Lecture 11:

Cache Coherence: Part II

Parallel Computer Architecture and Programming
CMU 15-418/15-618, Spring 2015

Tunes

Bang Bang
(My Baby Shot Me Down)

Nancy Sinatra

(Kill Bill Volume 1 Soundtrack)

"It's such a heartbreaking thing... to have your performance silently crippled by cache lines dropping all over the place due to false sharing."
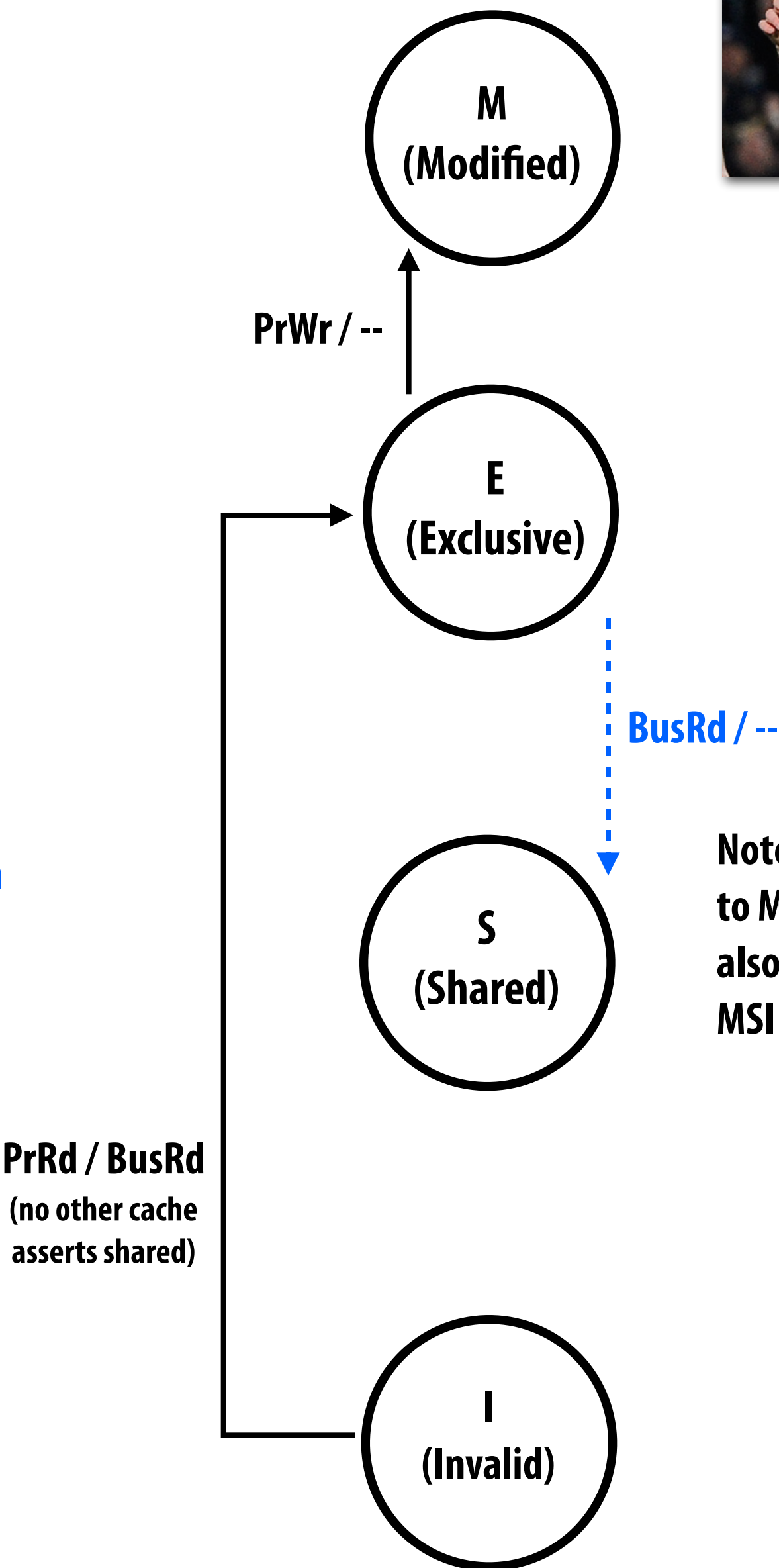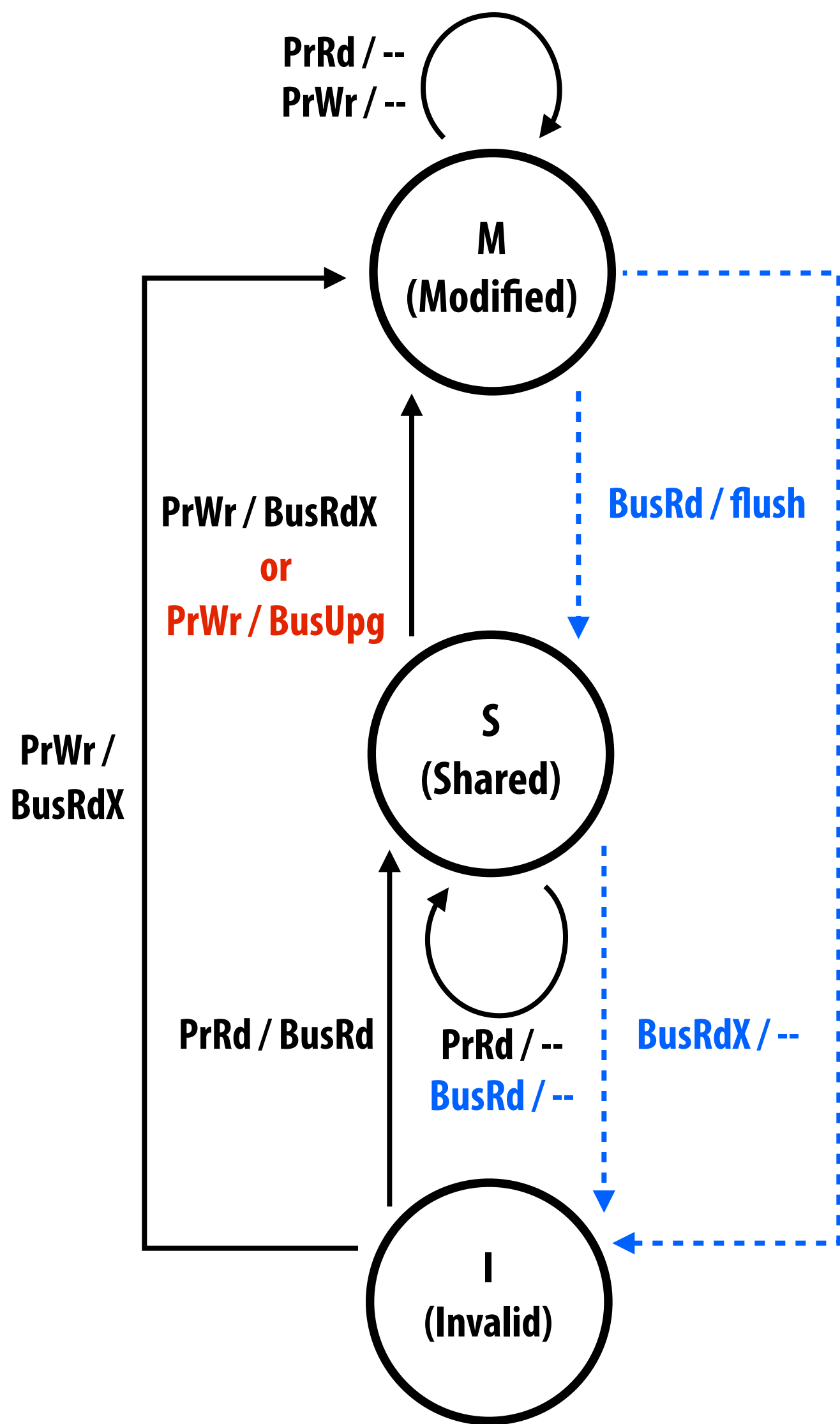
- Nancy Sinatra

Review: two big ideas from last class

- Snooping-based coherence: every time a cache must a change to the state of a line that make affect coherence, it notifies all other caches before proceeding.

- Exclusive ownership in an invalidation-based coherence protocol: in order to write to an address, cache first guarantees that no other cache has a copy of the data.
    - It does so by waiting to "hear" that any cache with the data has "dropped the line".

Demo: false sharing
(on six-core CPUs in lab)

Review: MSI and MESI

MESI, not Messi!

MSI (left diagram)

M (Modified)
- PrRd / --
- PrWr / --

S (Shared)
- PrRd / --
- BusRd / --

I (Invalid)

Transitions:
- PrWr / BusRdX **or** PrWr / BusUpg
- PrWr / BusRdX
- PrRd / BusRd
- BusRd / flush
- BusRdX / flush
- BusRdX / --

MESI (right diagram)

M (Modified)

E (Exclusive)
- PrWr / --

S (Shared)
- BusRd / --

I (Invalid)
- PrRd / BusRd (no other cache asserts shared)

Note: only showing transitions unique to MESI on this diagram (MESI protocol also contains all the transitions in the MSI diagram at left)

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 can 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 is present)

- **How can larger cache line size hurt? (assume: fixed-size cache)**
  - Increases cost of a miss (larger line to load into cache may increase miss latency)
  - Can increase misses due to conflicts (fewer lines in cache)
  - Can increase misses due to false sharing

Impact of cache line size on miss rate

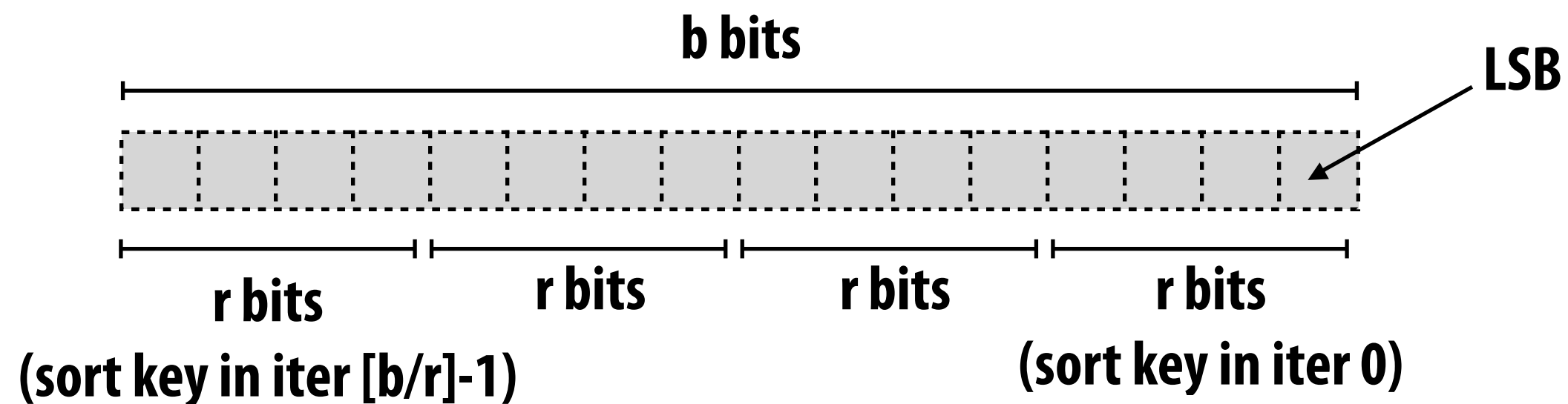**Results from simulation of a 1 MB cache (four example applications)**
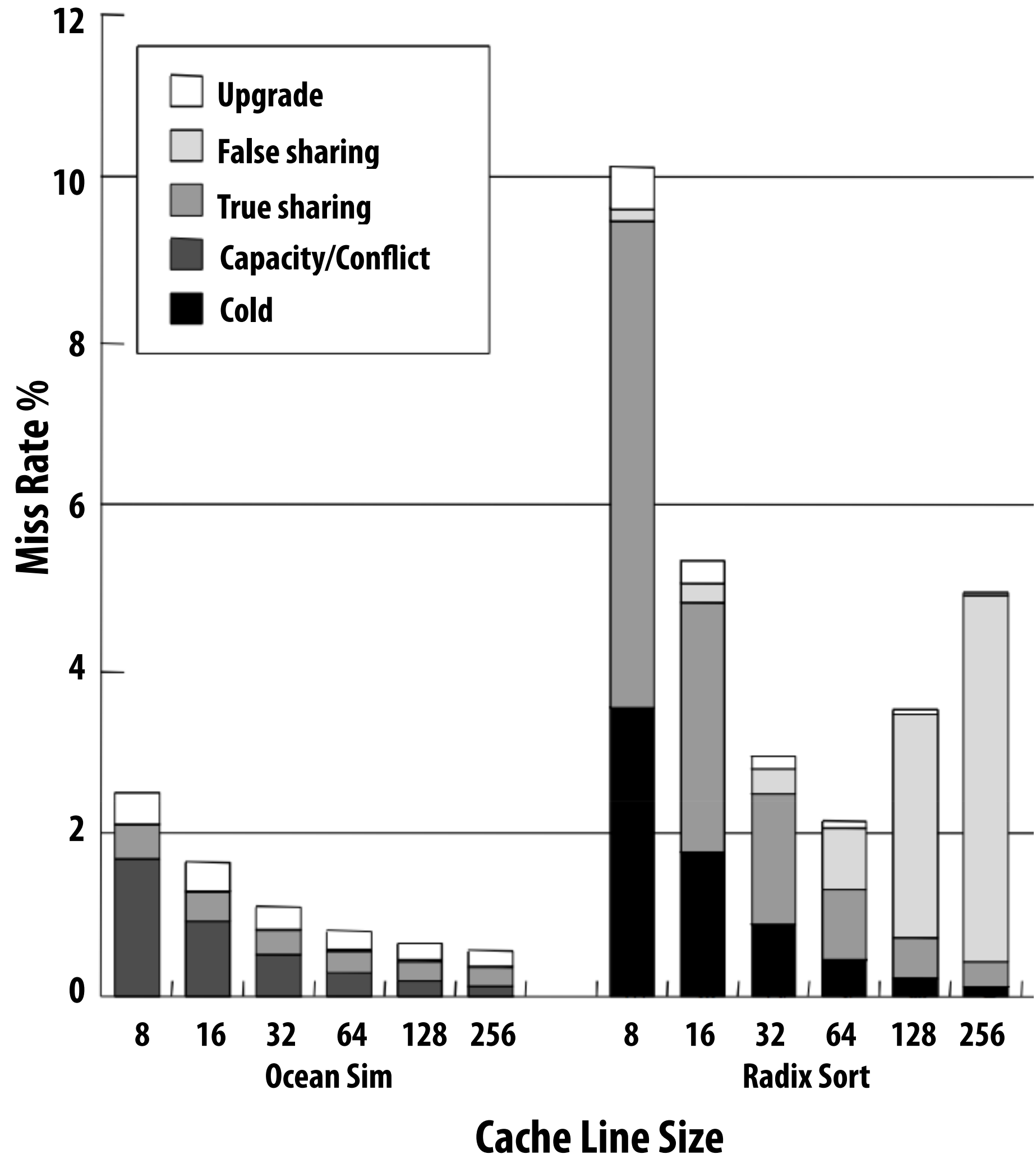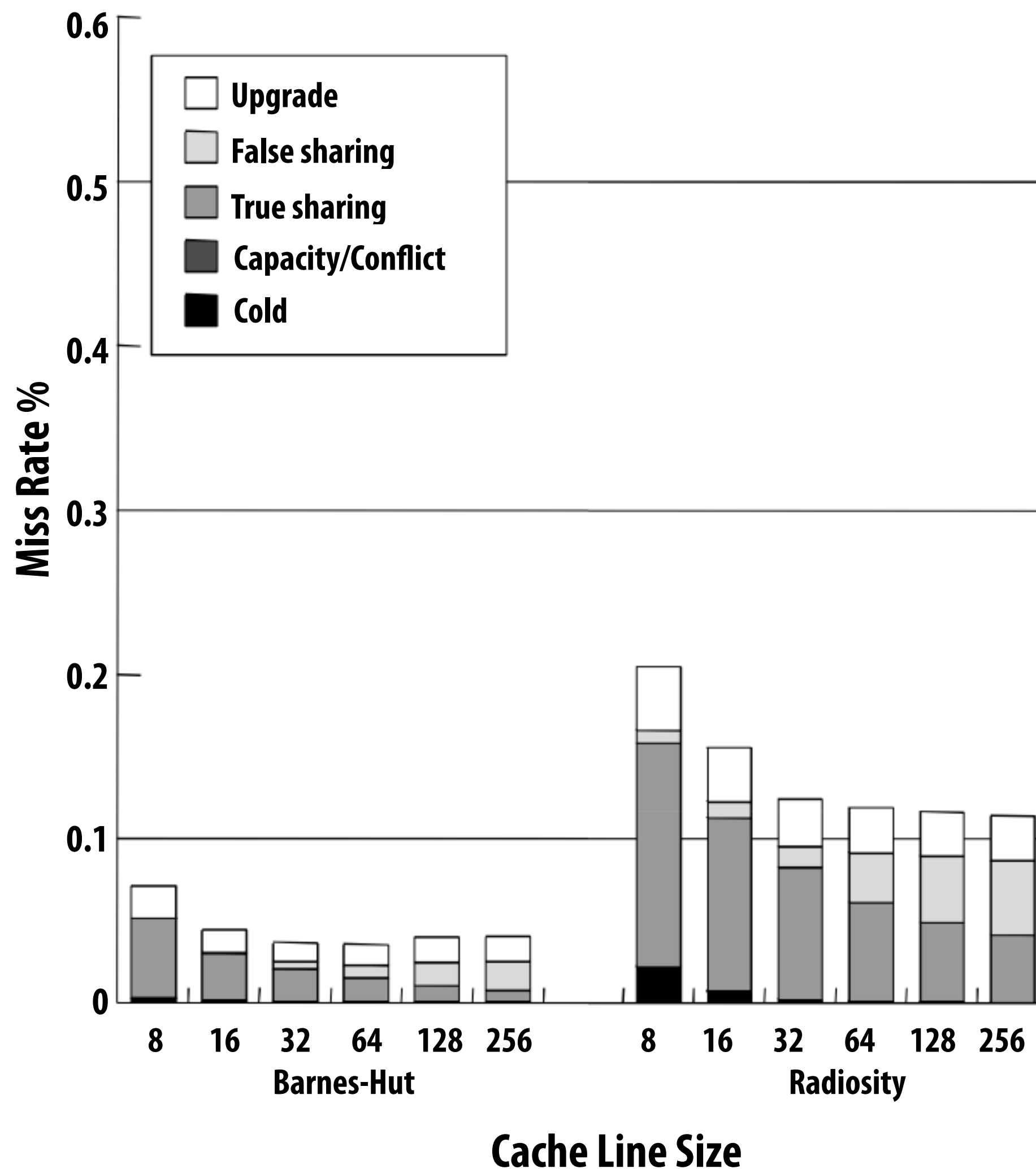
* **Note: I separated the results into two graphs because of different Y-axis scales**
Figure credit: Culler, Singh, and Gupta

Example: parallel radix sort of b-bit numbers

Sort array of N, b-bit numbers
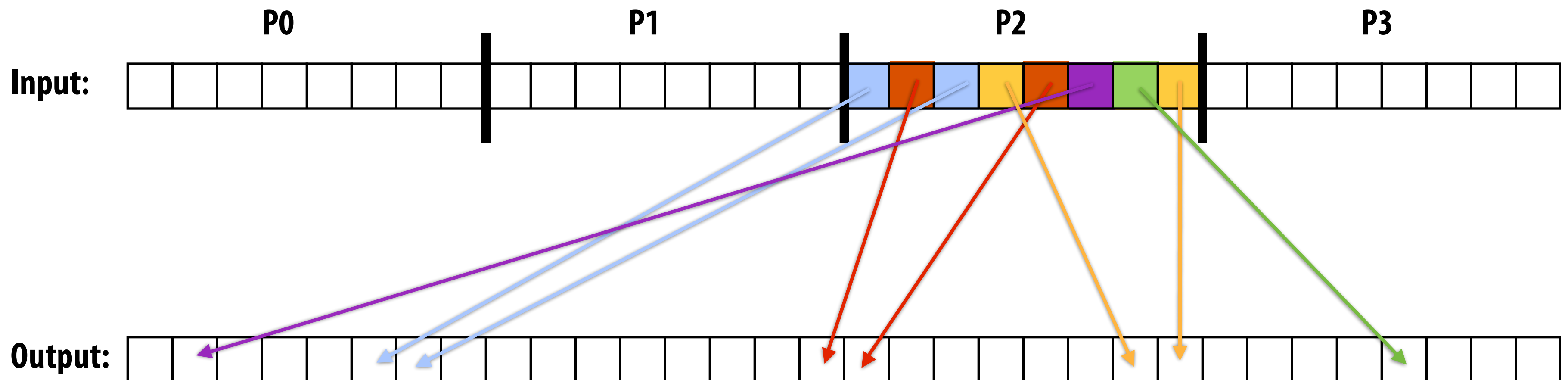Here: radix = $2^4 = 16$

For each group of *r* bits: (serial loop)
In parallel, on each processor:
Sort elements by r-bit group value
Compute number of elements in each bin
Aggregate per-processor counts to compute
compute bin starts
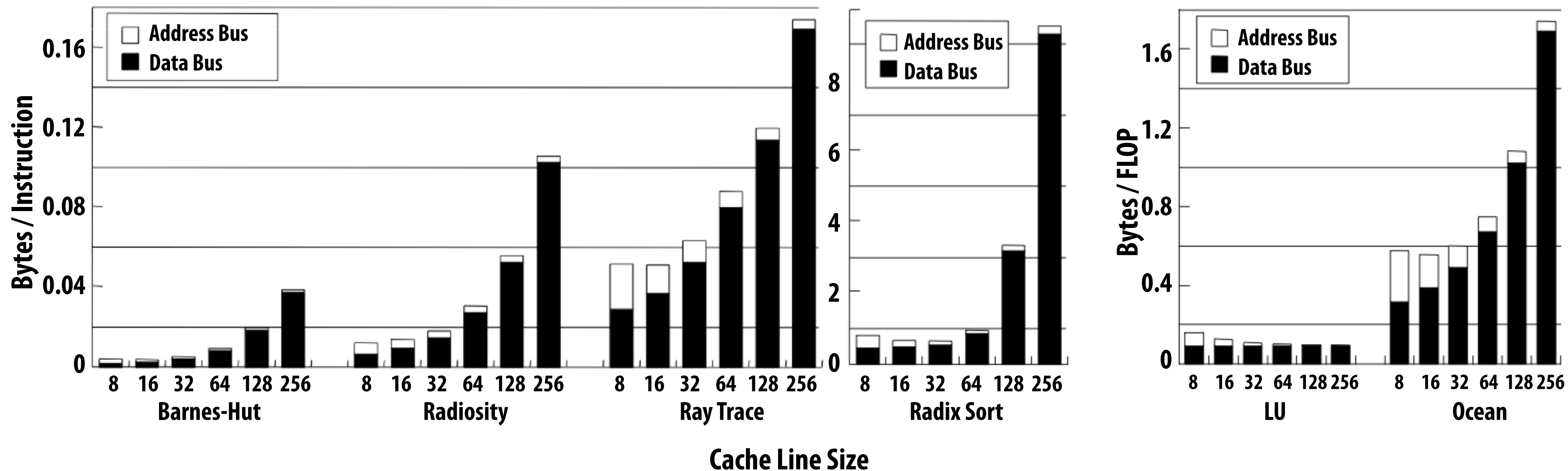Write elements to appropriate position

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

Impact of cache line size on interconnect traffic

Results from simulation of a 1 MB cache

Figure credit: Culler, Singh, and Gupta

Some thoughts

- **In general, larger cache lines:**

  - **Fewer misses**

  - **But more traffic (unless spatial locality is perfect -- see LU example on prev slide)**

- **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)**

- **If you were building a system, you would simulate many important apps and make decisions based on your graphs and performance 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 <u>exclusive</u> 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**
  - **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)

  - <u>Exclusive-clean</u> (E): only one cache has line, memory up-to-date

  - <u>Shared-clean</u> (SC): multiple caches may have line, and memory <u>may or may not</u> ** be up to date

  - <u>Shared-modified</u> (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

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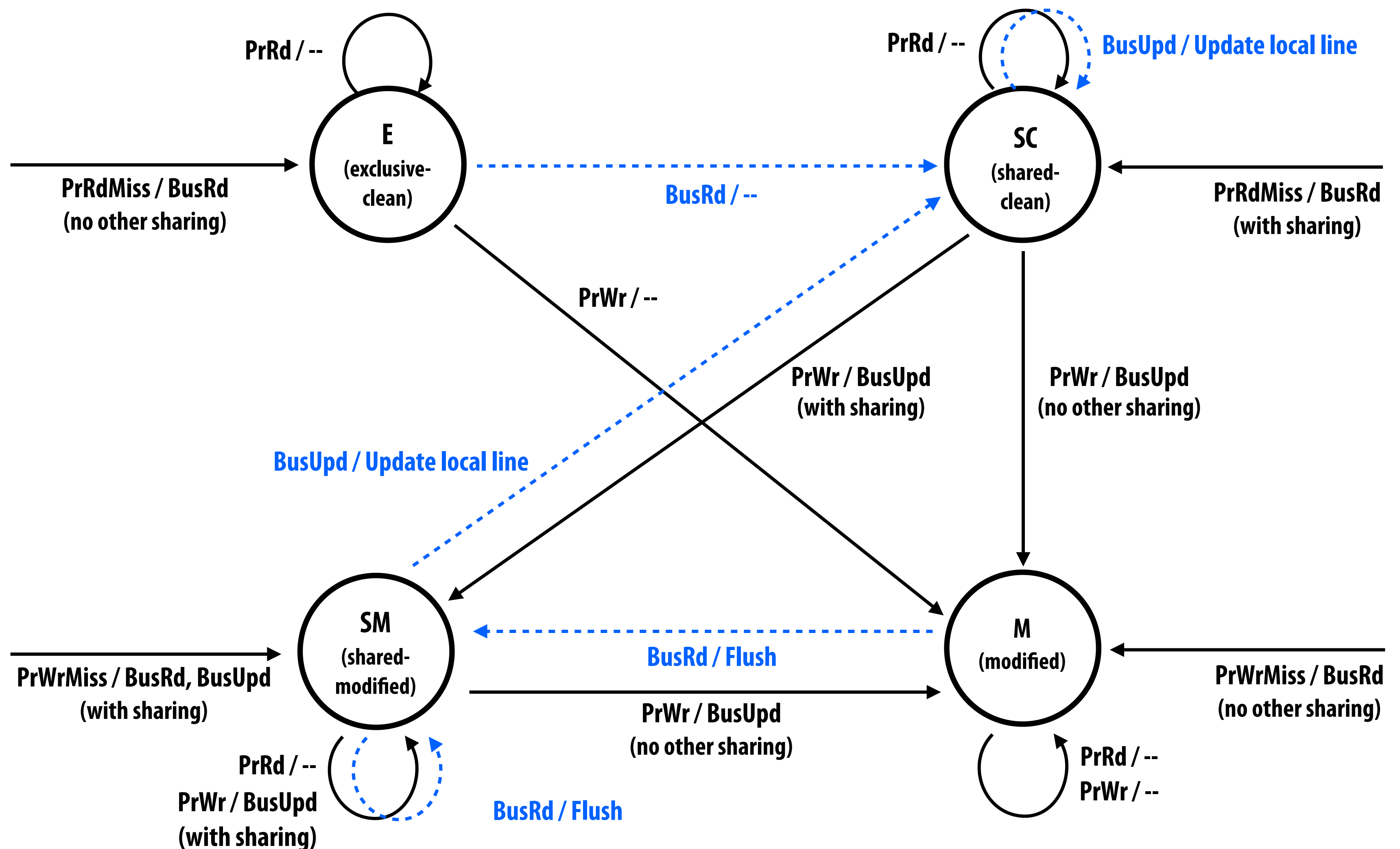
**Not shown: upon line replacement, cache must flush line to memory if line is in SM or M state
(Note: there's no invalid state here: why?)**

Invalidate vs. update

- **Which is better?**

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

- **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**
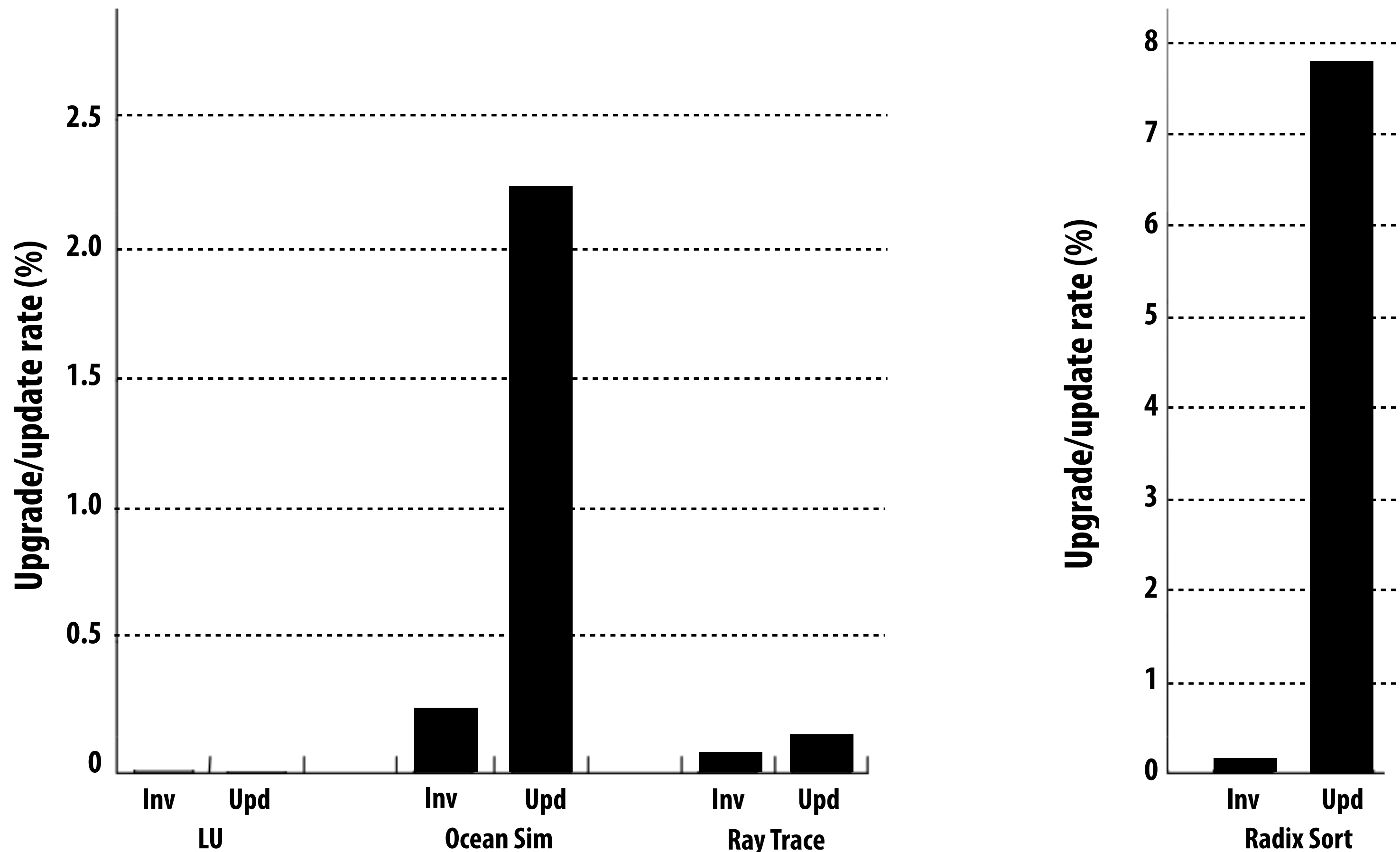
Invalidate vs. update evaluation: miss rate
Simulated 1 MB cache, 64-byte lines
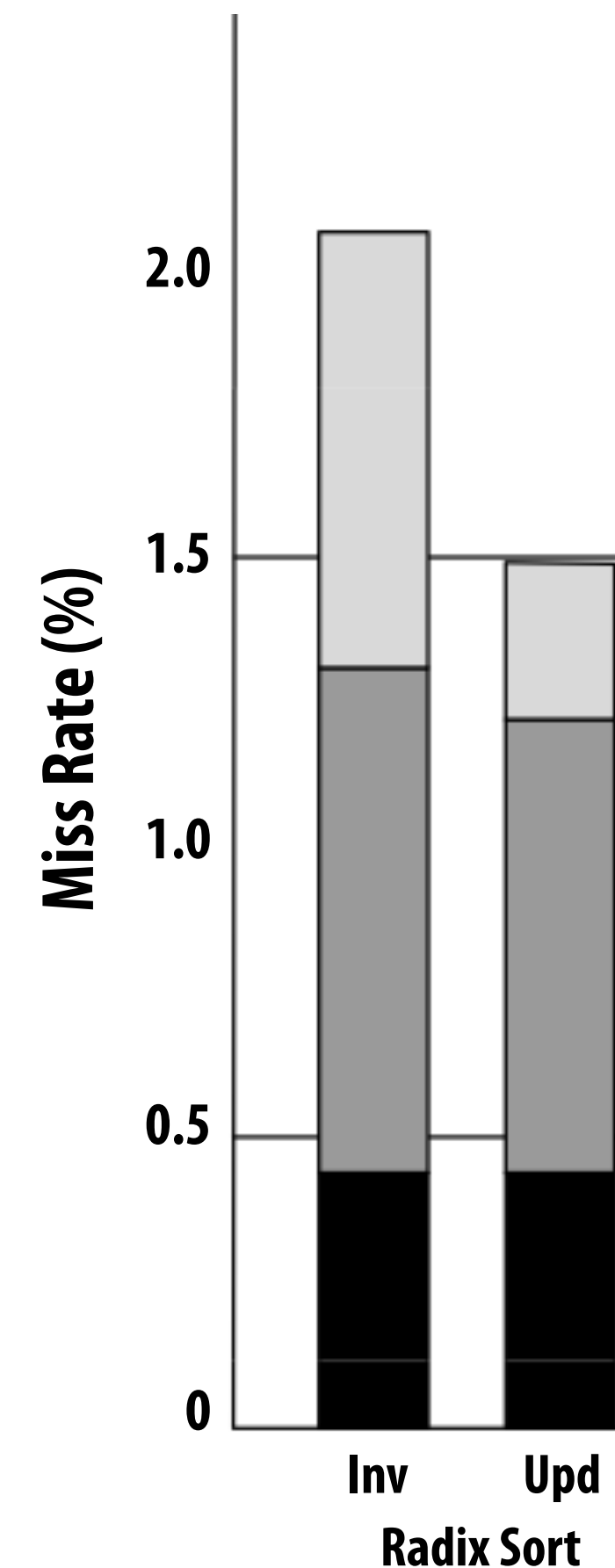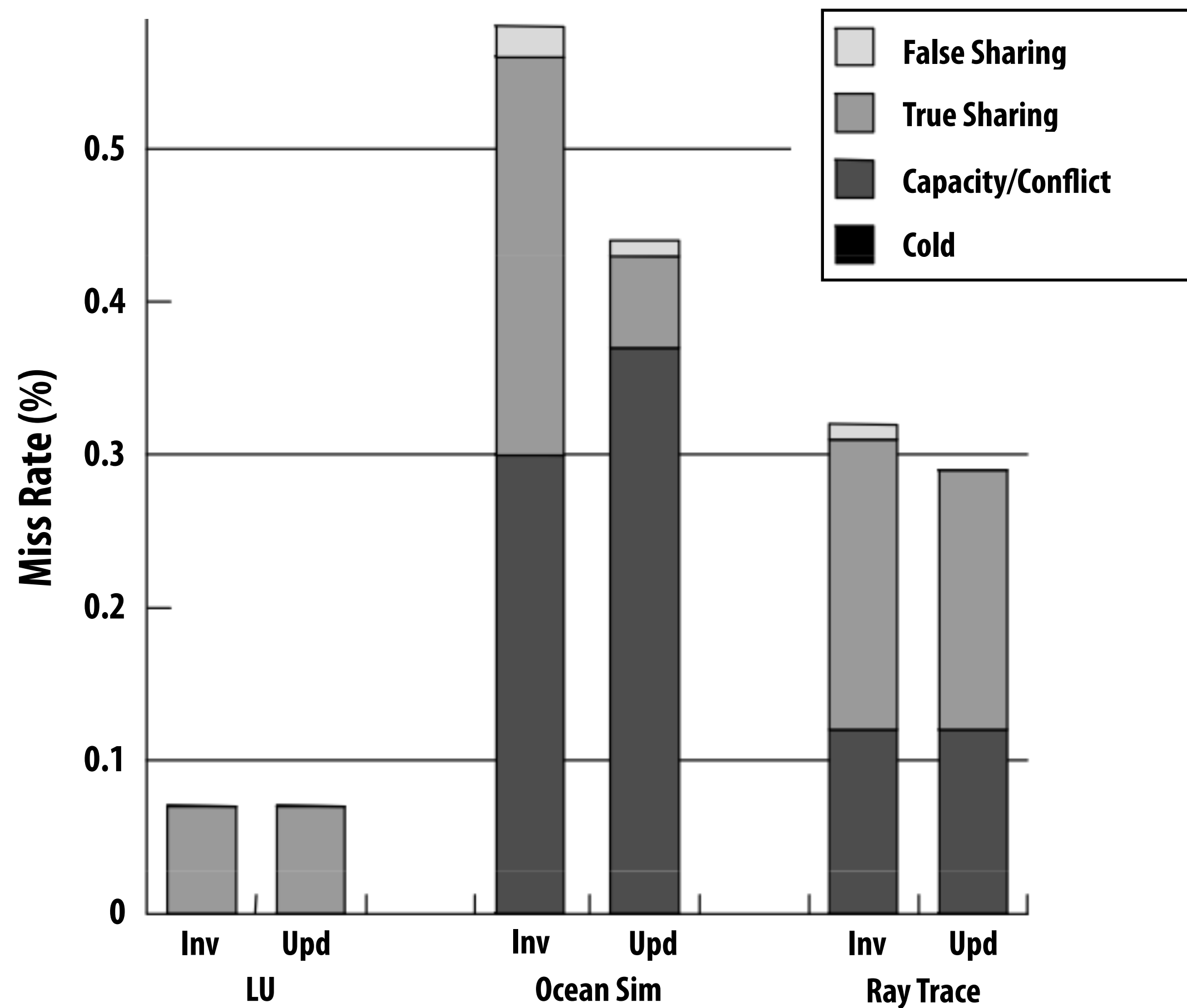
So... is update better?

Invalidate vs. update evaluation: traffic
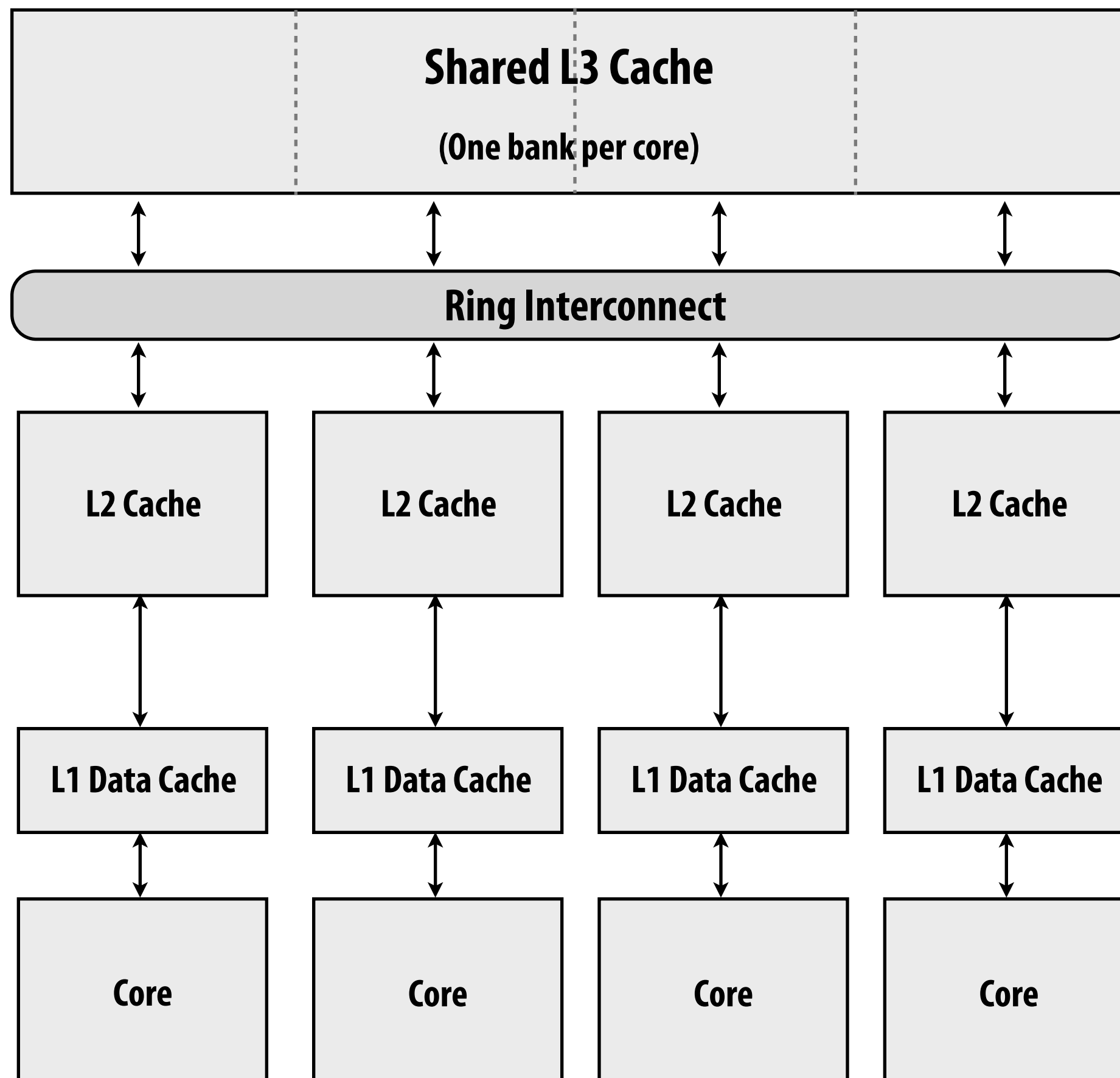## Simulated 1 MB cache, 64-byte lines

- 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

Reality: multi-level cache hierarchies

Recall Intel Core i7 hierarchy

	Shared L3 Cache		
	(One bank per core)		

Ring Interconnect

L2 Cache	L2 Cache	L2 Cache	L2 Cache

L1 Data Cache	L1 Data Cache	L1 Data Cache	L1 Data Cache

Core	Core	Core	Core

- **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

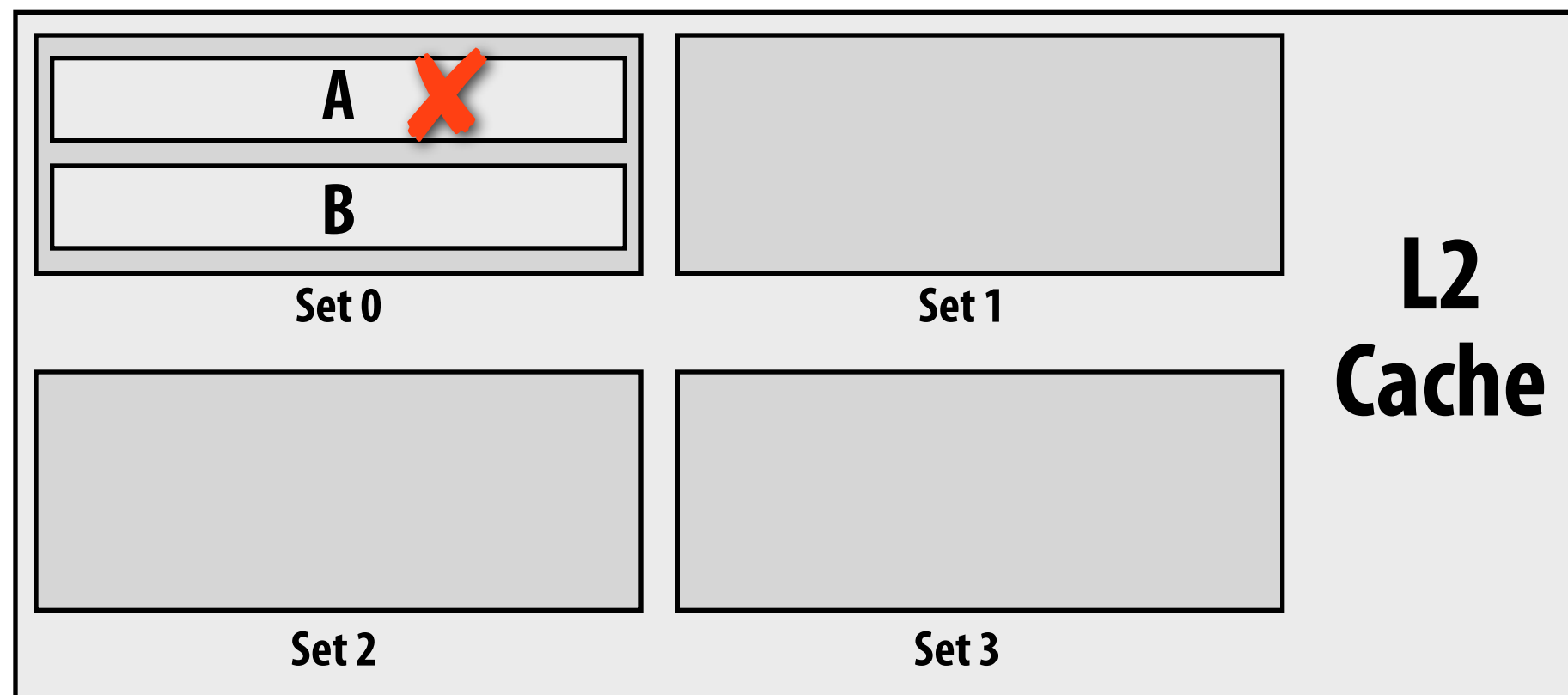# Is inclusion maintained automatically if L2 is larger than L1?   No!

- **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**
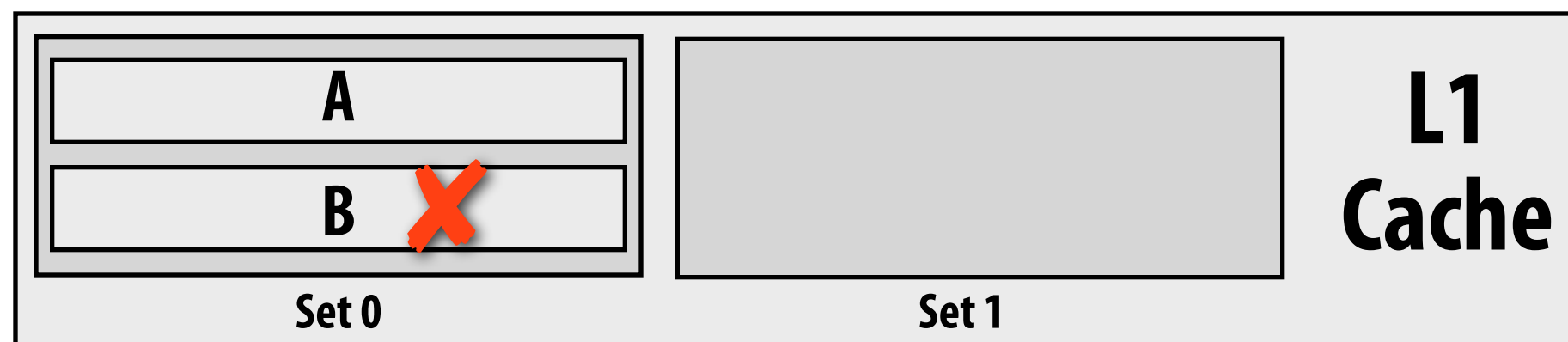
Processor accesses A (L1+L2 miss)
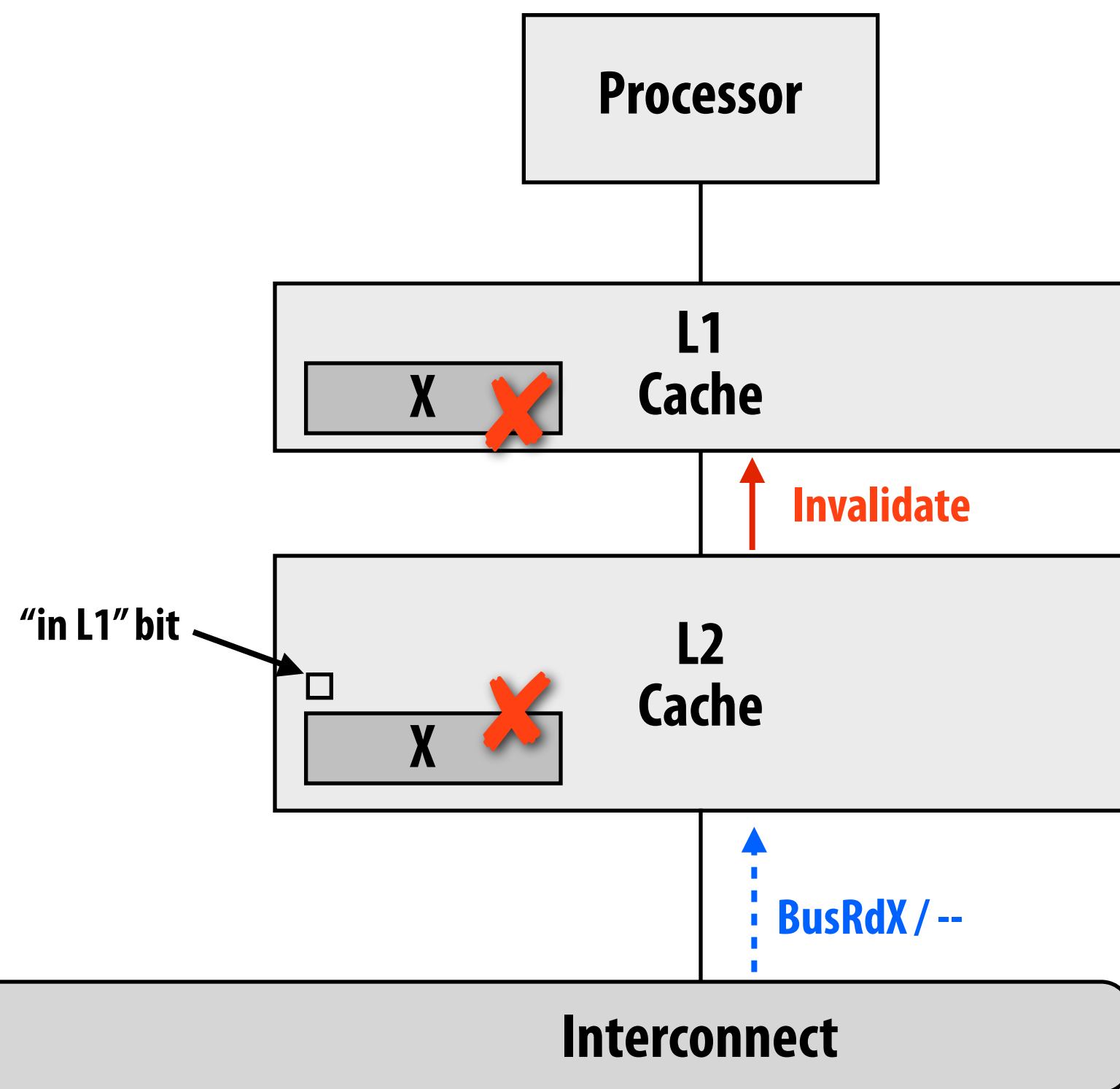
Processor accesses B (L1+L2 miss).

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 the access histories differ.

As a result, inclusion no longer holds!

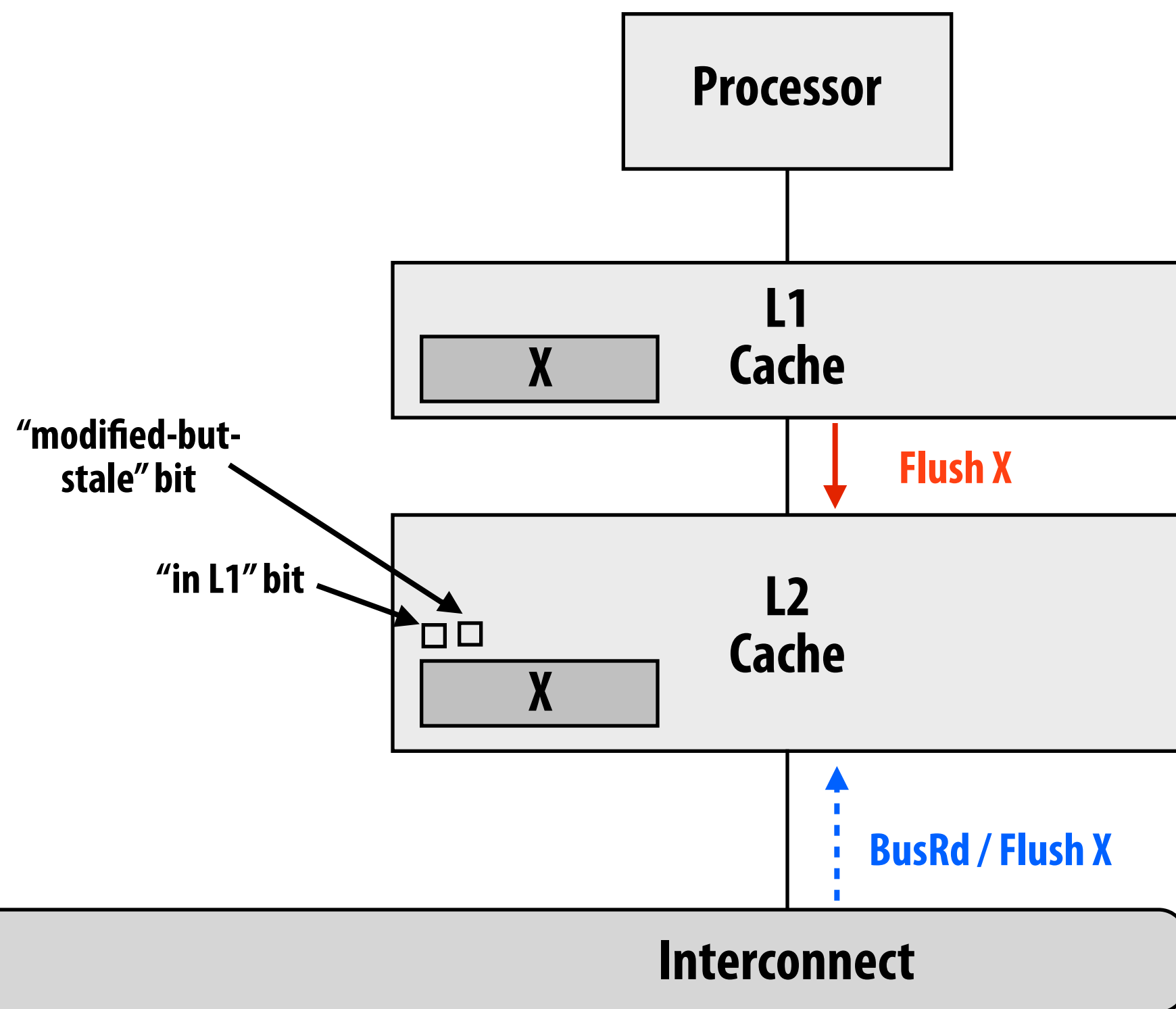# Maintaining inclusion: handling invalidations

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

Processor

L1 Cache

X

"modified-but-stale" bit

"in L1" bit

Flush X

L2 Cache

X

BusRd / Flush X

Interconnect

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 affect coherence are <u>broadcast</u> to all other caches**

    – **Analogy I've used often in class: all caches "shout"**

- **Caches listen ("snoop") for these messages, then 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!**

    - **Next time: scaling cache coherence via directory-based approaches**