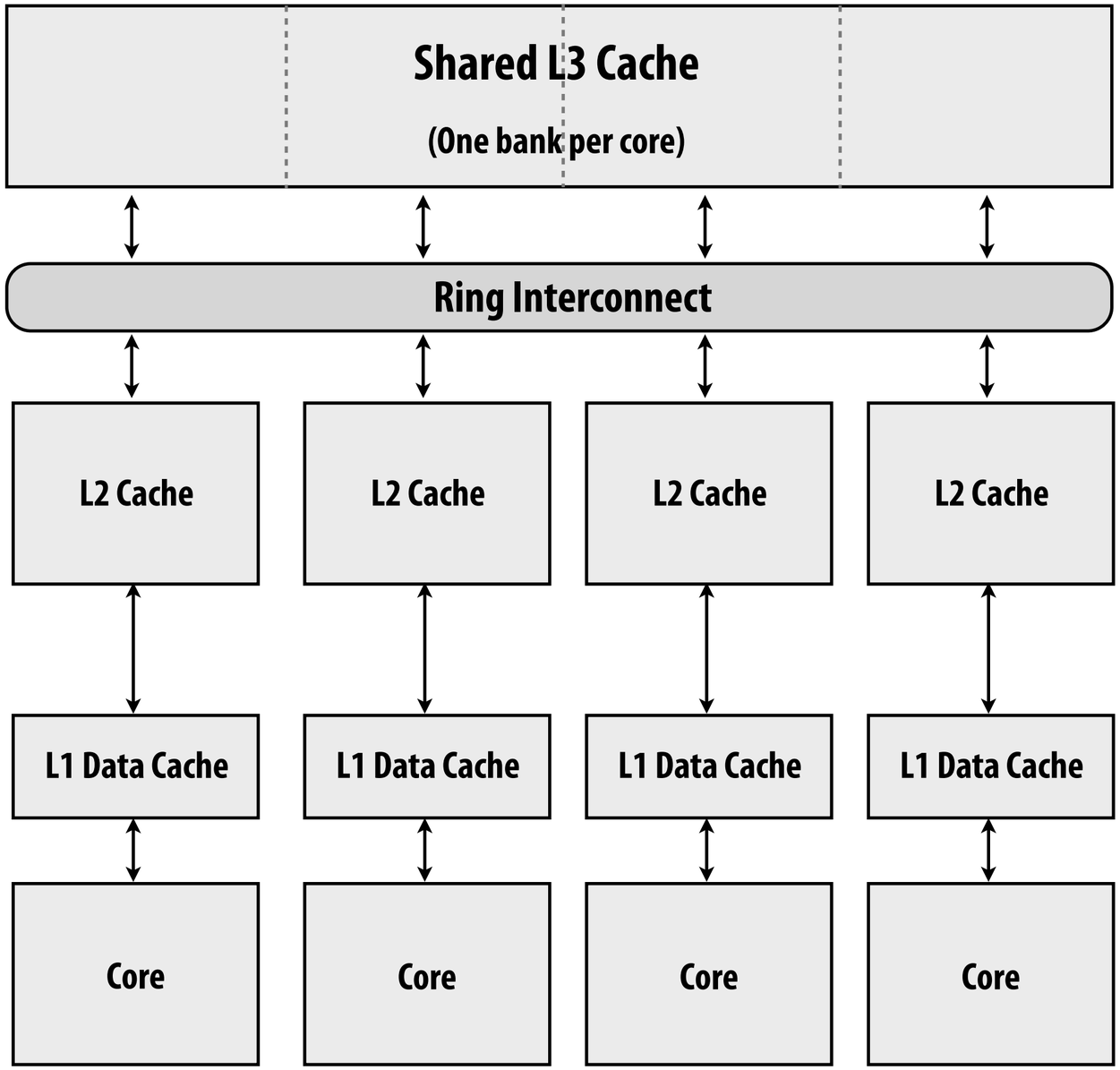


- Update can suffer from high traffic due to multiple writes before the next read by another processor
- Current AMD and Intel implementations of cache coherence are invalidation based

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

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"

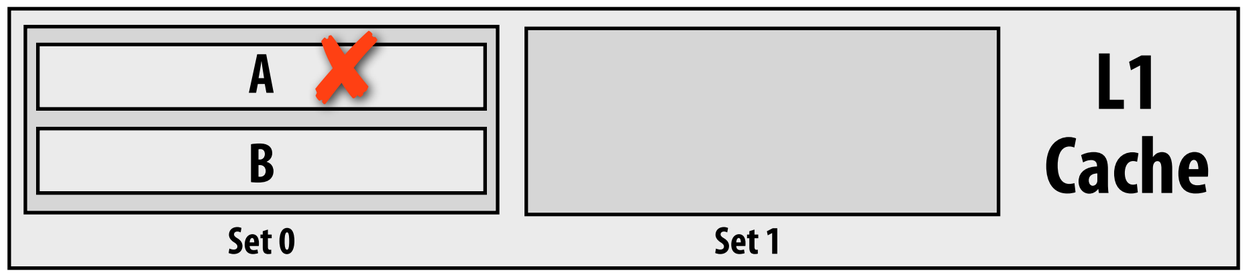
Inclusion property of caches

- All lines in closer [to processor] cache are 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 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 cache line in L1 without requiring information from L1

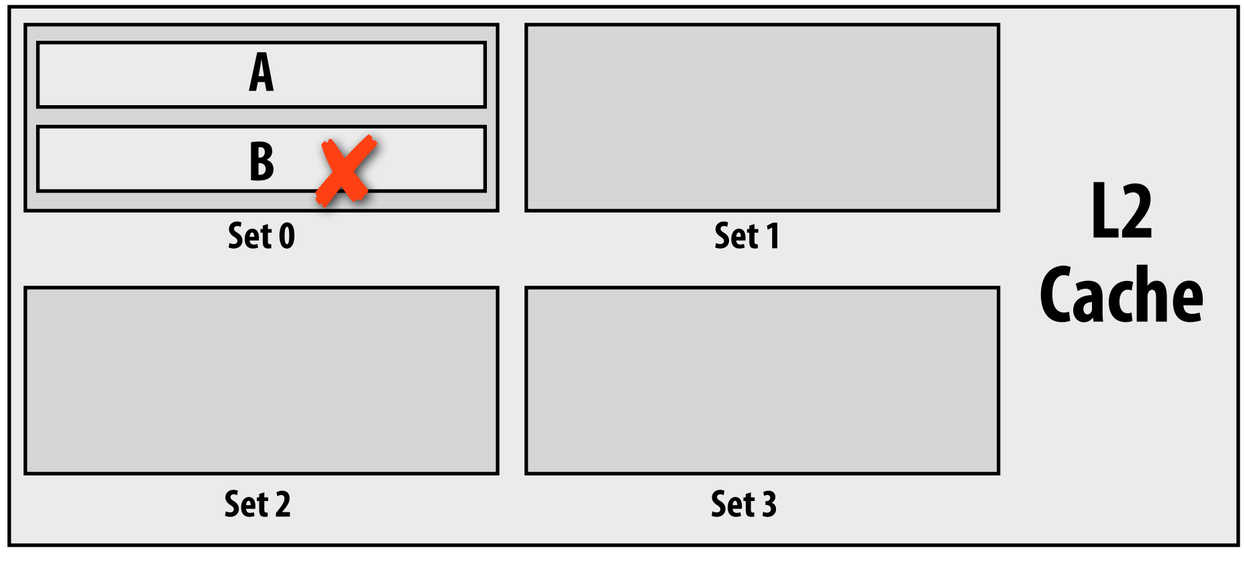
Is inclusion maintained automatically if L2 is larger than L1? No!

- 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).

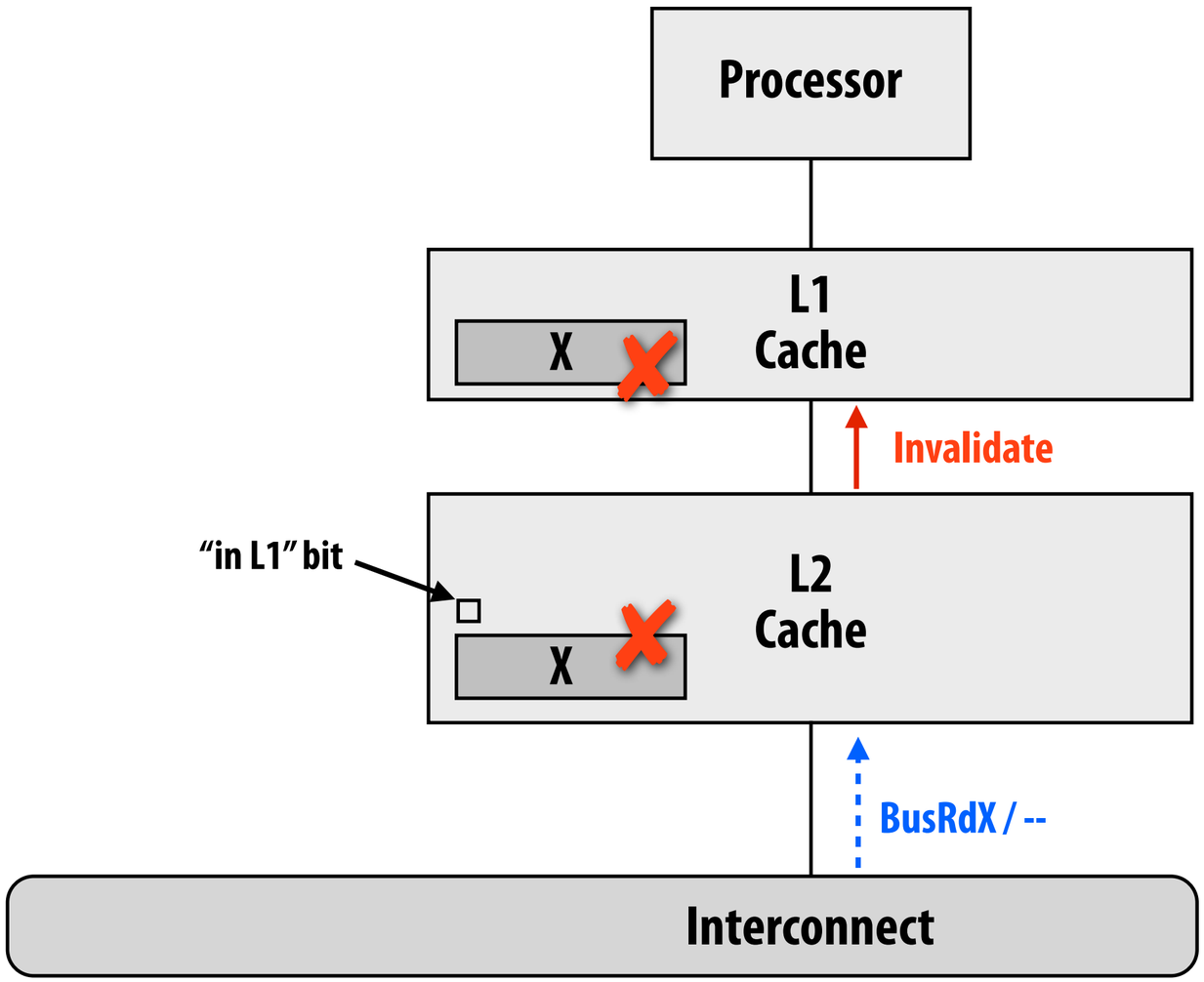


Processor accesses A many times (all L1 hits).

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

As a result, inclusion no longer holds!

Maintaining inclusion: handling invalidations



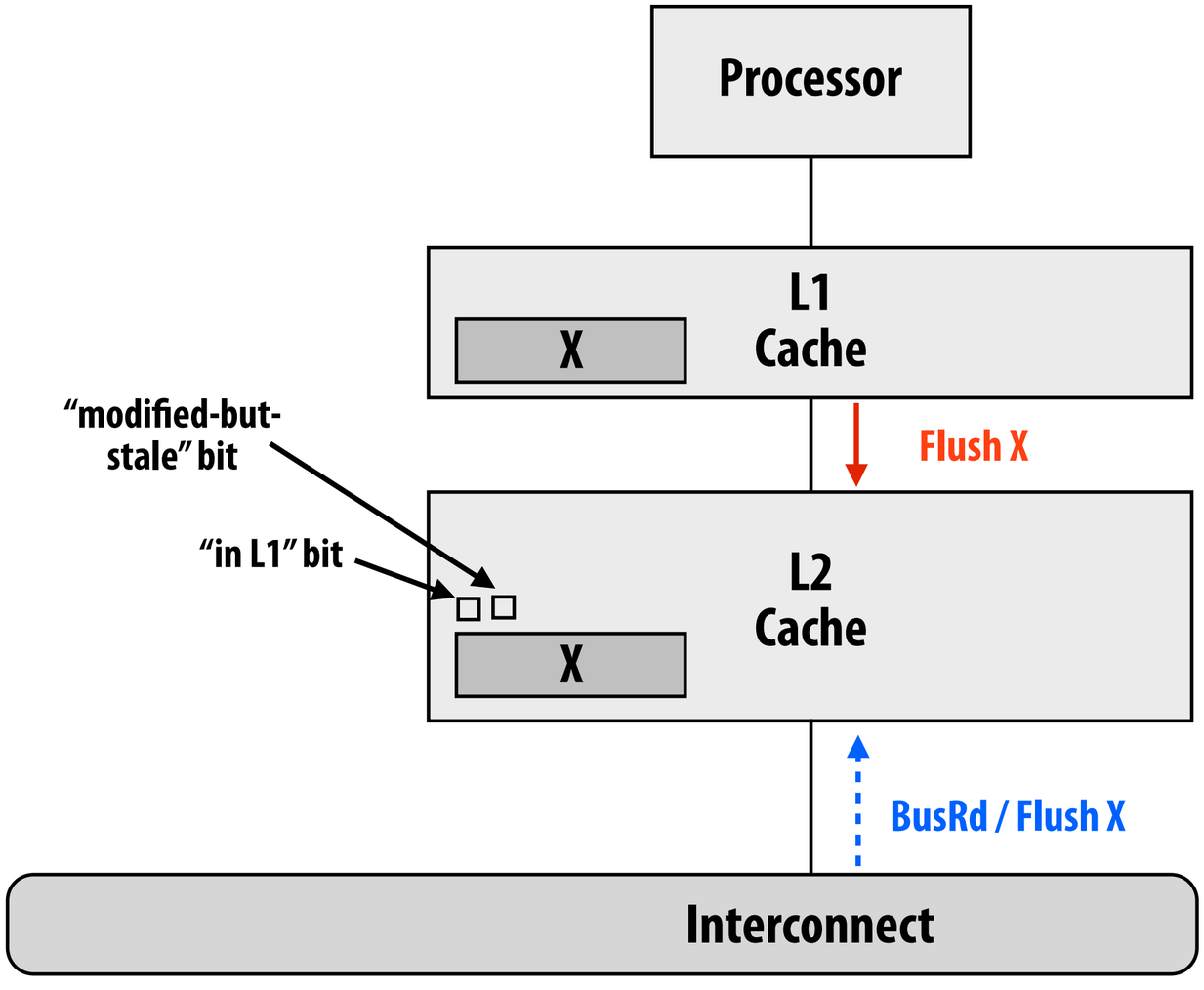
When line X is invalidated in L2 cache due to BusRdX from another cache.

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-state" L2 line requires getting the real data from L1 first.)

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 line sizes...
  - Decrease cold, capacity, true sharing misses
  - Can increase false sharing misses
  - Increase interconnect traffic

- Scalability of snooping implementations is limited by ability to broadcast coherence messages to all cache!
  - 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