Lecture 11:
Snooping Cache Coherence: Part II

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

Announcements

- **Assignment 2 due tonight 11:59 PM**
  - **Recall 3-late day policy**

- **Assignment 3 out tonight**
  - **No rest for the weary**

Bug!

Bug in circleBoxTest.cu_inl

I noticed a painful #bug in `circleBoxTest.cu_inl` that I figured I'd sh

```
areece:render% git diff circleBoxTest.cu_inl
diff --git a/render/circleBoxTest.cu_inl b/render/cir
index 3ea865d..1e01cf3 100644
--- a/render/circleBoxTest.cu_inl
+++ b/render/circleBoxTest.cu_inl
@@ -12,7 +12,7 @@ circleInBoxConservative(
     if ( circleX >= (boxL - circleRadius) &&
          circleX <= (boxR + circleRadius) &&
          circleY >= (boxB - circleRadius) &&
-         circleY >= (boxT + circleRadius) ) {
+         circleY <= (boxT + circleRadius) ) {
         return 1;
     } else {
         return 0;
```

Review: false sharing

What could go wrong with this code?

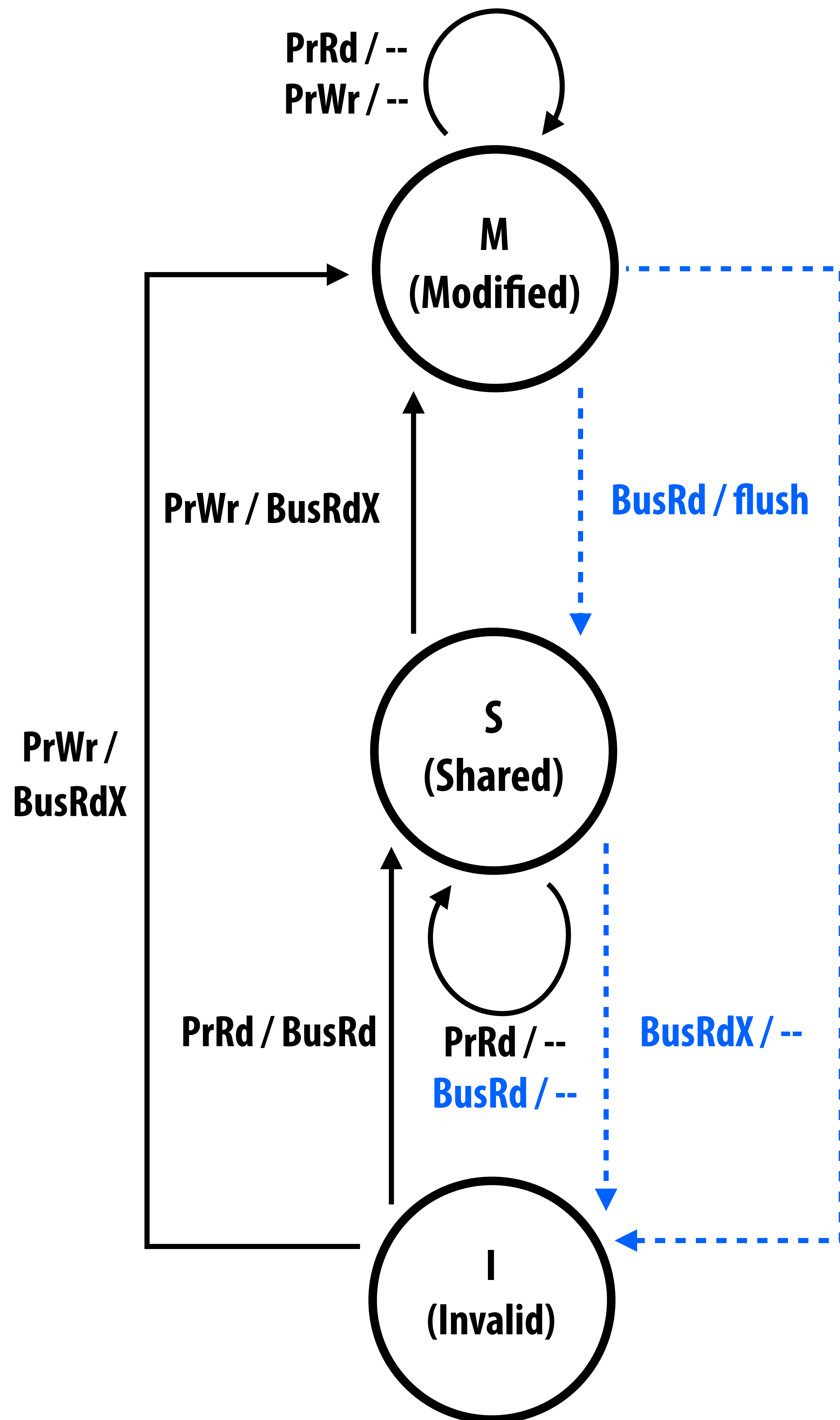
```
// allocate per-thread variable for local accumulation
int myCounter[NUM_THREADS];
```

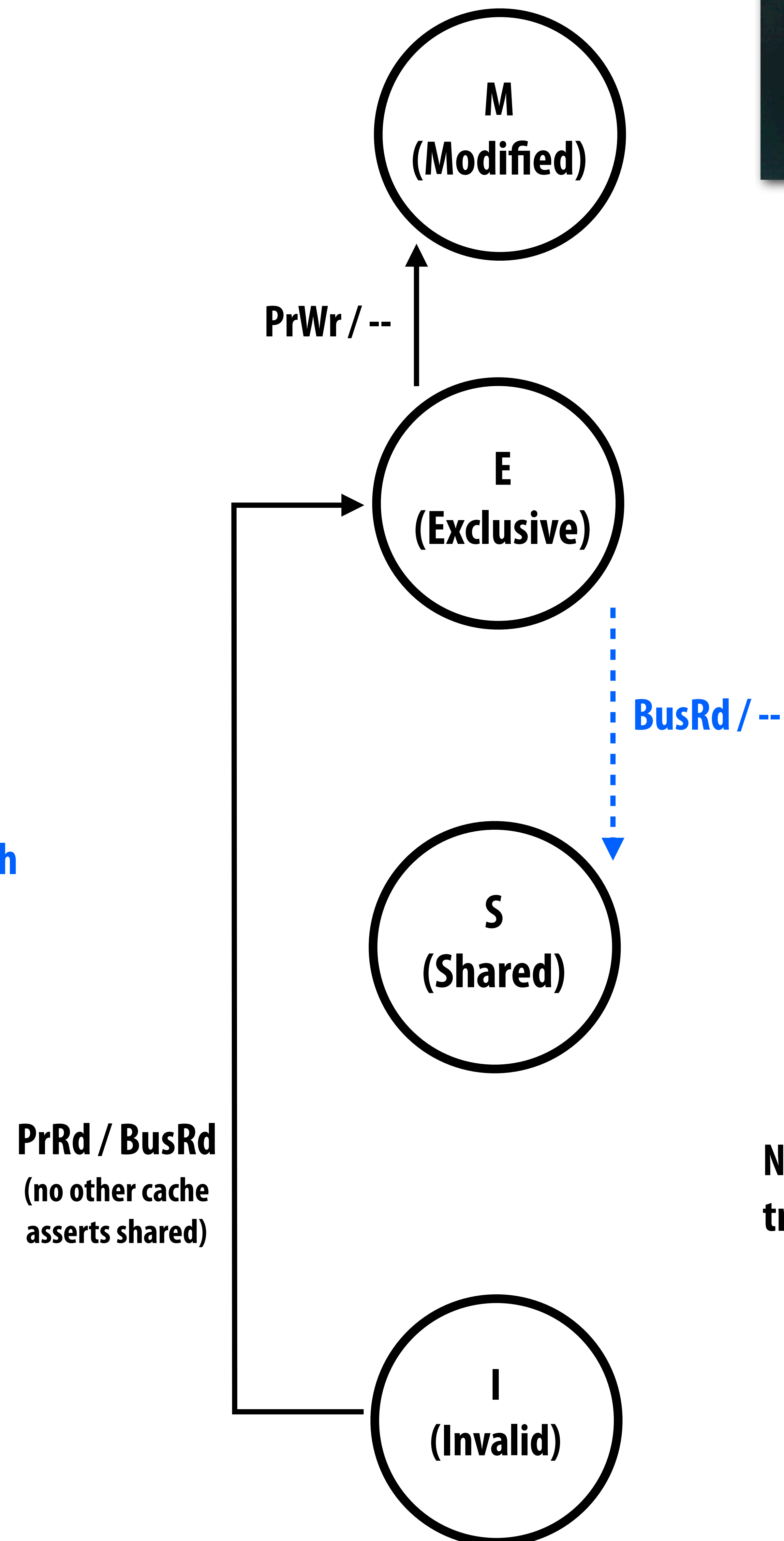
Better:

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

Review: MSI and MESI

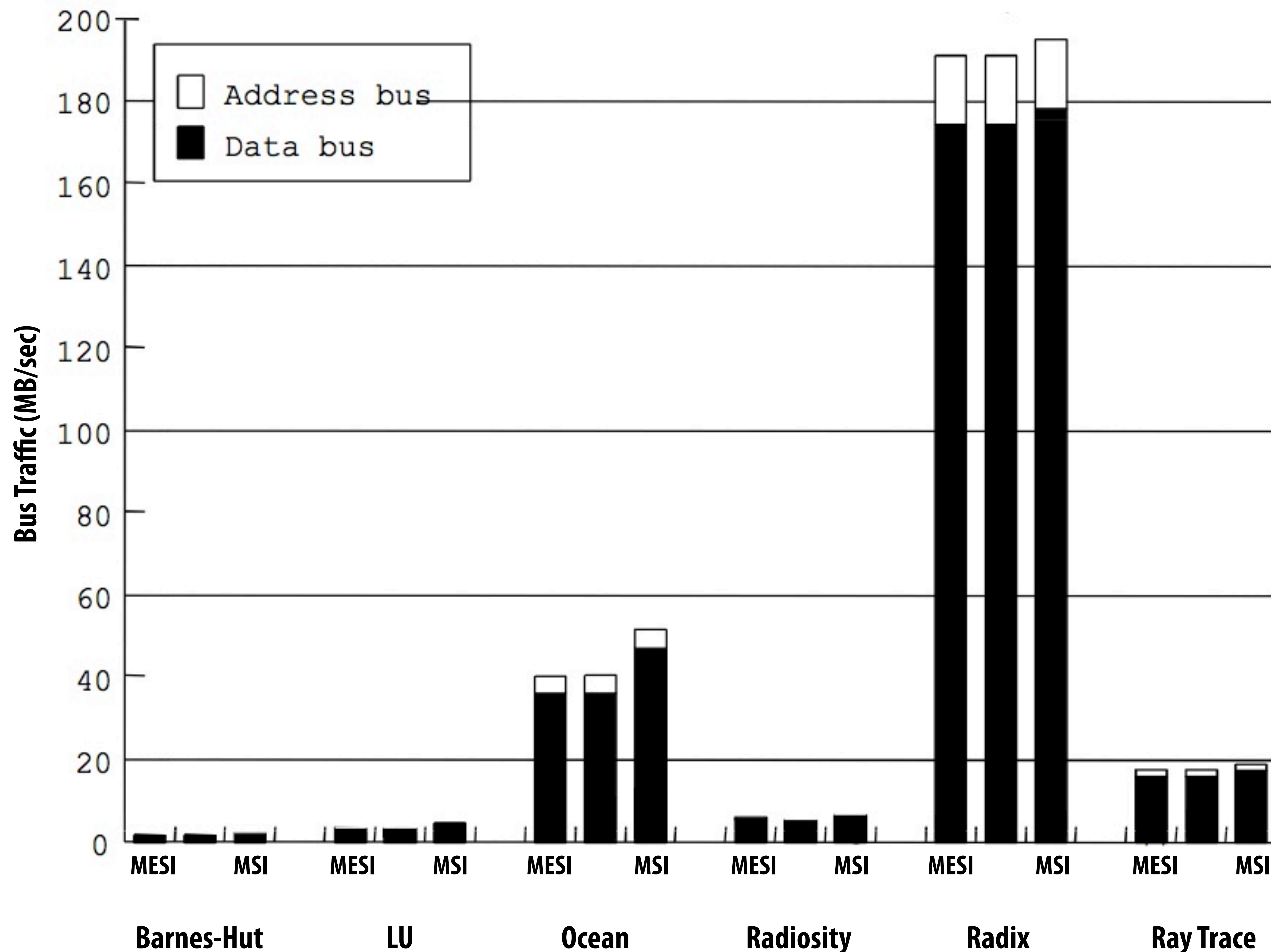
MESI, not Messi !





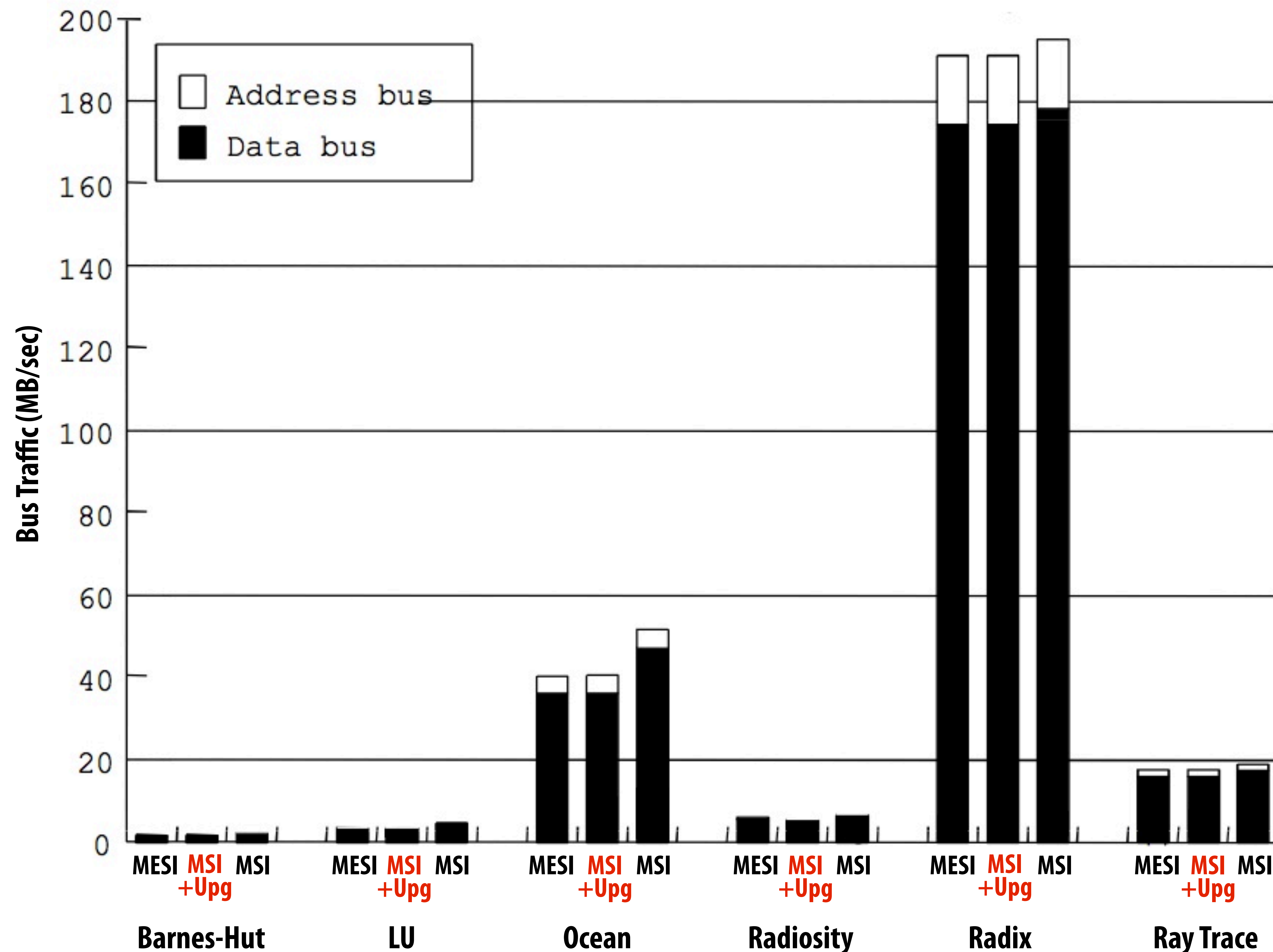
Note: only showing transitions unique to MESI

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, rather than BusRdX, is used (no need to transfer a cache line, just broadcast "upgrade")

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

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

Today's topics

- **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

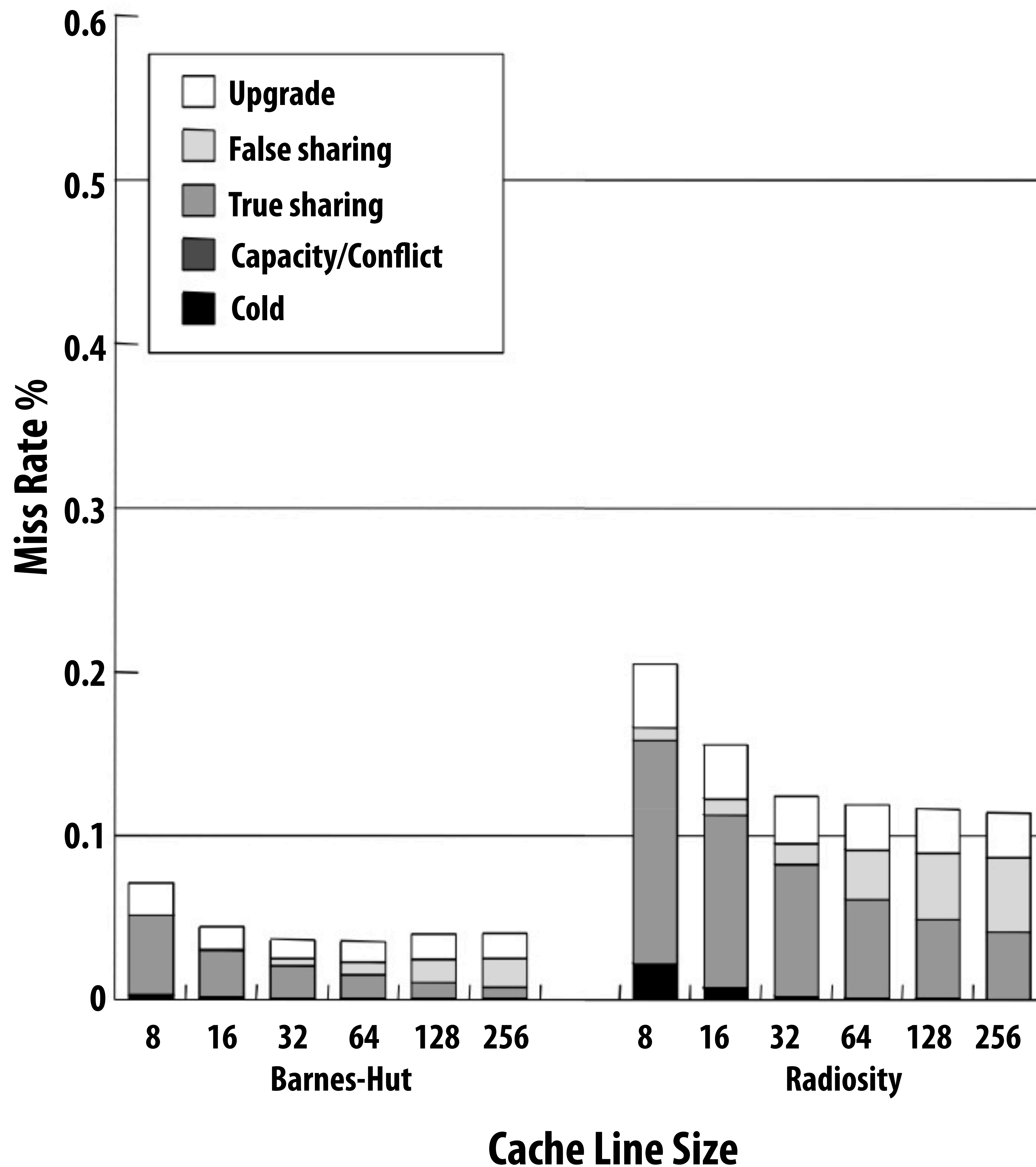
- Recall that cache coherence adds a fourth type of miss: coherence misses

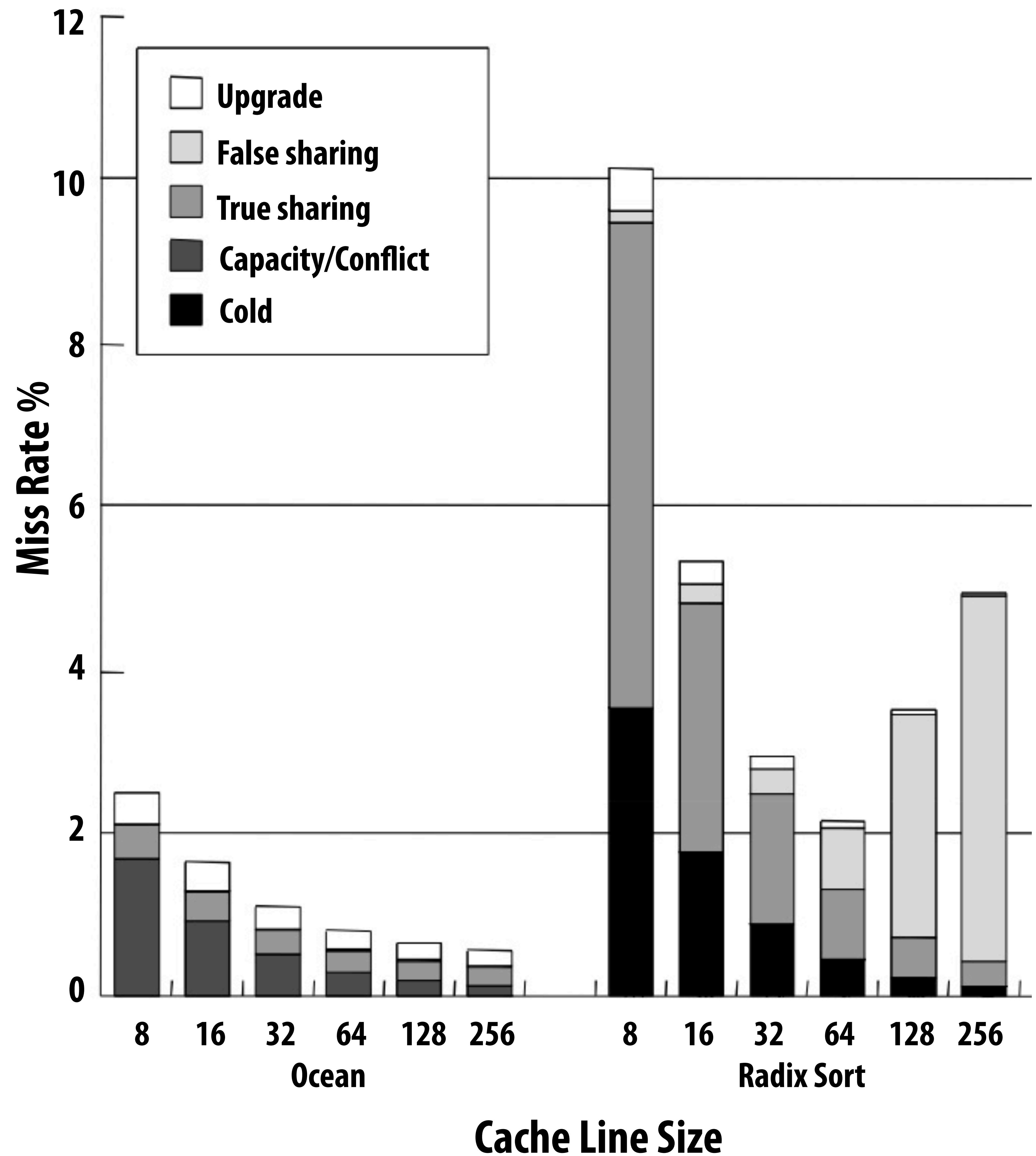
- How to reduce cache misses:
  - Capacity miss: enlarge cache, increase block size
  - Conflict miss: increase associativity
  - Cold/true sharing coherence: increase block size

- 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

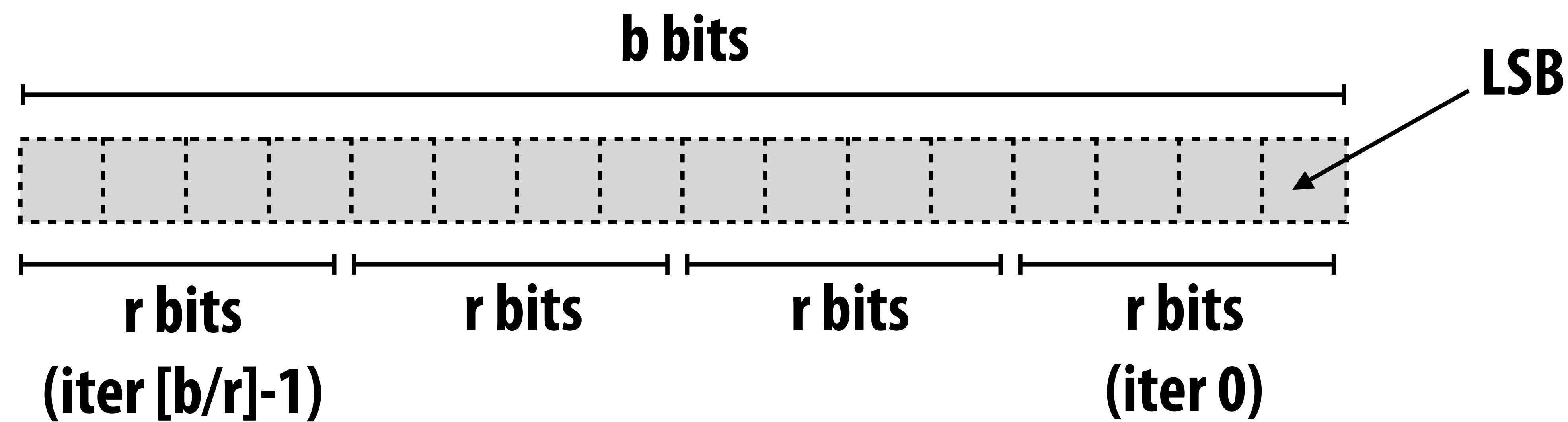


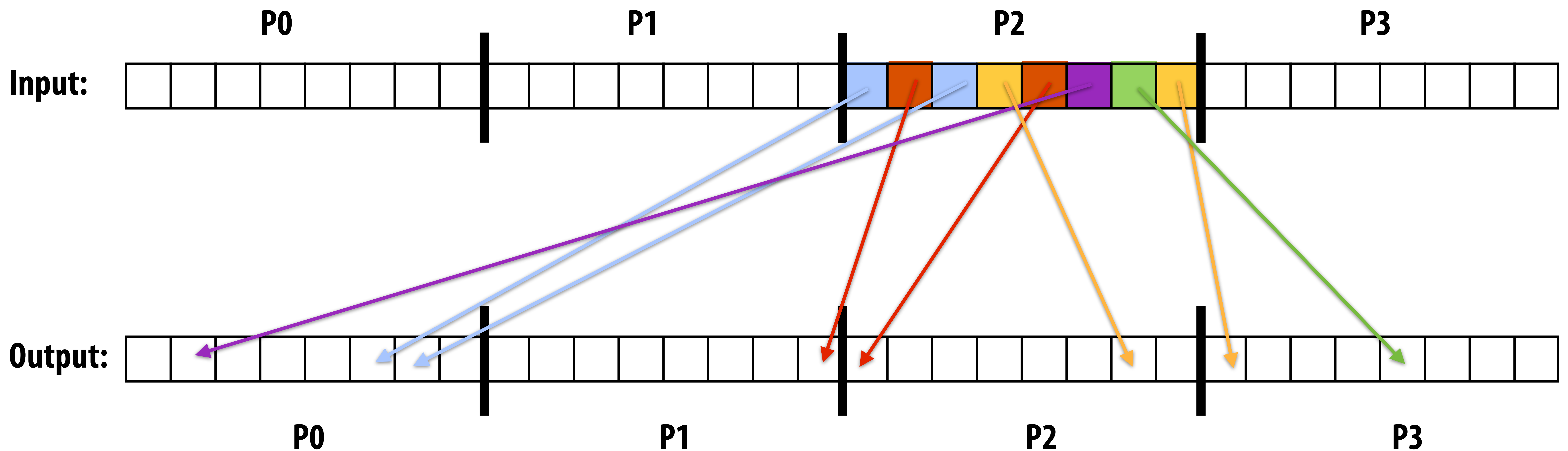


Example: parallel radix sort

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)

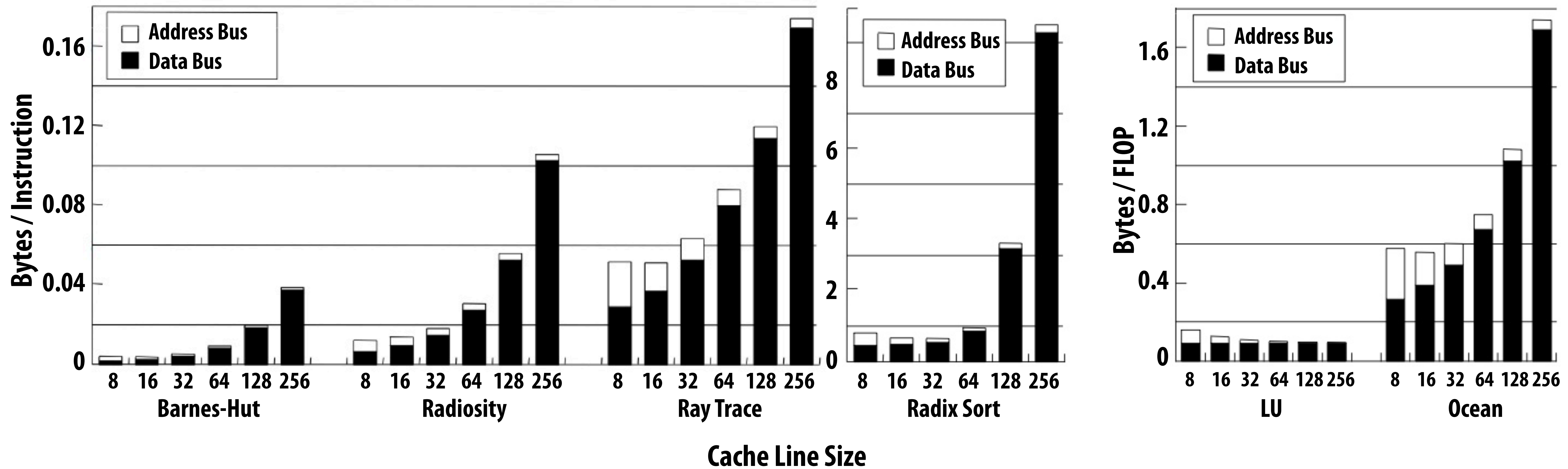




Potential for lots of false sharing
False sharing decreases with increasing array size

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

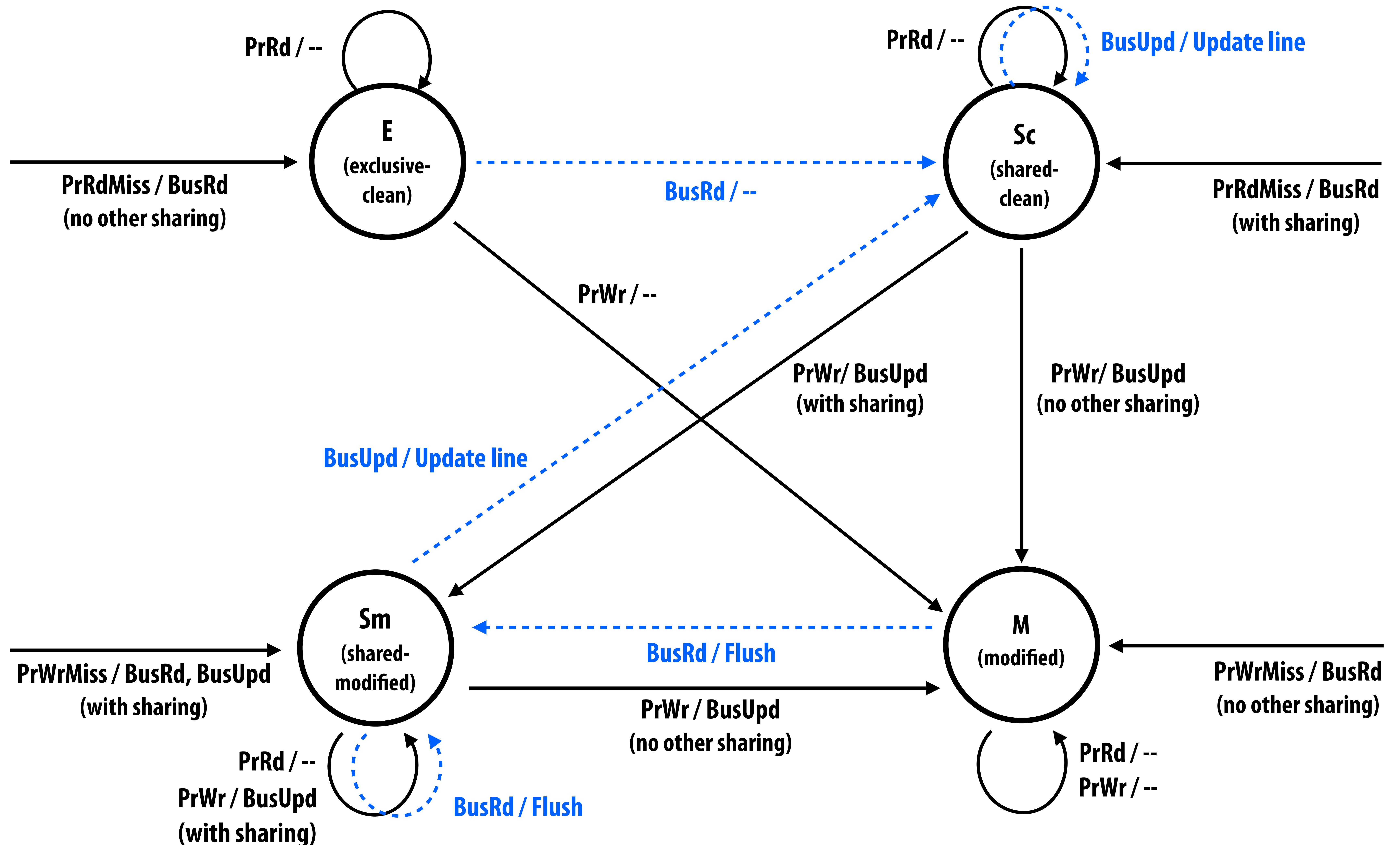
- Thus far, we've talking only about invalidation-based protocols
  - Main idea: cache obtains exclusive access to line in order to write to it
  - Possible issues:
    - Cache must reload entire line after invalidation
    - False sharing

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

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 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 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), bus write back (BusWB), bus update (BusUpd)

Dragon write-back update protocol



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

(CMU 15-418, Spring 2012)

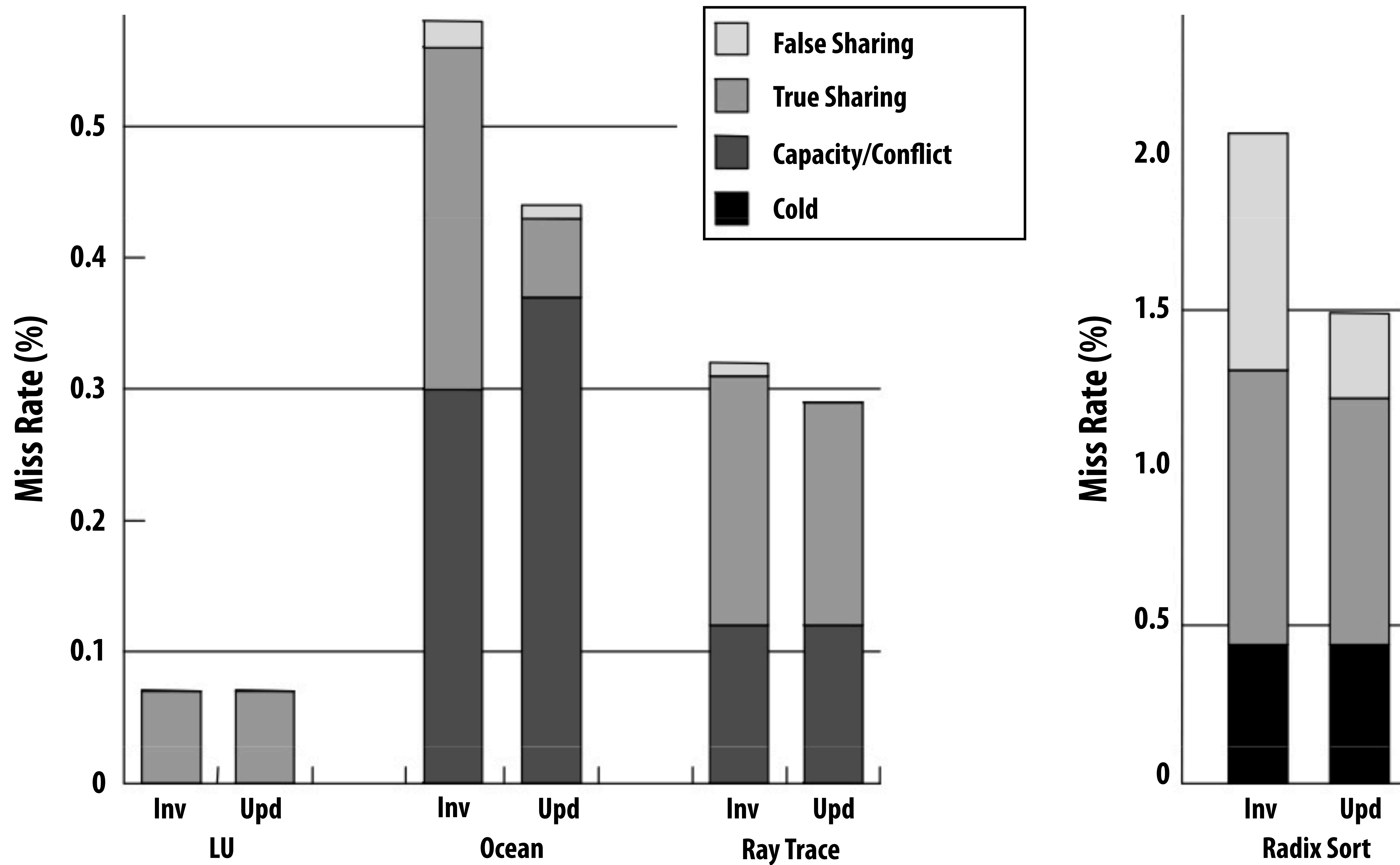
Invalidate vs. update

- 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

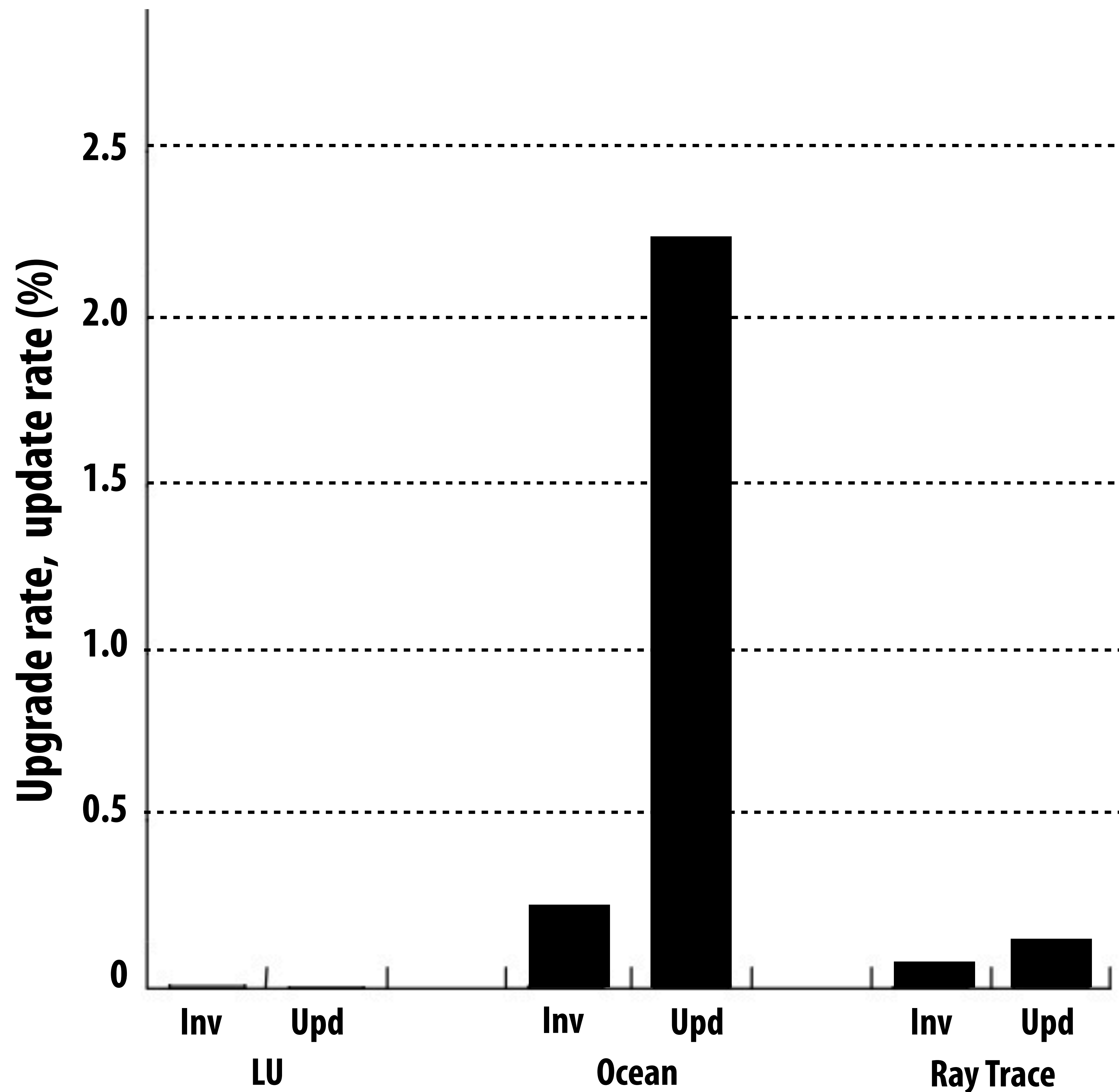
Invalidate vs. update evaluation: miss rate

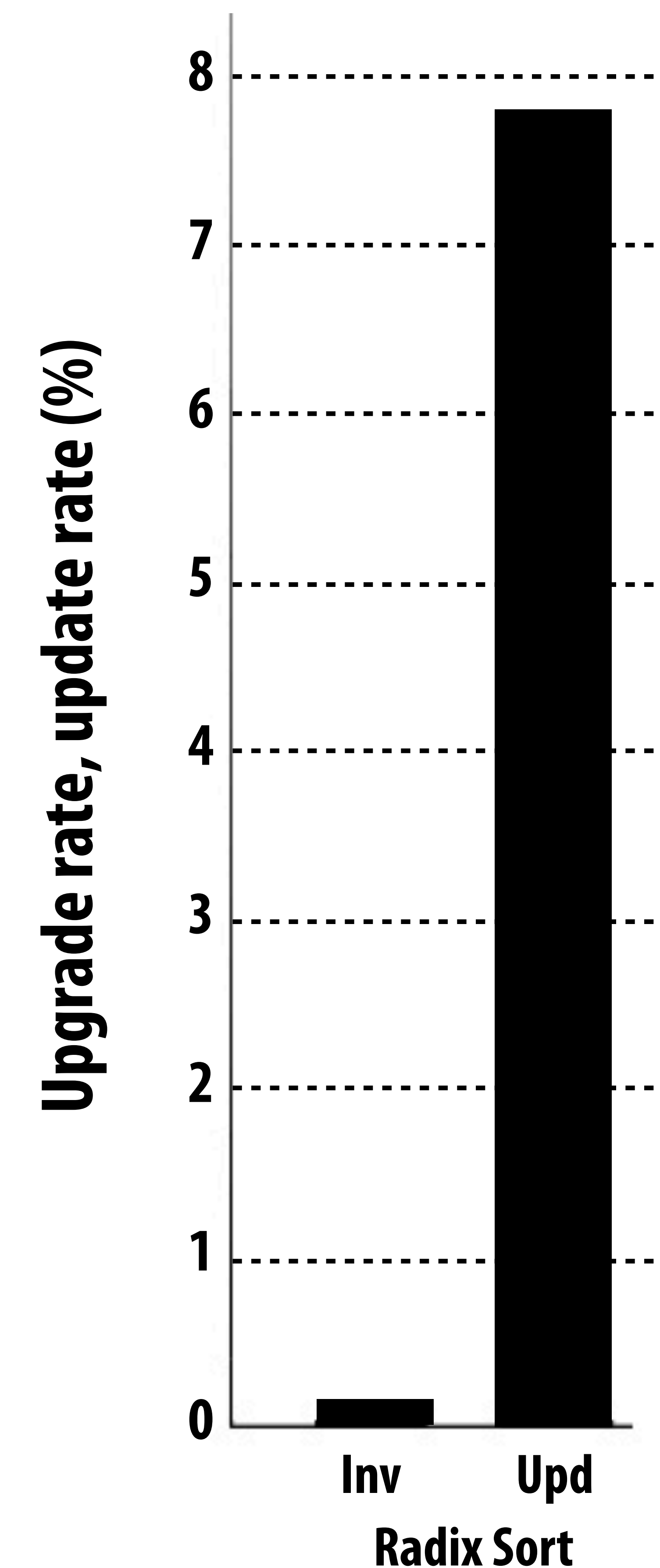
Simulated 1 MB cache, 64 B lines



Invalidate vs. update evaluation: traffic

Simulated 1 MB cache, 64 B lines

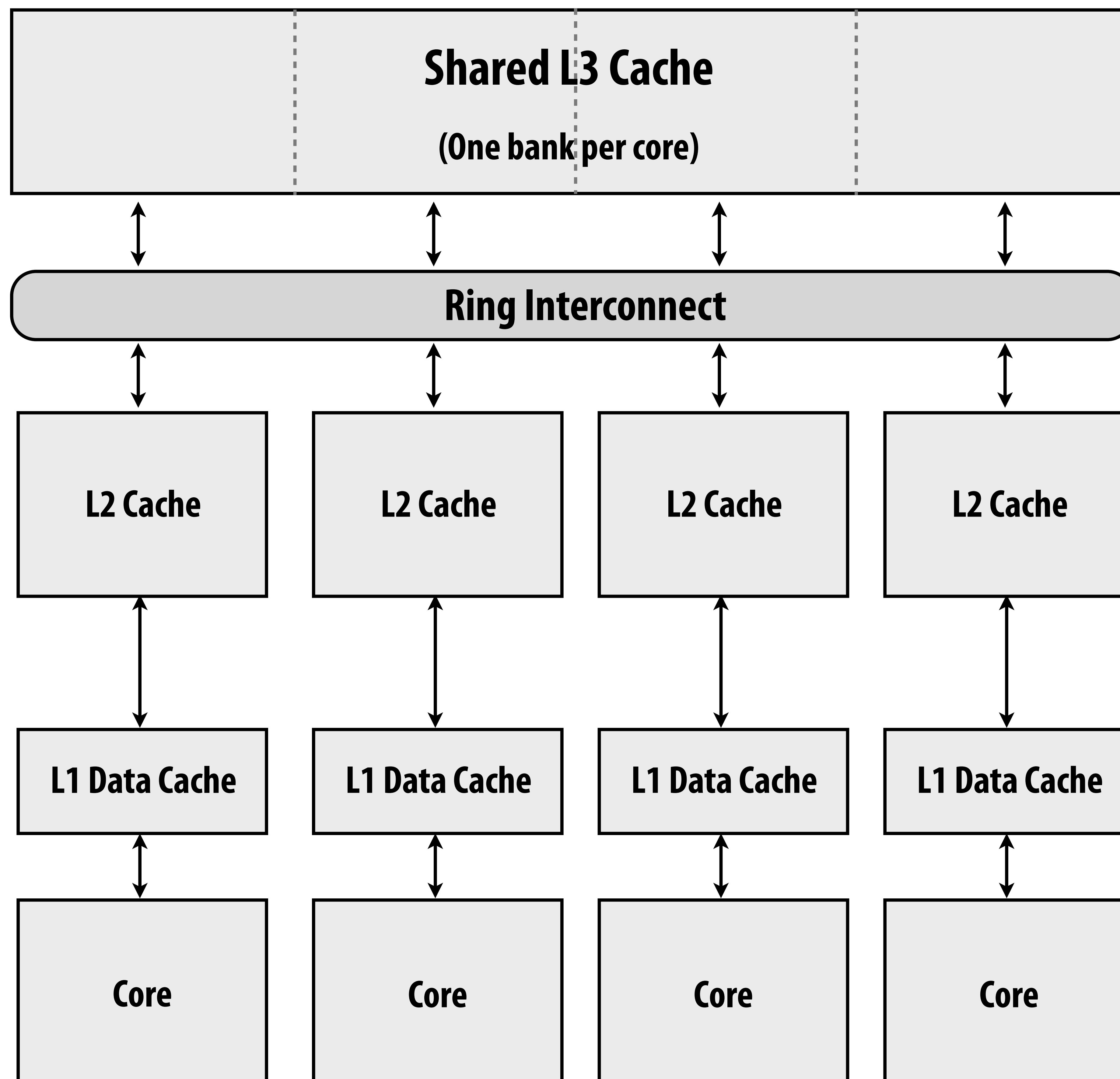




- Update can suffer from high traffic due to multiple writes before the next read by another processor
- Current AMD and Intel implementations of cache coherence are invalidation based

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"**

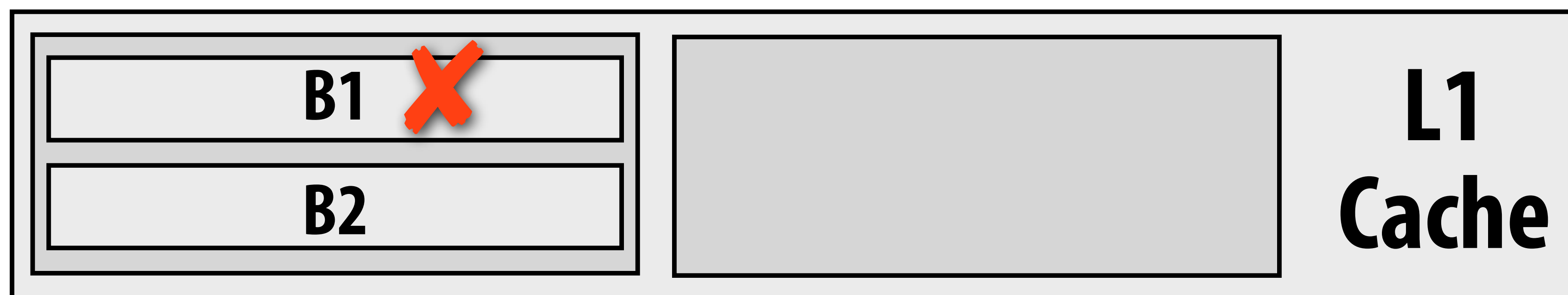
Inclusion property of caches

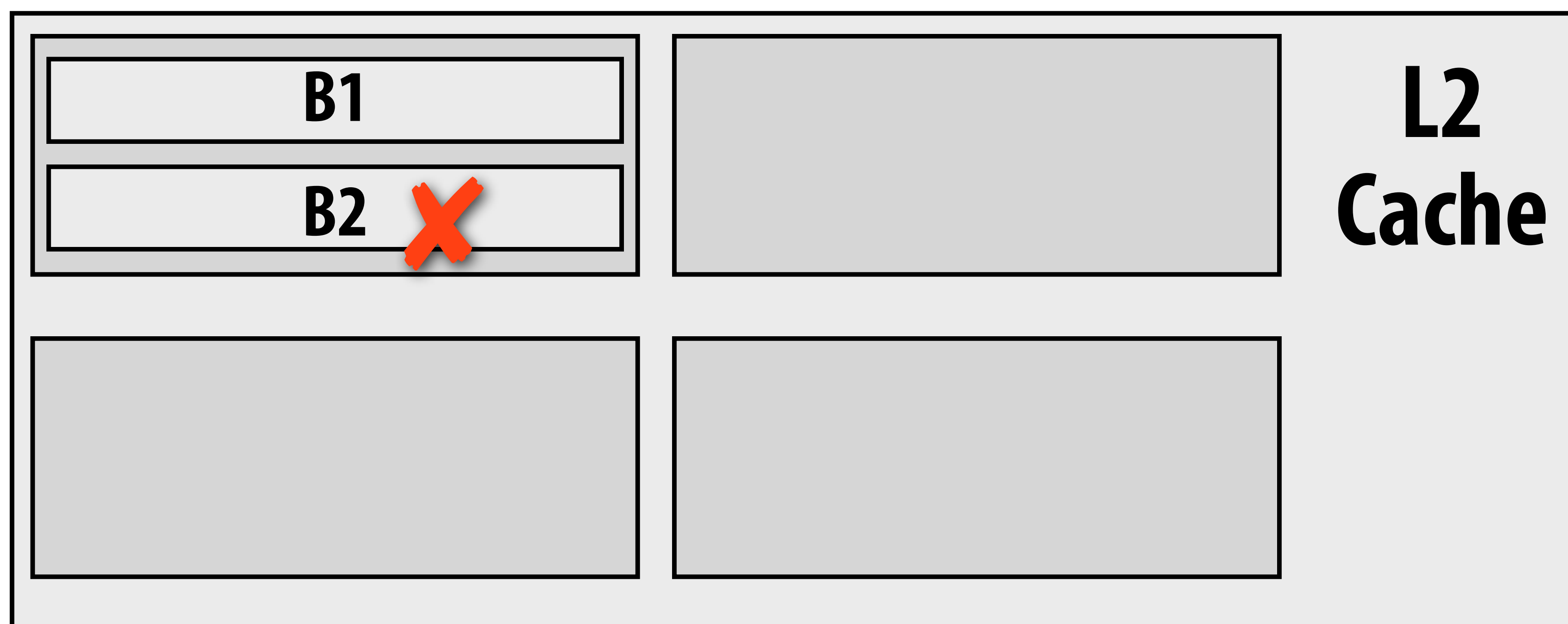
- All lines in closer [to processor] cache are in farther cache
  - e.g., contents of L1 are a subset of contents of L2
  - Thus, all transactions relevant to L1 are also relevant to L2, so it is sufficient for only the L2 to snoop the interconnect

- 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





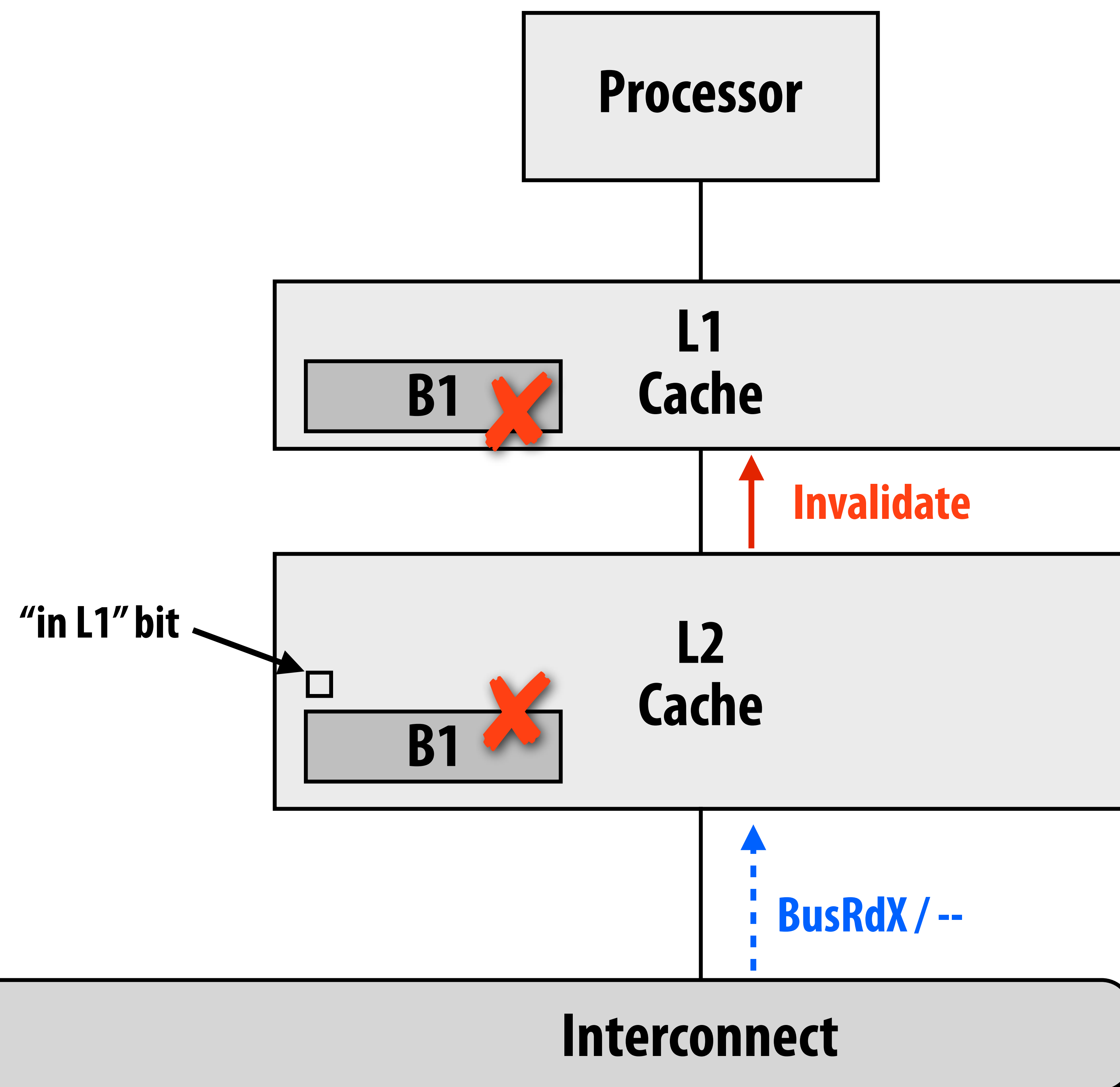
Processor references to B1 and B2 are serviced by L1 cache. The access history to B1 and B2 are different in the L1 than in the L2!

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



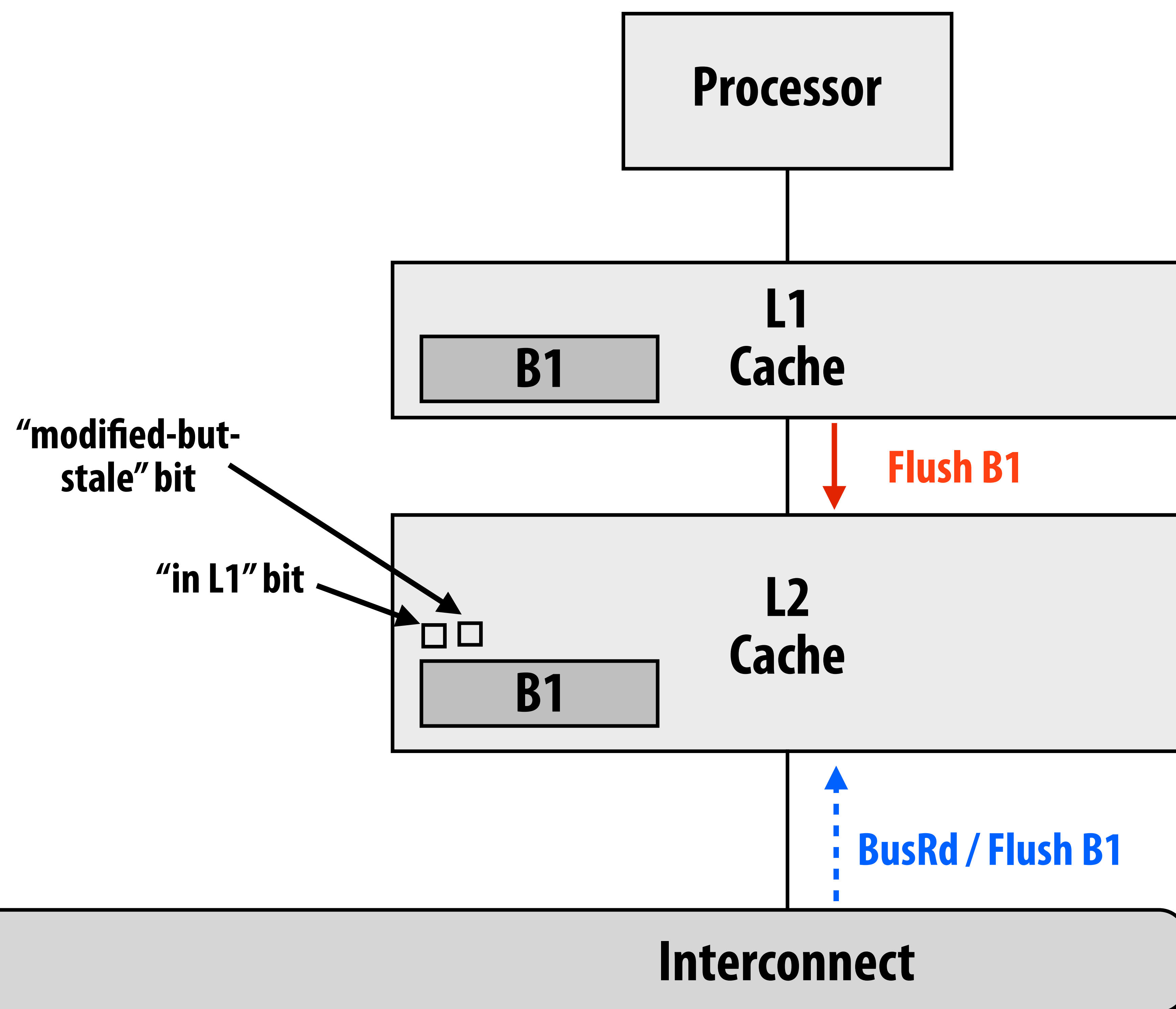
Block invalidated in L2 cache due to BusRdX from another cache.

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